USENIX Association

Proceedings of ICAC '14: 11th International Conference on Autonomic Computing

June 18–20, 2014
Philadelphia, PA

Conference Organizers

General Chair
Xiaoyun Zhu, *VMware*

Program Co-Chairs
Giuliano Casale, *Imperial College London*
Xiaohui (Helen) Gu, *North Carolina State University*

Program Vice-Chairs for Management of Big Data Systems
Karsten Schwan, *Georgia Institute of Technology*
Vanish Talwar, *HP Labs*

Program Vice-Chair for Self-Aware Cyber-Physical Systems
Ron Ambrosio, *IBM T.J. Watson Research Center*
Sokwoo Rhee, *NIST*

Poster Chair
Christopher Stewart, *The Ohio State University*

Publicity Chairs
Martina Maggio, *Lund University*
Ming Zhao, *Florida International University*

Program Committee
Tarek Abdelzaher, *University of Illinois at Urbana-Champaign*
Danilo Ardagna, *Politecnico di Milano*
Sara Bouchenak, *University of Grenoble*
Rick Buskens, *Google*
Rajkumar Buyya, *University of Melbourne*
Lydia Chen, *IBM Zurich*
Lucy Cherkasova, *HP Labs*
Ira Cohen, *HP Labs*
Paolo Costa, *Microsoft Research Cambridge*
Dilma Da Silva, *Qualcomm Research*
Elisabetta Di Nitto, *Politecnico di Milano*
Renato Figueiredo, *University of Florida*
Salima Hassas, *University of Lyon*
Yuxiong He, *Microsoft Research*
Tom Holvoet, *Katholieke Universiteit Leuven*
Longbo Huang, *Tsinghua University*
Alex Iosup, *Delft University of Technology*
Vana Kalogeraki, *Athens University of Economics and Business*
Evangelia Kalyvianaki, *City University London*
Yasuhiko Kanemasa, *Fujitsu Labs*
Jeff Kephart, *IBM T.J. Watson Research Center*
Fabio Kon, *University of Sao Paulo*
Samuel Kounev, *Karlsruhe Institute of Technology*
Diwakar Krishnamurthy, *University of Calgary*
Cristian Lumezanu, *NEC Labs*
Xiaosong Ma, *Qatar Research Foundation*
Julie McCann, *Imperial College London*
Daniel Menasche, *Federal University of Rio de Janeiro*
Arif Merchant, *Google*
Frank Mueller, *North Carolina State University*
Klara Nahrstedt, *University of Illinois at Urbana-Champaign*
Alma Riska, *EMC*
Kai Sachs, *SAP AG*
Hartmut Schmeck, *Karlsruhe Institute of Technology*
Onn Shehory, *IBM Research Haifa*
Kai Shen, *University of Rochester*
Prashant Shenoy, *University of Massachusetts Amherst*
Yasushi Shinjo, *University of Tsukuba*
Evgenia Smirni, *College of William and Mary*
Christopher Stewart, *The Ohio State University*
Jordi Torres, *Barcelona Supercomputing Center and UPC Barcelona Tech*
Mustafa Uysal, *VMware*
Timothy Wood, *George Washington University*
Ding Yuan, *Toronto University*
Ming Zhao, *Florida International University*
Xiaobo Zhou, *University of Colorado, Colorado Springs*

Tutorial Chair
Diwakar Krishnamurthy, *University of Calgary*

External Reviewers

David Aikema
Nipun Arora
Daniel Batista
Jorge Cabrera
Dazhao Cheng
David Fiala
Neha Gholka
Robin Givens
Nikolaus Huber
Jinho Hwang
Gregory Jean-Baptise
Sandeep Kandula
Palden Lama
Wenji Li
Douglas Otstott
Mauro Passacantando
Arash Rezaei
Michel Roger
Nikolas Roman Herbst
Kelly Rosa Braghetto
Piotr Rygielski
Zhikui Wang
Jiawei Wen
Yufeng Xin
Ji Xue
Feng Yan

ICAC '14:
11th International Conference on Autonomic Computing
June 18–20, 2014
Philadelphia, PA

Message from the Program Co-Chairs . . . . . **vii**

Wednesday, June 18, 2014

Model-Driven Management and Self-Adaptation

Storage Workload Isolation via Tier Warming: How Models Can Help **1**
Ji Xue and Feng Yan, *College of William and Mary;* Alma Riska, *EMC Corporation;* Evgenia Smirni, *College of William and Mary*

Model-driven Elasticity and DoS Attack Mitigation in Cloud Environments **13**
Cornel Barna and Mark Shtern, *York University;* Michael Smit, *Dalhousie University;* Hamoun Ghanbari and Marin Litoiu, *York University*

Integrating Adaptation Mechanisms Using Control Theory Centric Architecture Models: A Case Study . . . **25**
Filip Křikava, *University of Lille 1 and Inria;* Philippe Collet, *Université Nice Sophia Antipolis;* Romain Rouvoy, *University of Lille 1 and Inria*

Cloud Resource Management

ShuttleDB: Database-Aware Elasticity in the Cloud **33**
Sean Barker, *University of Massachusetts Amherst;* Yun Chi, *Square Inc.;* Hakan Hacıgümüş, *NEC Laboratories America;* Prashant Shenoy and Emmanuel Cecchet, *University of Massachusetts Amherst*

Matrix: Achieving Predictable Virtual Machine Performance in the Clouds **45**
Ron C. Chiang, *The George Washington University;* Jinho Hwang, *IBM T. J. Watson Research Center;* H. Howie Huang and Timothy Wood, *The George Washington University*

Adaptive, Model-driven Autoscaling for Cloud Applications **57**
Anshul Gandhi, Parijat Dube, Alexei Karve, Andrzej Kochut, and Li Zhang, *IBM Research*

Exploring Graph Analytics for Cloud Troubleshooting **65**
Chengwei Wang, Karsten Schwan, Brian Laub, Mukil Kesavan, and Ada Gavrilovska, *Georgia Institute of Technology*

Network and System Management

Inferring Origin Flow Patterns in Wi-Fi with Deep Learning **73**
Youngjune L. Gwon and H. T. Kung, *Harvard University*

Guarded Modules: Adaptively Extending the VMM's Privilege Into the Guest **85**
Kyle C. Hale and Peter A. Dinda, *Northwestern University*

Active Control of Memory for Java Virtual Machines and Applications **97**
Norman Bobroff, Peter Westerink, and Liana Fong, *IBM T. J. Watson Research Center*

Is Your Web Server Suffering from Undue Stress due to Duplicate Requests? **105**
Fahad A. Arshad, Amiya K. Maji, Sidharth Mudgal, and Saurabh Bagchi, *Purdue University*

Thursday, June 19, 2014

MDBS Track

A Model-Based Namespace Metadata Benchmark for HDFS 113
Cristina L. Abad, *Escuela Superior Politécnica del Litoral;* Yi Lu and Roy H. Campbell, *University of Illinois at Urbana–Champaign;* Nathan Roberts, *Yahoo, Inc*.

Towards Combining Online & Offline Management for Big Data Applications 121
Brian Laub, Chengwei Wang, Karsten Schwan, and Chad Huneycutt, *Georgia Institute of Technology*

An Enterprise Dynamic Thresholding System 129
Mazda A. Marvasti, Arnak V. Poghosyan, Ashot N. Harutyunyan, and Naira M. Grigoryan, *VMware, Inc*.

User-Centric Heterogeneity-Aware MapReduce Job Provisioning in the Public Cloud 137
Eric Pettijohn and Yanfei Guo, *University of Colorado, Colorado Springs;* Palden Lama, *University of Texas at San Antonio;* Xiaobo Zhou, *University of Colorado, Colorado Springs*

SCPS Track

Exploiting Temporal Diversity of Water Efficiency to Make Data Center Less "Thirsty". 145
Mohammad A. Islam, Kishwar Ahmed, Shaolei Ren, and Gang Quan, *Florida International University*

Real-time Edge Analytics for Cyber Physical Systems using Compression Rates 153
Sokratis Kartakis and Julie A. McCann, *Imperial College London*

Self-Optimizing Citizen-centric Mobile Urban Sensing Systems. 161
Usman Adeel, Shusen Yang, and Julie A. McCann, *Imperial College London*

Gait Recognition using Encodings with Flexible Similarity Metrics. 169
Michael B. Crouse, Kevin Chen, and H.T. Kung, *Harvard University*

Friday, June 20, 2014

Scheduling, Pricing, and Incentive

On-demand, Spot, or Both: Dynamic Resource Allocation for Executing Batch Jobs in the Cloud 177
Ishai Menache, *Microsoft Research;* Ohad Shamir, *Weizmann Institute;* Navendu Jain, *Microsoft Research*

Real-Time Scheduling of Skewed MapReduce Jobs in Heterogeneous Environments 189
Nikos Zacheilas and Vana Kalogeraki, *Athens University of Economics and Business*

Colocation Demand Response: Why Do I Turn Off My Servers? 201
Shaolei Ren and Mohammad A. Islam, *Florida International University*

Resource and Workload Management

Self-Tuning Intel Transactional Synchronization Extensions 209
Nuno Diegues and Paolo Romano, *INESC-ID and Instituto Superior Técnico, University of Lisbon*

CloudPowerCap: Integrating Power Budget and Resource Management across a Virtualized Server Cluster 221
Yong Fu, *Washington University in St. Louis;* Anne Holler, *VMware;* Chenyang Lu, *Washington University in St. Louis*

A Comprehensive Resource Management Solution for Web-based Systems 233
Filippo Seracini, Massimiliano Menarini, and Ingolf Krüger, *University of California, San Diego;* Luciano Baresi, Sam Guinea, and Giovanni Quattrocchi, *Politecnico di Milano*

PCP: A Generalized Approach to Optimizing Performance Under Power Constraints through Resource Management 241
Henry Hoffmann, *University of Chicago;* Martina Maggio, *Lund University*

Energy in Data Centers

Coordinating Liquid and Free Air Cooling with Workload Allocation for Data Center Power Minimization 249
Li Li, Wenli Zheng, Xiaodong Wang, and Xiaorui Wang, *The Ohio State University*

Managing Green Datacenters Powered by Hybrid Renewable Energy Systems 261
Chao Li, *University of Florida;* Rui Wang, *Beihang University;* Tao Li, *University of Florida;* Depei Qian, *Beihang University;* Jingling Yuan, *Wuhan University of Technology*

WattValet: Heterogenous Energy Storage Management in Data Centers for Improved Power Capping 273
Shen Li, Shaohan Hu, Shiguang Wang, Siyu Gu, Chenji Pan, and Tarek Abdelzaher, *University of Illinois at Urbana–Champaign*

Message from the ICAC '14 Program Co-Chairs

Welcome to Philadelphia, and to ICAC 2014, the 11th International Conference on Autonomic Computing! Since its inception, ICAC has attracted top quality papers in all areas of autonomic computing and continues to catalyze attention of researchers in this important area. This year's program includes a set of excellent contributions covering topics in model-driven management and self-adaptation, cloud-resource management, network and system management, scheduling and pricing, resource and workload management, and energy management in data centers.

As in past editions of the conference, the paper selection process was highly competitive. A total of 53 papers were submitted. All submissions were evaluated through a double-blind process. Papers were allocated four reviewers from program committee (PC) members not in conflict of interest with the authors. In some cases, additional reviews from experts outside the PC were solicited. The paper selection process culminated in the virtual PC meeting on April 7, 2014. Twelve full papers were selected for inclusion in the final program, leading to a competitive acceptance rate of 23%. Additionally, nine thought-provoking submissions were accepted as short papers, and two submissions were forwarded to the MDBS and SCPS special tracks. Following this selection process, a best paper award has been awarded to the best-ranked paper, namely "Self-Tuning Intel Transactional Synchronization Extensions" co-authored by Nuno Diegues and Paolo Romano; we congratulate the authors for their achievement.

The two special tracks, MDBS and SCPS, received in total 13 papers, 4 of which were accepted to each of the tracks, adding two additional sessions to the conference program. We also attempted to include both papers with mature results and thorough evaluations, as well as early-stage papers that propose new concepts/problems or reach into new areas. In addition, ICAC '14 will offer daily keynote speeches from Lucy Cherkasova, Ion Stoica, and Yuanyuan Zhou, a panel on Big Data Systems, and a joint poster session and reception with the USENIX Annual Technical Conference.

We are extremely grateful to the PC members and external reviewers for providing timely and thorough reviews, for exchanging a large volume of comments in the active online discussions and online PC meeting, and for their help in the shepherding process. We would also like to thank all the authors who submitted papers. Their efforts and enthusiasm are the true driving forces of ICAC.

We are very grateful to USENIX for sponsoring ICAC and hosting it as part of the Federated Conferences Week this year. The USENIX staff assisted us in a variety of tasks, including configuring and using the conference submission site, preparing the proceedings, advertising the conference, and last but not least, hosting the conference. Special thanks go to Xiaoyun Zhu, the general chair, and Jeff Kephart, the chair of the ICAC steering committee, for their constant help and support throughout the preparation of this edition of ICAC; the special track chairs for organizing MDBS and SCPS; to Martina Maggio and Ming Zhao for their activity as publicity chairs; to Christopher Stewart for his continuous efforts in organizing the joint poster session with ATC; and to Diwakar Krishnamurthy for selecting an excellent tutorial program.

We hope that you will find this year's program stimulating and highly innovative. Enjoy!

Giuliano Casale, *Imperial College London*
Xiaohui (Helen) Gu, *North Carolina State University*
ICAC '14 Program Co-Chairs

Storage Workload Isolation via Tier Warming: How Models Can Help

Ji Xue[1]*, Feng Yan*[1]*, Alma Riska*[2]*, and Evgenia Smirni*[1]

[1]College of William and Mary, Williamsburg, VA, USA, xuejimic,fyan,esmirni@cs.wm.edu

[2]EMC Corporation, Cambridge, MA, USA, alma.riska@emc.com

Abstract

Storage systems are often deployed in a tiered form to enable high performance and availability. These tiers utilize all possible volatile and non-volatile storage technologies, including DRAM, SSD, and HDD. The tradeoffs among their cost, features, and capabilities can make their effective integration into a single storage entity complex. Here, we propose an autonomic technique that learns user traffic patterns in a storage system over long time-scales to optimize user performance but also volume of completed system work. Our purpose is to multiplex as best as possible user workload with storage system features (e.g., voluminous internal system work) such that the latter is not starved but rather completed with minimal impact on user performance. Key to achieving the above is to use an autonomic learning engine to predict *when* the user workload intensity increases/decreases and then *proactively* stop/start bulky internal system work. Being proactive allows the system to effectively bring into the fast tier the active user working set just-in-time and right before it is needed most, i.e., when user traffic suddenly peaks. We illustrate the effectiveness of this mechanism by using both trace driven simulations from production systems as well experiments on a real testbed.

1 Introduction

As data storage technologies such as flash make their way into either enterprise storage [18, 11] or cloud storage [1], it is essential to integrate them with existing and (usually) slower, cheaper, and even vintage ones, e.g., hard disk storage, in order to strike a good balance among overall performance, availability, and cost [10, 16]. Following tradition in costs across the data path of a computer system, the fastest data tier, e.g., DRAM, is the most expensive, while the slowest tier, e.g., hard disk storage, is the least expensive per unit of data stored. In high-performing enterprise storage systems, it is common to find large DRAM caches, reaching as much as hundreds of GByte capacity, SSD caches reaching tens of TBs capacity, and a variety of high-end HDDs (15KRPM SAS HDDs) and low-end HDDs (7200RPM Nearline-SAS HDDs), all together exceeding the PByte-range of storage capacity. Given such a structure in the IO hierarchy, the expectation is that the bulk of user workload is served by the fast tiers, while slow tiers are used to persistently store the majority of data as well as improve on storage capacity and data reliability. The challenge lies in determining *what* portion of the voluminous data set to bring up to the smaller fast tiers and *when* to do that such that the benefits with regard to user performance and overall system operation are the highest.

Storage systems, both enterprise ones and those supporting web services, often receive the bulk of user traffic during business hours on weekdays. In addition, the storage system generates itself a considerable amount of internal traffic as a result of complex features that aim to enhance performance, reliability, availability, and integrity. Such system internal work includes, but is not limited to, making additional copies of the data off-site for added disaster recovery capabilities, snapshooting, deduplication, and policy compliance in a multi-tenant system. Recently, other sources of work have emerged in scaled-out storage systems such as virtual data analytics clusters that are brought up on demand to conduct analysis on large data sets without moving the data to a separate compute cluster, requiring effective interleaving of user and system workloads. In Figure 1, we show the average arrival intensity of user and system internal work over three days in a single data node of a large distributed storage cluster supporting a web application. During the day, the data node receives user requests that peak in intensity at around noon. During night, at regular intervals, the system generates its own work, which clearly is bulky and would have impacted user performance significantly if not scheduled during off-peak hours.

Because the system work shown in Figure 1 is the result of several features, it greatly surpasses in intensity all other user traffic. More importantly, its working set is (usually) much larger than the user working set. As a result, in a well balanced multi-tiered system, system work could negatively impact data placement policies which

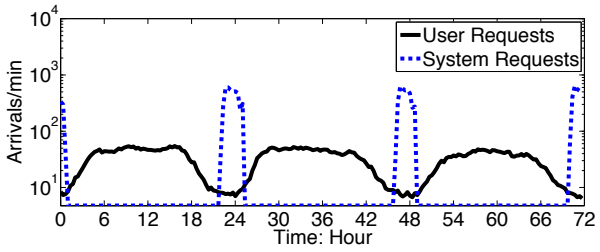


Figure 1: User and system arrival intensity (number of arrivals per minute) over a three day period.

ensure that the most active working set is in the highest performing tier. To schedule system work at regular intervals (e.g., at around midnight, as in Figure 1) does not necessarily guarantee good user performance (note that the figure illustrates only the arrival intensity, not work that needs to be done). Here, we contend that it is necessary to pair scheduling of system work with other system metrics measuring user activity and workload, such as utilization due to user-traffic, user traffic intensity, and fast storage tier hit rates, in order to improve overall system usage while ensuring that system work completes timely.

A critical difference between system internal work and user traffic is that the corresponding working sets are vastly different in both footprint and location within the storage system. Generally, the system working set is much larger than the user working set. As a result, in a multi-tiered storage system with tiers having different capacities and performance characteristics, standard efforts to isolate system and user workload in-time may still result in poor user performance. As the system transitions between system work to user work, high performing tiers could experience high user miss rates. Warming up the faster tier with the user working set is not instantaneous [25] and unless done proactively, performance degradation of user traffic can become unsurmountable.

In this paper, we propose an autonomic technique that learns the intensity patterns of user work within a probabilistic model over long time-scales, i.e., days and hours. The prediction of user intensity is paired with the knowledge of tier capacity, performance differences across tiers, and other metrics such as active user data set to derive a schedule for system work, i.e., *when* to start and stop it. Predicting user intensity patterns at large time scales can support

- proactively stopping system work *before* the user intensity increases and warming up the fast tier with the user working set,
- scheduling system work according to predicted user activity patterns such that large amounts of system work is completed with minimal impact on user performance, and
- avoiding instability due to short-lived, low user intensity that may erroneously initiate voluminous system work if short time-scale prediction or if reactive (i.e., feedback-based) scheduling is used.

Our methodology is light-weight and robust. Its benefits are evaluated via trace-driven simulation and actual experiments on a real test-bed. We do comparisons against feedback-based techniques that are usually applied in industry. Our experiments indicate that the larger the fast tiers and the larger the active user working set, the higher the benefits of having predictive models to schedule bulky system work.

This paper is organized as follows. Section 2 presents a workload characterization of user workload in a storage system supporting web data services. In Section 3, we present our predictive algorithm. Section 4 presents experiments on a real system and trace driven simulations that demonstrate the effectiveness of our technique. Section 5 discusses related work. We conclude in Section 6.

2 Trace Overview and Motivation

In this section, we present a set of production traces that describe how a storage system is utilized by a large scale web application. The traces contain the user IO intensity (in IO requests per second) in a scale-out storage back-end of a mid-size web service provider[1]. The web service has multiple locations that serve the user workload based on geography. As a result, in each location the workload intensity follows well the day/night pattern (working hours vs. non-working hours) as well as weekday/weekend patterns. Here we focus on the traffic received by a single data node. However, because of the load balancing in the storage system, the behavior observed in a single node persists across all other data nodes in the cluster.

Figures 2 and 3 show the average arrival intensity of user requests per minute averaged over 10 minutes and 1 hour intervals for 10 days and 35 days periods, respectively. There is a clear daily and weekly pattern in the workload intensity. This is expected since nowadays large scale web services, although available 24/7 worldwide, are deployed in geographically distributed data enters, resulting in clear day-and-night patterns in each of the available locations. Similar patterns are seen also in enterprise storage, which although different in nature from web storage, serves heavy traffic during business hours and much less during night hours. These patterns suggest opportunities for predicting user traffic intensity. The ability to predict these drastic changes can be used to prepare the system proactively for the heavy user workload, for example by moving the active user data set to the fast tier *before* it starts being accessed. Effective prediction should also ensure that the system schedules long and resource demanding internal work only when it is safely predicted that the system is to enter a long period of low utilization.

[1]Due of confidentiality agreements, the traces or provider details can not be made publicly available.

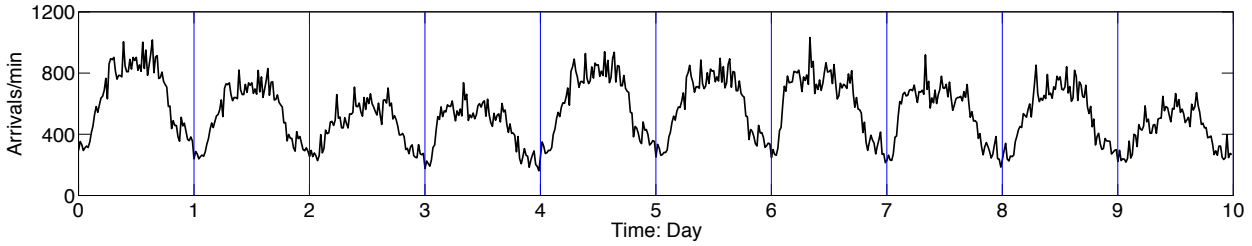


Figure 2: IO request arrival intensity (number of arrivals per 10 minutes) over 10 days.

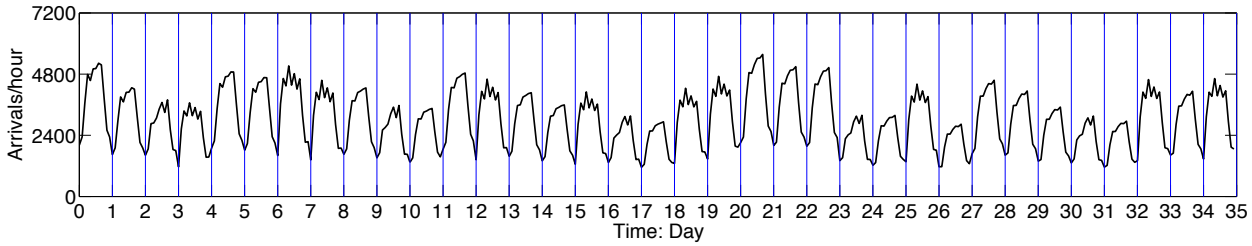


Figure 3: IO request arrival intensity (number of arrivals per hour) over 35 days.

We plot the empirical density of the user arrival rate at a granularity of a minute in Figure 4 for all 35 days. The graph illustrates a clear bi-modal pattern, which confirms that the arrival intensity changes between two general states that we roughly classify as high/low. In the next section, we show how we incorporate the stochastic characteristics of the user arrivals to derive a model that predicts the duration of each high and low intensity period.

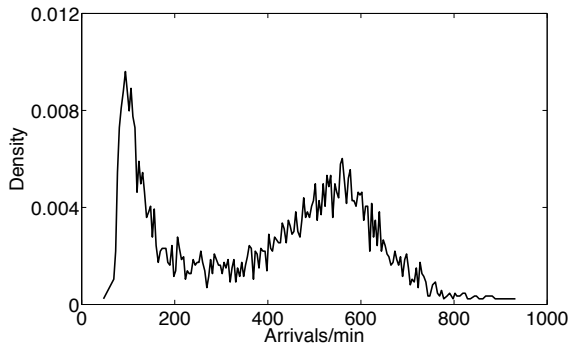


Figure 4: Empirical density of the arrival rate of user requests.

Prediction of low/high intensity periods provides the system with the information it needs to intelligently interleave user workload with voluminous system internal work. For example, it can proactively stop the system work and warm up the fast tier storage just before the high user intensity period so that the majority of user traffic is served by the fast tier rather than the slow tier. As discussed later in the paper, we do not determine here what data to bring but rather when to bring them in the fast tier. Determining the user working set (i.e., what to bring) is outside the scope of this paper.

To illustrate the benefits of predicting arrivals of high and low intensity periods and motivate our work, we evaluate three scenarios on handling system work in a data node:

- *user only*: no system work is interleaved with user traffic,

- *reactive*: system work runs only during low user utilization periods; when user high utilization is detected, the system work is stopped and reactively the fast tier cache is warmed up with active user data,

- *proactive with future knowledge*: system work runs only during low user utilization periods; since we know *a priori* when user high utilization starts, we stop system work early enough to allow for the fast tier cache to be warmed up with active user data right before the surge of user work.

Figure 5 illustrates the CDF of user response time when a single day of trace data (see Figure 3) is used to drive a simulation of the above three scenarios. For the two policies that allow system work, the simulation starts and stops it at the same time, with the purpose of evaluating only the benefit of proactive vs. reactive fast tier warm up (which takes the same amount of time in both scenarios). Clearly, with a reactive warm up a large portion of user requests experience long response time by being served from the slow tier of the system. Both the body and the tail of the user response time distribution benefit greatly by a proactive fast tier warm up, which can be possible only if a model enables prediction of arrival of high user utilization periods.

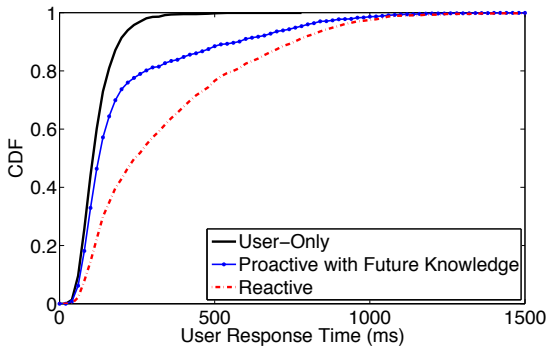


Figure 5: CDFs of user response time of day 20 for different algorithms.

3 Model-based Storage Tiering

The data patterns described in Section 2 (see Figures 2 and 3) favor prediction. User intensities go through a clear high/low pattern which if captured accurately can be used to intelligently interleave workloads with widely different demands. Indeed, this is the current state of the practice in most data centers [6], a practice that is also illustrated in Figure 1. Yet, rather than schedule system work conservatively as in Figure 1, we aim to develop a model that would allow for better overall system resource utilization by completing aggressively system work and achieve better performance isolation across user and system workloads.

We first present a Markovian-based model that captures the duration of low/high traffic intensities in user arrivals across different time scales (i.e., daily distinguishing between weekday/weekend and hourly distinguishing between day/night activity). We also develop a model that captures the changes in user performance as a function of fast tier hit rate. Finally, we apply these models to predict when such periods of high/low intensities arrive to schedule system work and cache warm up with the goal of optimizing performance.

3.1 Traffic Intensity Prediction Model

The preliminary workload analysis in Section 2 shows that there are repeatable low/high daily intensity patterns. We refer to the state with high average arrival intensity as the *High* state and the lower one as the *Low* state. The threshold for distinguishing the *High* and *Low* states can be discovered via statistical analysis. Alternatively, the threshold could be user-defined. We need to determine the following: how long does the user traffic resides in each state, i.e., the *duration* of each state and the conditional probability that there is a transition from one state to the next. The analysis of Section 2 also shows that weekend *High* state intensity is different from weekday *High* state intensity. We aim to capture these patterns in order to distinguish days with overall less intensity from days of higher intensity.

To capture this effect, we use a hierarchical model that captures different *types* or *classes* of high/low intensities. The difference between these is that the average intensity for the *High* or *Low* states may be different as well as the duration of each state. Note that in addition to transitions within the *High*/*Low* states within each class, there are probabilistic transitions from the states that represent one class (e.g., weekdays) to another class (e.g., weekend or holidays). This hierarchical model is shown in Figure 6. The model in Figure 6 has two classes of high/low intensities (capturing day/night and weekday/weekend patterns). However if more classes are detected then the hierarchy of the model can grow to accommodate them.

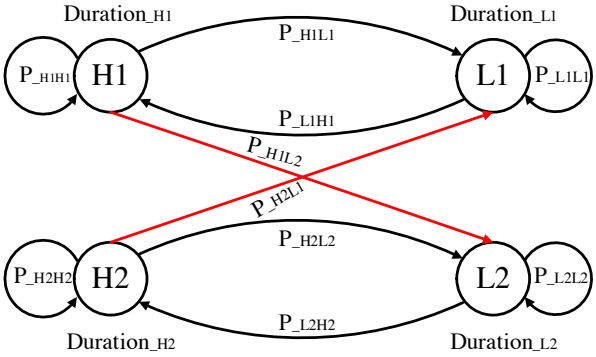


Figure 6: High level Markovian model.

We start building the model by first categorizing the observed arrival intensities. We use clustering to determine how many types of low/high intensities exist in the workload using *Silhouette* [19] and *K-means*. *Silhouette* is used to calculate the dissimilarity value $s(i)$ of the average arrival intensity of day i. The dissimilarity value $s(i)$ is defined as:

$$s(i) = \frac{b(i) - a(i)}{max\{a(i), b(i)\}},$$

where i is the day index, $a(i)$ is the average dissimilarity of day i to all other days within the same cluster, and $b(i)$ is the lowest average dissimilarity of day i to all days in a different cluster. Distance measures are the most common for calculating the dissimilarity values $a(i)$ and $b(i)$. The values of $s(i)$ are in $[-1,1]$ and the larger its value the better, e.g., when $s(i)$ approaches to 1, $a(i) \ll b(i)$, which means that the distance between data within each cluster is the smallest. More specifically, the following three steps are performed to determine the number of clusters in the model:

1. Define the upper bound of the number of clusters as $\sqrt{\frac{n}{2}}$, where n is the total number of days in the historical information[2];
2. Calculate the average $s(i)$ for a different number of clusters;

[2] $\sqrt{\frac{n}{2}}$ is used as a rule of thumb in *K-means* to avoid too many clusters and unnecessary overheads [13].

3. Choose the number of clusters with the highest $s(i)$.

After the number of clusters is determined, we calculate the transition probabilities between them. We also estimate the duration of *High*/*Low* states within each cluster as well as their transition probabilities.

In a live system, the goal is to have an initial model built with the data collected over the first few weeks of operations. Then, the model is updated continuously as new data on user workload is collected, so that any changes in system operation and user access patterns are reflected in the model.

Figure 7 illustrates the effectiveness of prediction by comparing it with actual state changes. The dashed lines illustrate the points where the model detects a change in the state (from *High* to *Low* or *Low* to *High*). The dotted line illustrates the actual state changes. The graph shows that the model predicts effectively changes from one user intensity state to the next.

3.2 Fast Tier Hit Rate

The fast tier hit rate in a storage system is related to many factors, including its capacity and active user working set. Here, we provide an estimation method for the instant fast tier hit rate with the goal of estimating how it changes as active user data moves from the slow tier up to the fast tier and vice versa.

As we focus mostly on large capacity fast tiers as well as large active data sets, it becomes necessary to warm up the fast tier cache rather than allow it to be warmed up gradually by the user accesses. Figure 5 clearly illustrates that warming up the cache can tremendously affect performance.[3]

The average IO service rate for user traffic is a combination of fast storage tier access speed and slow storage tier access speed and can be expressed as follows:

$$\mu(t) = (1+S(t)) * \mu_{origi} = H * R_{fast} + (1-H) * R_{slow}, \quad (1)$$

where $\mu(t)$ is the average service rate of user traffic at time t. μ_{origi} is the original average service rate of user traffic, e.g., when there is no system work. $S(t)$ is the service slowdown which describes how the average service rate changes from the original one. H is the fast tier hit rate, R_{slow} is the average slow storage tier access speed and R_{fast} is the average fast storage tier access speed, implying that the fast tier hit rate can be defined as follows:

$$H = \frac{(1+S(t)) * \mu_{origi} - R_{slow}}{R_{fast} - R_{slow}} \quad (2)$$

with $0 \leq H \leq 100\%$.

[3]The model presented here can be trivially extended to capture the no warm up case, i.e., passively move data when first accessed, by changing the parameter of the average transfer speed between the fast storage tier and slow storage tier to a function that is determined by the intensity of arrivals.

When system work is served, the average service rate of user traffic unavoidably decreases due to sharing of the fast storage tier with the working set of the system workload. We assume that the Service Slowdown increases linearly over time during the periods of serving system work:

$$S(t) = S(t_{i-START}) + a * t, \quad (3)$$

where $S(t_{i-START})$ is the Service Slowdown at the beginning of the time window i serving the additional work. This parameter is necessary because the slowdown effects may propagate through several windows of time. Finally, a is an coefficient that describes how fast the slowdown increases during system work serving periods.

The maximum slowdown occurs when the fast storage tier is filled with the system working set, and unavoidably all user traffic is served from the slow storage tier, therefore

$$R_{slow} = (1+S_{max}) * \mu_{origi}, \quad (4)$$

or

$$S_{max} = \frac{R_{slow}}{\mu_{origi}} - 1. \quad (5)$$

Note that μ_{origi} may not equal to R_{fast} because the Fast Tier Hit Rate may not equal to 100% even when no system work is served.

When $S(t_{i-START}) = 0$, i.e., when the fast storage tier is filled with the user working set, T_{fast} expresses the period before system work data starts occupying the entire fast storage tier. After the user working set is removed from the fast storage tier, the user service slowdown reaches its maximum, i.e., no user IO requests can use the fast storage tier to improve performance. According to Eq. 3, we have:

$$S_{max} = a * T_{fast}. \quad (6)$$

By definition, the capacity C equals to the transfer speed F multiplied by time, therefore:

$$C = F * T_{fast} \quad (7)$$

and

$$\mu(t) = (1+S(t)) * \mu_{origi}. \quad (8)$$

From Eq. 3 and Eq. 5 – Eq. 7, when $t > t_{i-END}$, we have :

$$\begin{aligned} S(t) &= S(t_{i-START}) + \tfrac{S_{max}*F}{C} * t. \\ &= S(t_{i-START}) + \tfrac{F}{C} * (\tfrac{R_{slow}}{\mu_{origi}} - 1) * t, \end{aligned} \quad (9)$$

which shows that the user service slowdown is related to the average fast and slow storage tier access speed, the average transfer speed between fast and slow storage tiers, the original average service rate of user traffic, and the fast tier capacity.

When the system stops serving system work, the user service slowdown due to sharing of the fast tier with system workload decreases over time. We assume that this decrease is linear across time:

$$S(t) = S(t_{i-END}) - b * t. \quad (10)$$

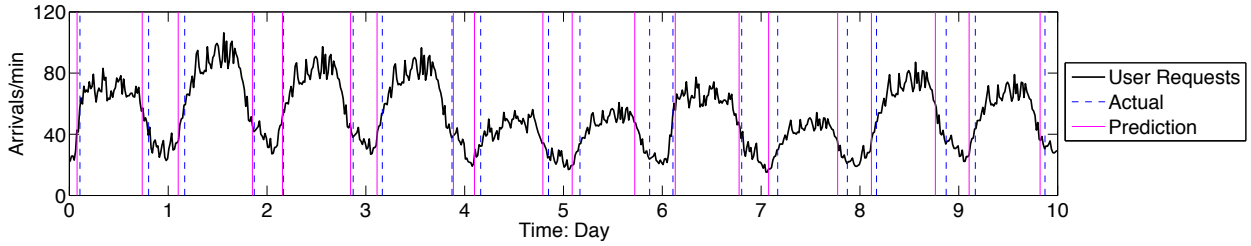


Figure 7: Comparison of actual and predicted arrival intensity state changes.

$S(t_{i-END})$ is the user service slowdown at the end time of system work serving window i and b is an coefficient that describes how quickly the slowdown decreases over non-system work serving periods.

Similarly, when $S(t_{i-END}) = S_{max}$, i.e., when the fast tier is filled with all system work data, it takes T_{fast} time units before the user working set refills the entire fast tier. After the user working set is restored, the user service slowdown reaches its minimum, i.e., to the original service rate without any system work. According to Eq. 10, when $S(t_{i-END}) = S_{max}, S(t) = 0$:

$$0 = S_{max} - b * T_{fast}. \tag{11}$$

By comparing Eq. 3 and Eq. 10, we have $b = a$. Therefore, for $t_{i-START} \leq t \leq t_{i-END}$, we have:

$$\begin{aligned} S(t) &= S(t_{i-START}) - \tfrac{S_{max}*F}{C} * t \\ &= S(t_{i-START}) - \tfrac{F}{C} * (\tfrac{R_{slow}}{\mu_{origi}} - 1) * t. \end{aligned} \tag{12}$$

Using Eq. 9 and Eq. 12

$$S(t) = \begin{cases} S(t_{i-START}) - \frac{F}{C} * (\frac{R_{slow}}{\mu_{origi}} - 1) * t, \\ \qquad \text{for } t_{i-START} \leq t \leq t_{i-END}, \\ S(t_{i-END}) + \frac{F}{C} * (\frac{R_{slow}}{\mu_{origi}} - 1) * t, \\ \qquad \text{for } t > t_{i-END}. \end{cases} \tag{13}$$

From Eq. 2 and Eq. 13, it follows that the hit rate is:

$$H = \begin{cases} \frac{\mu_{origi}(S(t_{i-START})+1) - R_{slow} - \frac{F}{C}*(R_{slow}-\mu_{origi})*t}{R_{fast}-R_{slow}}, \\ \qquad \text{for } t_{i-START} \leq t \leq t_{i-END}, \\ \frac{\mu_{origi}(S(t_{i-START})+1) - R_{slow} + \frac{F}{C}*(R_{slow}-\mu_{origi})*t}{R_{fast}-R_{slow}}, \\ \qquad \text{for } t > t_{i-END}. \end{cases} \tag{14}$$

1. if system in *characterization state* do
- **a.** collect arrival intensity information and use *Silhouette* and *K-means* to do clustering.
- **b.** Compute duration of *High*/*Low* states per cluster and transition probabilities within and across clusters
- **c.** Build the Markovian model with the computed parameters and continue updating the model while the system is in operation

2. if the system is in *serving system work* state do
- **a.** Predict how long the system will be in *Low* state and compute *warmup time* for fast storage tier T_{fast}
- **b. if** residual time in *Low* state $> T_{fast}$
  - **i.** compute the Fast Tier Hit Rate and average utilization $UTIL_{past}$ of past t minutes
  - **ii. if** no outstanding user request and $H >= H^{low}_{threshold-lower}$ and $UTIL_{past} <= UTIL_{threshold}$ schedule system work
  - **iii. else if** $H <= H^{low}_{threshold-lower}$, stop serving system work until $H = H^{low}_{threshold-upper}$ go to Step **2.b.iv**
  - **iv. else** process user request or stay idle
  - **v.** go to Step **2.b**
- **c. else if** the residual time in *Low* state $< T_{fast}$
  - **i.** stop serving system work and warm up the fast tier
  - **ii.** go to Step **2.b**
- **d. else if** system in high arrival intensity state
  - **i. if** no outstanding user request and $H >= H^{high}_{threshold-lower}$ and $UTIL_{past} <= UTIL_{threshold}$ schedule system work
  - **ii. else if** $H <= H^{high}_{threshold-lower}$, stop serving system work until $H = H^{high}_{threshold-upper}$ go to Step **2.d.iii**
  - **iii. else** process user request or stay idle
  - **iv.** go to Step **2.b**

go to Step **1**

Figure 8: Prediction-based deployment of systems work.

3.3 Storage Tiering

Figure 8 presents an algorithm for scheduling system work in a multi-tier storage system. In the *characterization state*, the algorithm collects arrival intensity information to compute the parameters and build the Markovian model. Based on the *Low* and *High* states duration and the fast tier warm up time, the algorithm schedules system work. For example, during the *Low* state, the system work is served concurrently with the low user traffic because the overall performance impact is small. The thresholds of the fast tier hit rate can be considered as a control knob. For example, the thresh-

olds of *Low* state can be set much smaller than the threshold of those of the *High* state so that more system work can be finished.

The algorithm proactively warms up the fast tier by stopping system work ahead of the predicted arrival of the *High* state. The warm up time depends on the fast tier capacity and can be computed via Eq. 7. Such proactive action is critical because the fast tier can not be warmed up instantly. For large fast tiers, the warm up may take a long time, hours or in some cases even days [25]. Without proactive warm up, the user requests that arrive in the initial period of the *High* state are to be impacted significantly.

4 Experimental Evaluation

In this section, we evaluate the proposed scheduling framework via an extensive set of experiments in a real system and through trace-driven simulations. We first describe the testbed and the workload we use in Section 4.1 and then show the respective experimental results that validate our method in Section 4.1.1. Then we use our traces from Section 2 to drive a set of simulation experiments for more sensitivity analysis of our predictive model. Throughout this section we compare our framework with other common practices such as feedback-based techniques.

4.1 Experimental Testbed and Workloads

Our testbed consists of a server with a disk enclosure attached to it, which provides data services to a host. Its memory is 12GB and the disk enclosure has 12 SATA 7200RPM HDDs of 3TB each. In our experiments the system memory emulates the fast tier and the disk enclosure the slow tier used for the bulk of the data. The benefits of effective workload prediction are high for system with large gaps in the performance characteristics across tiers. For the shake of presentation clarity, we evaluate here the predictive framework on a system with two-tiers only that provide services that differ by one order of magnitude. We stress that our approach can be directly applied in a system with *any* number of storage tiers.

The workload is generated and measured at the host machine. We use `fio` [2] as the IO workload generator for the flexibility it provides to generate a wide range of IO workload intensities and general patterns. We generate two types of IO workloads, user and system, which differ on the active working set size rather than their access pattern. The working set size for the user workload is 1GB[4], i.e., such that it always fits into the memory of the server that emulates the fast tier. The system work has an active working set of 24GB, i.e., it does not fit fully into the fast tier and the large slow tier is accessed to retrieve the data. The access pattern for both user and system workload is 100% small random reads to emulate common enterprise workloads that would benefit from prefetching (warm-up) only if the working set can fully (or almost) fit in the high performing tier (i.e., the SSD).

Our framework determines only *when* to warm up the cache with a pre-determined user data set. Determining what data should be brought into the cache is outside the scope of this paper. The user active working set can be determined by evaluating statistically access patterns such as the number of accesses per storage location. Here we also assume that the system is provisioned in such a way that the fast tier can fit the entire (or the majority of the) user active working set. The fast tier is warmed up via a sequential read of the user working set.

We measure the following system work scheduling policies:

- *user-only* - used only as a baseline to evaluate the impact of the additional system work,
- *feedback-based* - a reactive policy that monitors the current load intensity in the system and determines if it is in a high or low intensity period,
- *prediction-based* - a proactive policy (see Figure 8) that uses the proposed Markovian model to predict user traffic intensity by having learned from past data the duration of periods of high and low intensity.

Rules that determine the change of state (from *High* to *Low* or vice versa) for both the feedback-based and the prediction-based policy are the same and follow the discussion in Section 3. The main difference is that by predicting the arrival of the *High* state the system can prefetch the user working set before the state changes and avoid performance penalties in a large portion of user requests. The feedback-based policy is a reactive one: it acts *after* it detects a state change. As a result, user requests arriving right after the state change suffer from performance penalty of being served at the slow tier, till the fast tier is warmed up. The larger the fast tier, the longer it takes to warm it up and the higher the performance penalty of the feedback-based policy.

The prediction-based policy is designed to fall-back on the feedback-based policy: if the prediction time for a *High* state is in the future but the *High* state is already detected, then system work is stopped and the fast tier is warmed-up with the working set reactively.

4.1.1 Measurement Results

Using `fio`, we generate a random reads workload accessing data stored in our server. The intensity of user IO requests is shown in Figure 9 and it emulates very closely the load pattern of user requests shown in Section 2. Note that without any system work, the response time of user IO requests remains in the same range of about 150ms. All IOs are served from the fast tier. The user throughput however does increase by one order of magnitude as the arrival intensity increases. This confirms that the storage

[4] Experiments with 4GB and 8GB user working sets yielded similar results and are omitted here due to lack of space.

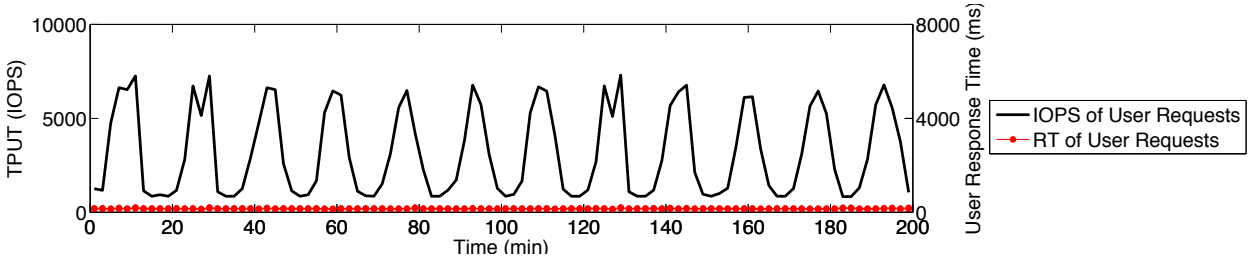


Figure 9: User IOPS (throughput) and user response time over time, *user-only* policy.

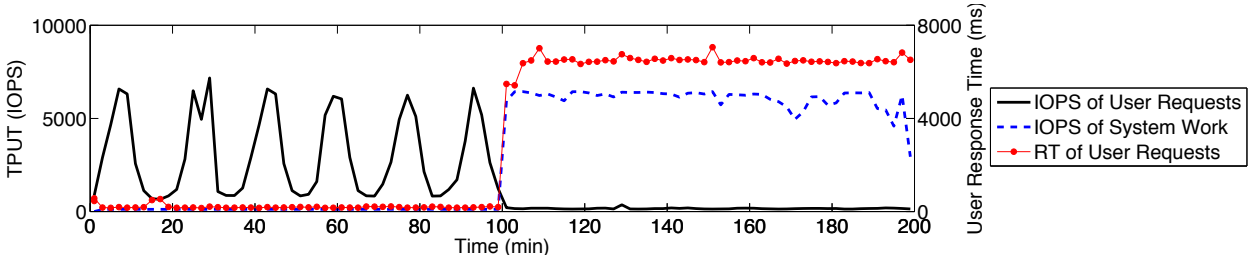


Figure 10: User and system IOPS (throughput) and user response time over time. In the first half of the experiment there is minimal system work, in the second half system work is increased by two orders of magnitude.

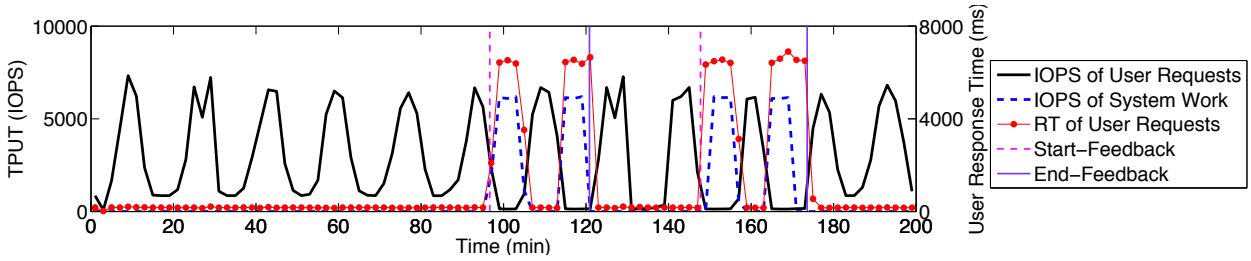


Figure 11: User and system IOPS (throughput) and user response time over time. Till the 100th minute, there is only user workload. In the time periods from the 100th to the 120th minute and the 150th to the 170th minute, the feedback policy is launched. The prediction policy is launched from the 120th to the 150th minute, as well as after the 170th minute to the end of the experiment.

system does not suffer from queuing delays and it has the capacity to sustain more user load.

We add on the same experiment some system work. Initially, the system work is slowed down as to not interfere with the user workload performance. In Figure 10, we show the same user workload but interleaved with system work with very low intensity. For the first 100 minutes, the system throughput reaches up to 121 IOPS. In the next 100 minutes, the intensity of system work increases and its throughput reaches 5968 IOPS, a two-orders of magnitude increase from the first half of the experiment. User performance is not impacted in the first half of the experiment, but system work here is minimal. The figure plots the throughput of both user and system work, as well as the response time of the user workload. In the second part of the experiment (100 min to 200 min) where system work is launched, the increase in user response time grows by two orders of magnitude, while its throughput is very low.

In Figure 11 we do the same experiment but we now activate the feedback and the prediction-based policies in the second half of the experiment, when the system work is launched. The feedback policy is used from the 100th to the 120th minute as well as for the time period between the 150th to the 170th minute. In the rest of the time periods, the prediction-based policy is used. The graph shows that when the prediction policy is activated (i.e., in time periods: 120-150, and 170-200), user performance remains unscathed, both with respect to throughput and response time. In the time periods when the feedback policy is used, there is high throughput of system work but also user response times that are orders of magnitude higher than the user-only case.

What makes the difference in user performance between the feedback and prediction-based policies is the timely fast-tier warm up. Figure 12 captures this effect. In this experiment, we use a very small data set of 1GB and let the fast tier warm up 1) by the accesses of the

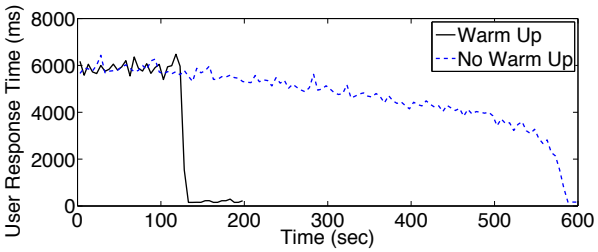


Figure 12: Response time with warm up and without warm up across the experiment time.

regular user workload (i.e., no explicit warm up) and 2) by specifically bringing the working set up to the fast tier (i.e., explicit warm up) via a sequential read of the user working set. While it takes only 130 seconds to bring 1GB of data into our fast tier by reading it sequentially (300 seconds and 700 seconds for 4GB and 8 GB of data, respectively), it takes 600 seconds to fully warm up the cache by the user workload alone (more than one hour for the 4GB and 8GB working sets). As the working sets and fast tier capacities grow to TBytes, it becomes imperative not only to warm up the cache before the high user load starts, but doing it proactively (with the aid of our model) than reactively (feedback). A more fine-grained evaluation of our predictive policy is done via trace-driven simulation in the next subsection.

4.2 Simulation Results

In order to evaluate our predictive approach at a fine-grain level and better understand its statistical properties, we experiment also with a trace-driven simulation that allows us to change the various parameters of the experiment. The simulation, in particularly, allows us to analyze the benefits of the predictive methods as the size of the fast tier increases, without being constricted by the specific hardware as in the case of our limited testbed.

Our simulation is driven by the traces described in Section 2. Since the traces contain only the arrival process, the service process is assumed to be exponentially distributed with a mean service rate that ensures that the response time remains flat during the full range of user arrival intensities. We simulate a two-tiered storage system with configurable capacities and user active data set sizes to experiment with different fast tier warm up times (i.e., 1 minute, 15 minutes, and 60 minutes).

We have implemented the feedback-based and the prediction-based policies for scheduling system work. As a baseline comparison, we also report performance data when no system work is launched (i.e., we also present the user-only case). The simulation, similar to our measurement experiments, is built such that when the system is experiencing high user arrival intensities, the system work is stopped. When the system experiences low arrival intensities then the system serves both user requests and system work. The difference between the predictive and feedback approaches lies in the exact time *when* the system work is stopped and resumed.

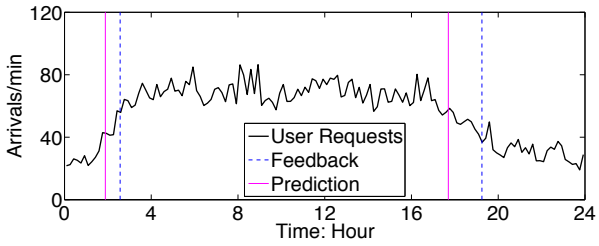
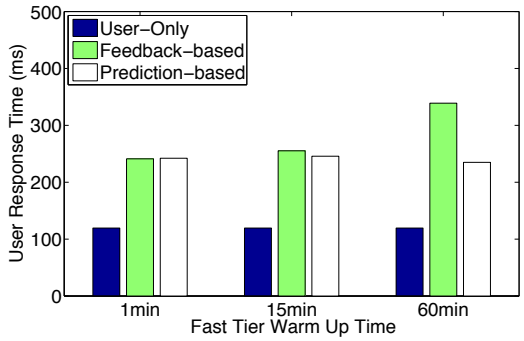


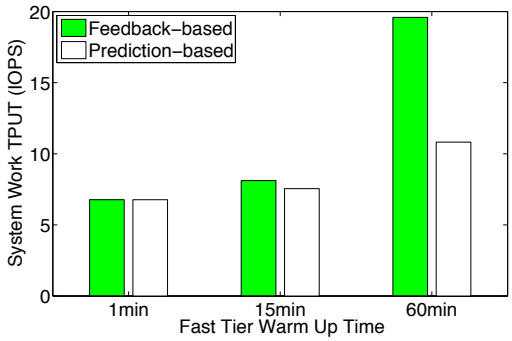
Figure 13: Predicted state change by feedback and prediction methods.

For the results that we present here, the model is already trained with two weeks of trace data and we present results when the model is applied in one of the days (day 20, see Figure 3). Figure 13 illustrates the points where the two models differ. The lines that are marked as feedback illustrate the points in time where the arrival intensity changes after observation. Note the feedback uses on-line detection, so there is a delay between the true change point and detected moment. The prediction lines correspond to the time stamps where the model predicts that there is an imminent load change. Due to the stochasticity of the arrival intensities and the Markovian-based model, the prediction model deviates from the real change that is accurately detected by the reactive, feedback-based method. Yet, this accuracy of the feedback model becomes almost a moot point since it cannot be used to enable tier warm up before the high user load.

In our simulation experiments, we compare the average user response time and the average system work throughput between different fast tier capacities (measured by the time it takes to warm up). The expectation is that the predictive method would detect the incoming *High* state and proactively warm-up the fast tier earlier than the feedback method detects the *High* state after the fact. As a result, system work runs for longer stretches under the feedback method than the predictive method. Consequently, the predictive method completes less system work, but also maintains high user performance. Note that the larger the fast tier, the higher the benefits of the predictive approach, otherwise the system is left to operate under high user arrival intensities and a cold fast tier for longer periods of time. These behaviors are captured in Figure 14 and further corroborated by Figure 15, where the CDF of the user response time is plotted under the scenario of a fast tier requiring 60 minutes to warm up. In the other two cases of smaller fast tiers, the differences between the feedback method and the predictive methods are not as pronounced. As a final note, note that in Figure 15 we have also added the ideal proactive policy that assumes full knowledge of the future workload to initiate the tier warm up. The response time CDF of the prediction-based policy is very close to that of the ideal one, which further argues about the effectiveness of the model prediction.



(a) average user response time



(b) throughput of system work

Figure 14: Performance comparisons via simulation. Note the throughput for system work is null in the user only case.

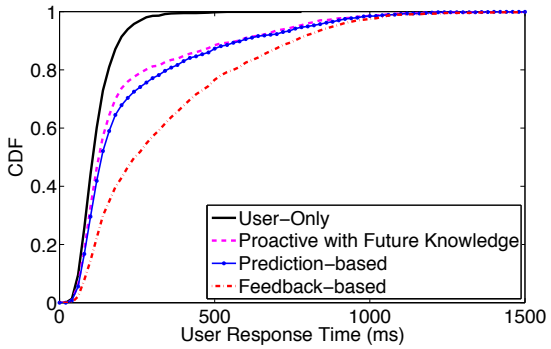


Figure 15: CDF of user response time.

5 Related Work

There is a rich body of work in the literature on storage tiering. Hierarchical storage systems are early examples of storage tiering techniques . HPSS [3] has higher tier (disk) and lower tier (tape), but only allows files to be read or written from the higher tier while the lower tier is treated as an offline device, e.g., data must first migrate to the higher tier before been accessed. VxFS [9] improves the flexibility of the early hierarchical storage systems by allowing user defined placement and migration rules. As the cost of SSDs reduced, SSDs have been introduced in the storage hierarchy. HP'S 3PAR [18] and EMC's FAST [11] are examples of such systems.

Storage tiering is usually critical for meeting service level agreements (SLA) because it can significantly boost the overall system performance. Amazon provides ElastiCache [1] for improving application performance by adding an in-memory caching layer to the infrastructure. FlashTier [20] proposes an interface that is designed for using an SSD as a fast storage tier. Everest [15] offloads bursty I/O workloads by using spare disk bandwidth to a virtual short-term persistent storage so that the I/O request latency during peaks is improved. There are other storage tiering works focused on improving reliability [5, 17, 7] and cost or energy savings [10, 16].

There are various system storage works that are usually scheduled as a "background" activity for various purposes, including replication [21, 23], security [12], and data analysis [22, 14]. Several workload interleaving techniques [8, 24] have been proposed for scheduling such system or background work, but they do not consider storage tiering.

The work most related to ours is optimizing storage cache warm up. Bonfire [25] accelerates the cache warmup by using more efficient pre-load methods. Windows SuperFetch [4] pre-loads the frequently used system and application information and libraries into memory based on the usage pattern in history to reduce the system boot and application launching time. While Bonfire [25] and SuperFetch [4] focus mostly on identifying the data that should be brought into the fast tier for higher efficiency, our work concentrates on identifying *when* to proactively bring a specific and pre-identified data set into the fast tier such that the system can be best utilized by system work with minimal impact on user perceived performance. In this regard, our proposed predictive framework can be viewed as complementary to Bonfire [25] and SuperFetch [4].

6 Conclusions

In this paper, we examine the effects of various workload interleaving techniques in tiered storage and demonstrate the performance benefits of a stochastic, Markovian-based model that can be used to first learn and then predict cyclic patterns in user workload intensity. Using a variety of user workload traces for production systems, we demonstrate the robustness of the model as it effectively suggests when to start and when to stop the deployment of system storage features in order to serve system work during the most opportune low utilization periods.

Acknowledgments

This work is supported by NSF grants CCF-0937925 and CCF-1218758.

References

[1] Amazon ElastiCache. `http://aws.amazon.com/elasticache`. Last visited on: May 19th, 2014.

[2] FIO Benchmark. `http://www.freecode.com/projects/fio`. Last visited on: May 19th, 2014.

[3] HPSS User's Guide. `http://www.hpss-collaboration.org/user_doc.shtml`. Last visited on: May 19th, 2014.

[4] Inside the Windows Vista Kernel. `http://technet.microsoft.com/en-us/magazine/2007.03.vistakernel.aspx`. Last visited on: May 19th, 2014.

[5] AGUILERA, M. K., KEETON, K., MERCHANT, A., MUNISWAMY-REDDY, K.-K., AND UYSAL, M. Improving recoverability in multi-tier storage systems. In *DSN* (2007), IEEE, pp. 677–686.

[6] BIRKE, R., PODZIMEK, A., CHEN, L. Y., AND SMIRNI, E. State-of-the-practice in data center virtualization: Toward a better understanding of vm usage. In *DSN* (2013), IEEE, pp. 1–12.

[7] CHEN, P. M., NG, W. T., CHANDRA, S., AYCOCK, C., RAJAMANI, G., AND LOWELL, D. The Rio file cache: Surviving operating system crashes. In *ASPLOS* (1996), pp. 74–83.

[8] EGGERT, L., AND TOUCH, J. Idletime scheduling with preemption intervals. In *SOSP* (2005), pp. 249–262.

[9] GANESH KARCHE, MURTHY MAMIDI, P. M. Using dynamic storage tiering. *Symantec Yellow Books* (2006).

[10] GUERRA, J., PUCHA, H., GLIDER, J. S., BELLUOMINI, W., AND RANGASWAMI, R. Cost effective storage using extent based dynamic tiering. In *FAST* (2011), pp. 273–286.

[11] LALIBERTE, B. Automate and optimize a tiered storage environment - FAST! *ESG White Paper* (2009).

[12] LI, Y., DHOTRE, N. S., OHARA, Y., KROEGER, T. M., MILLER, E. L., AND LONG, D. D. Horus: Fine-grained encryption-based security for large-scale storage. In *FAST* (2013).

[13] MARDIA, K. Multivariate analysis. *Academic Press* (1979).

[14] MIHAILESCU, M., SOUNDARARAJAN, G., AND AMZA, C. Mixapart: decoupled analytics for shared storage systems. *USENIX HotStorage* (2012).

[15] NARAYANAN, D., DONNELLY, A., THERESKA, E., ELNIKETY, S., AND ROWSTRON, A. I. Everest: Scaling down peak loads through I/O off-loading. In *OSDI* (2008), pp. 15–28.

[16] OH, Y., CHOI, J., LEE, D., AND NOH, S. H. Caching less for better performance: Balancing cache size and update cost of flash memory cache in hybrid storage systems. In *FAST* (2012).

[17] OUSTERHOUT, J., AGRAWAL, P., ERICKSON, D., KOZYRAKIS, C., LEVERICH, J., MAZIÈRES, D., MITRA, S., NARAYANAN, A., PARULKAR, G., ROSENBLUM, M., ET AL. The case for ramclouds: scalable high-performance storage entirely in dram. *ACM SIGOPS Operating Systems Review 43*, 4 (2010), 92–105.

[18] PETERS, M. 3PAR: Optimizing I/O service levels. *ESG White Paper* (2010).

[19] ROUSSEEUW, P. J. Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. *Journal of Computational and Applied Mathematics 20* (1987), 53–65.

[20] SAXENA, M., SWIFT, M. M., AND ZHANG, Y. Flashtier: A lightweight, consistent and durable storage cache. In *EUROSYS* (2012), pp. 267–280.

[21] SHILANE, P., HUANG, M., WALLACE, G., AND HSU, W. Wan-optimized replication of backup datasets using stream-informed delta compression. *ACM Transactions on Storage (TOS) 8*, 4 (2012).

[22] TIWARI, D., BOBOILA, S., VAZHKUDAI, S. S., KIM, Y., MA, X., DESNOYERS, P. J., AND SOLIHIN, Y. Active FLASH: Towards energy-efficient, in-situ data analytics on extreme-scale machines. In *FAST* (2013).

[23] WALLACE, G., DOUGLIS, F., QIAN, H., SHILANE, P., SMALDONE, S., CHAMNESS, M., AND HSU, W. Characteristics of backup workloads in production systems. In *FAST* (2012).

[24] YAN, F., RISKA, A., AND SMIRNI, E. Busy bee: how to use traffic information for better scheduling of background tasks. In *ICPE* (2012), pp. 145–156.

[25] ZHANG, Y., SOUNDARARAJAN, G., STORER, M. W., BAIRAVASUNDARAM, L. N., SUBBIAH, S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Warming up storage-level caches with Bonfire. In *FAST* (2013).

Model-driven Elasticity and DoS Attack Mitigation in Cloud Environments

Cornel Barna
York University
Toronto, Ontario, Canada
cornel@cse.yorku.ca

Mark Shtern
York University
Toronto, Ontario, Canada
mark@cse.yorku.ca

Michael Smit
Dalhousie University
Halifax, Nova Scotia, Canada
msmit@dal.ca

Hamoun Ghanbari
York University
Toronto, Ontario, Canada
hamoun@cse.yorku.ca

Marin Litoiu
York University
Toronto, Ontario, Canada
mlitoiu@yorku.ca

Abstract

Workloads for web applications can change rapidly. When the change is an increase in customers, a common adaptive approach to maintain SLAs is elasticity, the on-demand allocation of computing resources. However, application-level denial-of-service (DoS) attacks can also cause changes in workload, and require an entirely different response. These two issues are often addressed separately (in both research and application). This paper presents a model-driven adaptive management mechanism which can correctly scale a web application, mitigate a DoS attack, or both, based on an assessment of the business value of workload. This approach is enabled by modifying a layered queuing network model previously used to model data centers to also accurately predict short-term cloud behavior, despite cloud variability over time. We evaluate our approach on Amazon EC2 and demonstrate the ability to horizontally scale a sample web application in response to an increase in legitimate traffic while mitigating multiple DoS attacks, achieving the established performance goal.

1 Introduction

The use of software-defined infrastructure enables the addition or removal of resources (computation, storage, networking, etc.) to or from a deployed web application at run-time in response to changes in workload or in the environment. Central to this *elasticity* is the use of mechanisms that autonomically decide when to make these changes. Many approaches have been proposed and tested (see for example a recent survey [7]), including reactive approaches that establish thresholds or elasticity policies which determine when changes will be made (e.g., [10, 18, 17, 5]) and proactive approaches that attempt to anticipate future requirements using techniques like queuing models [30, 9], simulation-generated state machines [28], or reinforcement learning [4]. The focus is typically on meeting a desired service level by ensuring provisioned resources are sufficient to handle a workload, perhaps while minimizing the total infrastructure cost [9].

The typical assumption of elasticity mechanisms is that all traffic arriving at the application is desirable. This is not always the case. For example, an application-level denial of service (DoS) attack has many of the same characteristics of an increase in legitimate visitors, especially a low-and-slow DoS attack [25], but represents undesirable load on the application. Such attacks are increasing in volume and sophistication [6], due in part to freely available tools [14, 22]. A common response to a denial-of-service attack at the application layer is to add resources to ensure the service remains available (e.g. [14, 8, 16]), which resembles elasticity. However, deploying sufficient resources to handle a major DoS attack is expensive [23], with little return on investment. Another example is a cost of service attack, where the goal is not to deny service but to increase the cost of offering a service [12, 26]; or heavy traffic from an online community (the so-called Slashdot Effect) that does not generate revenue.

In this paper, we propose, implement, and evaluate a unified approach to enabling elasticity and mitigating DoS attacks. Rather than view DoS attempts as malicious traffic (in contrast to legitimate traffic), or even an evolved definition of "any workload beyond our capacity" [3], we define DoS traffic to be any segment of workload we cannot handle *while still providing value to the organization*. This perspective offers the opportunity to view self-management as a business decision based on a cost-benefit analysis of adding resources: if there is benefit (e.g. increased sales, ad impressions, profit, brand reputation, etc.) that exceeds the expected cost, then add resources; otherwise, manage the traffic. Workload is regarded not as malicious or legitimate, but rather as either (potentially) undesirable or desirable.

We describe three primary contributions of this work: an adaptive management algorithm for choosing which

portions of a workload need additional resources and which portions represent undesirable traffic and should be mitigated; adapting a layered queuing network (LQN) model to cloud environments to enable proactive cost-benefit analysis of workload; and an implementation and a series of experiments to evaluate this approach in the Amazon EC2 IaaS cloud environment.

Our algorithm (§2) examines portions of the workload and assesses whether incoming traffic is desirable or undesirable. This decision is based on a runtime software quality metric called the cloud efficiency metric [26], which at its most basic calculates the total cost of the software-defined infrastructure and calculates the ratio to the revenue generated by incoming traffic (though in the general case, value can be defined very broadly). Traffic considered undesirable is handled as described in previous work [3, 2], where instead of being discarded it is forwarded to a *checkpoint*. At this checkpoint, a challenge is issued and the user is asked to verify that they are a legitimate (and valuable!) visitor (for example, using a CAPTCHA test as in [20]). The management is completely autonomic; this avoids the known problems with the complexity of manually tuning threshold-based elasticity rules [10].

Estimating the cost-benefit potential requires a proactive approach that takes measurements from the deployed infrastructure and makes short-term predictions. Our overall approach does not prescribe which mechanism should be used; we have chosen to illustrate our approach using a LQN model called OPERA. Section 3 describes the challenges in using OPERA to predict cloud behavior in practice. We present experimental results demonstrating how OPERA diverges from reality over time due to the unpredictable variability of cloud services [24], describe the modifications required to account for unexplained delays, and a second set of results that demonstrate with the modifications the LQN remained synchronized with the actual performance of the real cloud system.

We implemented our algorithm (§4), deployed a sample e-commerce application protected with our updated protection/elasticity autonomic manager, and tested response to a several attack scenarios: DoS alone, increase in customer traffic alone, and both combined (the common case where a site becomes more popular but also attracts negative attention). Our results (§5) show that the application is protected from all forms of surging traffic by adding servers or mitigating undesirable traffic, as appropriate. We also identify a limitation of our approach: the reaction time is slow enough that a temporary backlog of requests can be created, which skews calculations and leads to the temporary mitigation of desirable traffic until the model recovers. We present an example of this limitation.

2 Methodology

The goal of our approach is to treat desirable traffic (which generates business value) differently than undesirable traffic (which consumes resources disproportionate to the value created). This novel broad view of elasticity better reflects business objectives, while also addressing issues that have historically been dealt with separately. To achieve this goal, the autonomic manager must be capable of differentiating between the two, and adding or removing resources to ensure desirable traffic encounters sufficient quality of service without overspending, while routing undesirable traffic through an additional checkpoint. In this section, we introduce an algorithm for an adaptive manager with these capabilities.

The overall model of the approach resembles a standard feedback-loop, with an adaptive manager accepting monitoring data, using a predictive model to inform decision-making, and executing decisions autonomically using a deployment component capable of adding and removing resources. The managed system is a standard three-tier web application.

The behavior of the adaptive manager is described in Algorithm 1. At each iteration, a new set of metrics is observed from the managed system and its environment, including current workload, current performance, and current deployment information (which includes any ongoing traffic redirection or scaling activities). Some of this information is provided for each distinct class of traffic. For example, traffic accessing features related to browsing an e-commerce catalog might be grouped into a single class of traffic; similarly, features related to the checkout process might get their own class. These classes (sometimes called usage scenarios) allow traffic to be treated more granularly than simply looking at overall traffic to the application. A class of traffic corresponds to classes of services used in Layered Queuing models (§3.1).

Included in the set of performance metrics is the current cost efficiency (CE) [26], a runtime software quality metric that captures the ratio of the benefit derived from the application to the cost of offering the application: $CE = \frac{\text{application benefit function}}{\text{infrastructure cost function}}$.

Due to size constraints, the details of the cost efficiency metrics are not included in this paper, but the reader can find an in-depth presentation in [26]. To summarize, the **cost function** must capture the real cost of offering a given application on the cloud for the given period (e.g., per hour). This function excludes other costs related to the application, e.g. the cost of development, the cost of goods sold, and customer support. It is not a measure of overall profitability; rather, it captures current infrastructure costs. The **benefit function** must capture the benefit the application provides to the

Algorithm 1: Decision Algorithm – The algorithm used by the adaptive manager to choose appropriate actions for the managed system.

input : $\mathbb{C}^u$ – the set of unaltered traffic classes;
input : $\mathbb{C}^f$ – the set of all classes of traffic redirected to a checkpoint;
input : L^u – the vector of current load on unaltered classes of traffic;
input : L^f – the vector of current load on each class of traffic redirected to a checkpoint;
input : M_m – the vector of measured performance metrics;
input : svr_{cur} – the number of current web servers;
input : svr_{max} – the maximum number of web servers that can be allowed to run;
input : err – the accepted error for the model estimations;
output: $\mathbb{A}$ – the deployment plan.

1 Use LQM to compute the estimated performance metrics, M_e, for the load L^u;
2 **while** $\left|1 - \frac{M_e}{M_m}\right| > err$ **do**
3 $D \leftarrow$ *Kalman*(M_m, L^u, *LQM*); $M_e \leftarrow$ *LQM*(D, L^u);
4 $svr_{ce} \leftarrow$ the maximum number of servers that can be added and still be cost effective;
5 $A \leftarrow$ {**do nothing**};
6 **if** M_m violates SLOs **then**
7 $svr \leftarrow min(svr_{max}, svr_{cur} + svr_{ce})$;
8 $n \leftarrow$ *CalculateServersToAdd*(L^u, M_m, svr_{cur}, svr);
9 **if** $n > 0$ **then**
10 $\mathbb{A} \leftarrow$ {**add** n web servers};
11 **else**
12 $\mathbb{C} \leftarrow$ *TrafficClassesToRedirect*(L^u, M_m, err, $\mathbb{C}^u$);
13 $\mathbb{A} \leftarrow$ {**redirect** traffic classes $\mathbb{C}$};
14 **else**
15 set in the model the number of web servers to $svr_{cur} + svr_{ce}$;
16 $\mathbb{C} \leftarrow$ *TrafficClassesToRestore*(L^u, L^f, M_m, err, $\mathbb{C}^f$);
17 **if** $\mathbb{C} \neq \emptyset$ **then**
18 $svr \leftarrow svr_{cur} - 1$;
19 $\mathbb{C}_{tmp} \leftarrow \emptyset$;
20 **while** $\mathbb{C}_{tmp} \neq \mathbb{C}$ **do**
21 $svr \leftarrow svr + 1$;
22 set in the model the number of web servers to svr;
23 $\mathbb{C}_{tmp} \leftarrow$ *TrafficClassesToRestore*(L^u, L^f, M_m, err, $\mathbb{C}^f$);
24 **if** $svr - svr_{cur} > 0$ **then**
25 $\mathbb{A} \leftarrow$ {**add** $svr - svr_{cur}$ web servers} $\cup$ {**stop redirecting** traffic classes $\mathbb{C}$};
26 **else**
27 $\mathbb{A} \leftarrow$ {**stop redirecting** traffic classes $\mathbb{C}$};
28 **else**
29 $n \leftarrow$ *CalculateServersToRemove*(L^u, M_m, svr_{cur});
30 **if** $n > 0$ **then**
31 $\mathbb{A} \leftarrow$ {**remove** n web servers};
32 **return** A

organization. Typically, organizations have mechanisms for assessing this benefit at least at the macro-level. The benefit may come from many sources: revenue, advertising, brand awareness, customer satisfaction, number of repeat customers, or any number of business-specific metrics. For example, a denial of service attack on an e-commerce website would reduce the value of incoming traffic (as fewer visitors would be able to make purchases), which would reduce the overall cost efficiency. If an adaptive manager were in use and were to add additional resources to handle the DoS traffic, the current value would be maintained, but the cost would increase: the overall effect would be the same.

OPERA is used to estimate a set of performance metrics (CPU utilization, response time, throughput), which are compared to the measured set of performance metrics to ensure the model is still synchronized with the system (line 2; if not, it is re-tuned using Kalman filters [32]).

On line 6, we test compliance with service-level objectives (SLOs); if any measured performance metrics are non-compliant (perhaps due to a decrease in cost efficiency, or an increase in response time), a remedial action must be taken (logic regarding cool-down times to avoid thrashing is omitted for clarity). The remedial action is chosen from two options (adding servers because the traffic has value or redirecting traffic to a checkpoint because it does not). To decide on which action is appropriate, we call a function which uses the predictive model to estimate the impact of adding one or more web servers, up to a maximum cap on the number of servers. If adding web servers is estimated to bring performance metrics in compliance with the SLO, this solution is executed and the chosen number of servers is added (line 8). Performance metrics include the cost efficiency metric, which means that if additional traffic does not add business value, adding servers will increase the cost and therefore lower cost efficiency. In this case the approach will be rejected and 0 will be returned. The algorithm will then consider redirecting some traffic to a checkpoint instead. The algorithm considers each class of traffic separately. For example, it might be appropriate to redirect the traffic of users who access only free services, while preserving the traffic of those who pay a monthly subscription fee. Or we might give priority to the class of traffic that includes the actual checkout process, versus the customer discussion forums. The function TrafficClassesToRedirect determines which class of traffic should be redirected. The complete definition is provided in [3], but conceptually the LQN model is used to produce a set of performance metric estimates based on blocking various classes of traffic. The set returned consists of the classes that produce the best performance metrics (including cost efficiency). Traffic to these classes are redirected to a checkpoint for verification of legitimacy; this checkpoint also serves as a *speedbump* for legitimate traffic.

In contrast, if measured performance complies with the SLOs, the algorithm checks to see if we can restore classes of traffic (or if we could restore classes of traf-

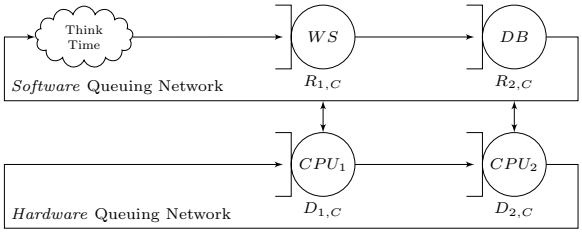


Figure 1: Software and hardware layers in a LQN of a 2-tier web system.

fic if we added additional servers, line 16) or if we can reduce the number of servers. The model is used to predict performance measures (including cost efficiency) under each possible action. If the model estimates that performance metrics would remain in compliance with the SLO after restoring traffic classes, these classes are restored; similarly for removing one or more servers (line 29). It is important to note that servers can be removed even while traffic is being redirected; the elasticity function is focused on the desirable traffic, not on overall traffic. The number of servers to remove is calculated as the largest possible reduction before the model estimates SLOs would be violated.

The algorithm will terminate returning a set of actions (which may be a null set), specifying which traffic classes to redirect (or restore) and specifying the number of servers to add (or remove).

This general approach allows an administrator to specify their expected SLOs, to define a benefit function, and to define classes of traffic; however, they are not expected to write procedural rules or detailed policies. The adaptive manager is responsible for deciding both what action to take (managing traffic or adding/removing resources) and the magnitude of that action (how much traffic to manage, how many resources to add/remove). The inclusion of a general cost efficiency metric allows the adaptive manager to make decisions that reflect business objectives, rather than going to great expense to handle large amounts of traffic.

3 A Layered Queuing Network for Cloud Environments

A layered queuing network model (LQN) is at the heart of our methodology. While our algorithm is general, we implement it using a particular layered queuing network (named OPERA) which we have used successfully to model transactional web applications deployed on hardware infrastructure. In the process of validating its accuracy in cloud environments, we found that over time it diverged from reality, and that for some values (such as modeled response time) it was consistently below the actual values. This section describes using a LQN to model an application on the cloud.

3.1 Previous model

For a transactional web application such as those examined in this paper, the user interaction with the system is modeled using *classes of services* (or simply *classes*), a service or a group of services that have similar statistical behavior and have similar requirements. When a user begins interacting with a service, a *user session* is created, and persists until the user logs out or becomes inactive. We define N as the number of active users at some moment t; these users can be distributed among different classes of services. For a system with C classes, we define N_c as the number of users in class C, thus $N = N_1 + N_2 + \cdots + N_C$. N is also called *workload intensity* or *population* while combinations of N_c are called *workload mixes* or *population mixes*.

Any software-hardware system can be described by two layers of queuing networks [21, 19]. The first layer models the software resource contention, and the second layer models the hardware contention. To illustrate the idea, consider a web based system with two software tiers, a web application server (WS) and database (DB) server (see Figure 1). Each server runs on dedicated hardware, which for the purpose of this illustration we will limit to CPUs only (CPU_1 and CPU_2 respectively). The hardware layer can be seen as a queuing network with two queues, one for each hardware resource. The software layer also has two queues, one for the WS process and another for the DB process, which queue requests waiting for an available thread (or for *critical sections*, *semaphores*, etc.). The software layer also has a *Think Time* centre that models the delay between requests that replicate how to model how long a user waits before sending their next request.

Each resource has a *demand* (or *service time*, i.e. the time necessary for a single user to get service from that resource) for each *class*. If in this example there are two classes of service, there will be four demands for the hardware layer: each class will have a demand for each CPU. The service times (demands) at the software layer are the *response times* of the hardware layer. In our case, for class C, they are $R^s_{1,C}$ and $R^s_{2,C}$, and they include the demand and the waiting time at the hardware layer (we use the upper script s to denote software metrics that belong to the software layer). Ideally, hardware demand is based on measured values; however, this is impractical for CPUs because of the overhead imposed by collecting such measurements. In our approach the CPU demands are estimated using Kalman filters. Once the model is created, it is iteratively tuned, also using Kalman filters.

This model has been used to inform a variety of

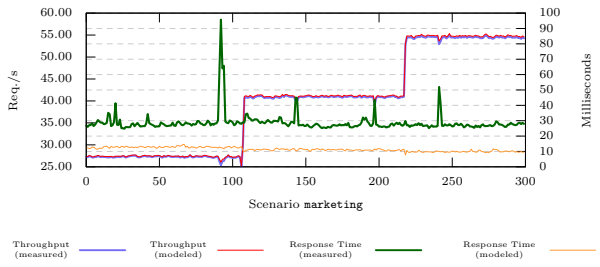


Figure 2: Observed versus estimated values for 2 key performance metrics, showing the `marketing` scenario only.

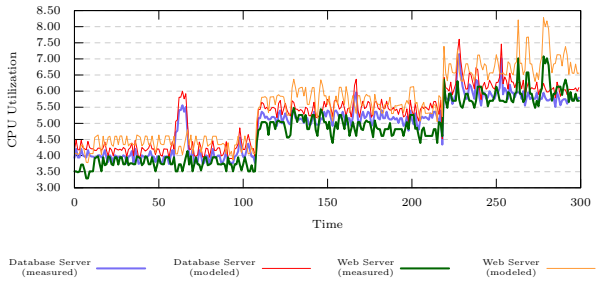


Figure 3: The CPU utilization for the `marketing` scenario; the model is not sychronized with observed measurements.

adaptive systems, including implementing elasticity policies [9] and mitigating DoS attacks [3, 2]. In earlier work, Zheng et al. [32] present a method for tuning model parameters for a web application. Properly tuned models have been shown to accurately estimate performance metrics [32].

3.2 OPERA in the Cloud

Despite the proven track record of this model, when we switched from modelling a private data center to modelling the public cloud (Amazon AWS EC2), we noticed significant differences between the model's estimates and the actual observed performance metrics.

To illustrate the issue, we deployed a sample web application (see §5) and generated an increasing workload to a single class of service, `marketing` scenario. Figure 2 shows the measured and modeled response time (right Y-axis) and throughput (left Y-axis) when the workload is increased. The picture shows that the model is well-synchronized with the system for throughput, but not for the response time; the measured response time is almost three times the estimated one. As the workload increases, the synchronization procedure manages to tune the model for the throughput, but does not improve the accuracy of predicted response time.

Figure 3 shows the CPU utilization for the database server and the application server, both observed and predicted. The distance between the observed and predicted values is noticeable, and diverges further as workload increases, demonstrating that this metric is also not synchronized between the model and reality.

LQN models are well-established approaches to predicting the performance of web applications; after ruling out measurement errors, we concluded our particular model is missing an unknown factor related to the known variance in EC2 [24, 27]. In order to improve the estimated metrics and reduce the modelling error overall, we added a "cloud delay centre" designed to capture all the undocumented work or delay. The cloud delay queue is shared by all classes of traffic, it has no limit, it exists at the software level, and it is the first queue encountered by incoming requests.

Following this change, we again tested synchronization and obtained substantially better results §5. The modified LQN is used throughout the remainder of this paper to make decisions about workload, to inform adaptive systems, and to implement elasticity policies; it demonstrated close alignment with measured values throughout this process. Other approaches that use LQN models to predict cloud behavior but do not validate their models against an actual cloud (e.g. [15],[1]) may wish to adopt a similar solution.

4 Implementation

We have implemented our adaptive management algorithm to manage applications deployed to the Amazon EC2 public cloud infrastructure, using a framework that can automatically deploy a topology in EC2 by launching all instances required (application servers, database, load balancer), install the required software on each machine (e.g. Tomcat and all dependencies, the web application and its libraries, etc.), and configure each application (for example, add the application servers to the load balancer). The framework provides an API that accepts requests to modify the deployed topology – for example, if the request is to horizontally scale the web tier, the framework can launch an instance for the application server, install all required software, and add the new server to the load balancer. The framework is general enough to allow us to install and configure any type of application in a linux environment, once the deployer provides installation and management scripts.

Our implementation uses Amazon CloudWatch detailed monitoring to acquire performance metrics from the deployed instances. As discussed in [27], these metrics are delayed by one minute from when they are recorded. Our implementation runs the main algorithm, and acquires Cloudwatch metrics, once per minute. The arrival rate, throughput, and response time for each class of traffic are acquired from a reverse proxy on the load balancer at the same time[1]. This reverse proxy moni-

tors incoming requests and assigns them to the appropriate traffic class (based on the URL); the classification rules can be modified at runtime if necessary. The administrator of an application is responsible for defining their traffic classes based on their business logic. Measuring response time at the load balancer allows us to focus on the component of user-experienced response time that we can control. While end-to-end response time will be higher and more variable, adapting to slow user connections is outside the scope of this paper.

5 Experiments

To evaluate the contributions of this paper, we performed a set of experiments. The first examines our modified LQN to assess its ability to synchronize with the managed application so its estimates have predictive value. Experiments 2-4 examine a sample web application.

In all experiments we have used a *bookstore* application that we have developed using Java EE, which emulates an e-commerce web application. We defined six classes of traffic: `marketing` (browsing reviews and articles); `product selection` (searching the store catalog, product comparisons); `buy` (add to cart); `pay` (complete checkout); `inventory` (inventory tracking); and `auto bundle` (an upselling / discounting system). The workload used is generated randomly by a workload generator using an unevenly weighted mix of the 6 classes. Each class of traffic has a different performance impact on the deployed application.

Each experiment starts with a deployed topology, with a single application server in the web tier. The web tier is scaled horizontally by adding and removing resources. The traffic filtering approach described in [3] is used to refer undesirable traffic to a captcha to serve as the checkpoint.

5.1 Experiment 1: Synchronizing the model with public cloud resources

To verify the ability of our LQN to synchronize with reality in a cloud environment, we generated a constant workload and compared the estimated performance metrics to actual measurements at each sampling interval (Figure 4). We generated workload for all scenarios, though not all are shown due to space constraints. At each iteration or sampling interval, we measure the arrival rate for each scenario. We feed this workload into the model, and then solve the model to calculate the estimated metrics. If the error between measured and estimated values exceeds a specified threshold (10% in our case), we run the Kalman filters on the model in order to find more accurate values for the model parameters (this process is called *tuning the model*).

At $t = 0$, the estimated CPU utilization numbers (Fig. 4a) for the database server are almost double the measured values. Within 25-50 iterations, the Kalman filter settles on accurate model parameters, and the difference between measured and estimated values was around 1%. Importantly, once synchronization is achieved, it is not lost. Before we added the Cloud Delay Center, this synchronization was never achieved.

In plots 4b-4c, we show the measured response time (green line, using right Y-axis) and measured throughput (blue line, using left Y-axis) for two classes of traffic. After some initial error, the estimated values and measured values also remain within several percentage points, even through peaks and valleys.

We conclude from this data that the modified LQN has addressed the challenges of modelling a cloud environment by treating the variability of the cloud as a delay center in an LQN, and modifying the demand on that delay center using Kalman filters to account for unpredictable variability.

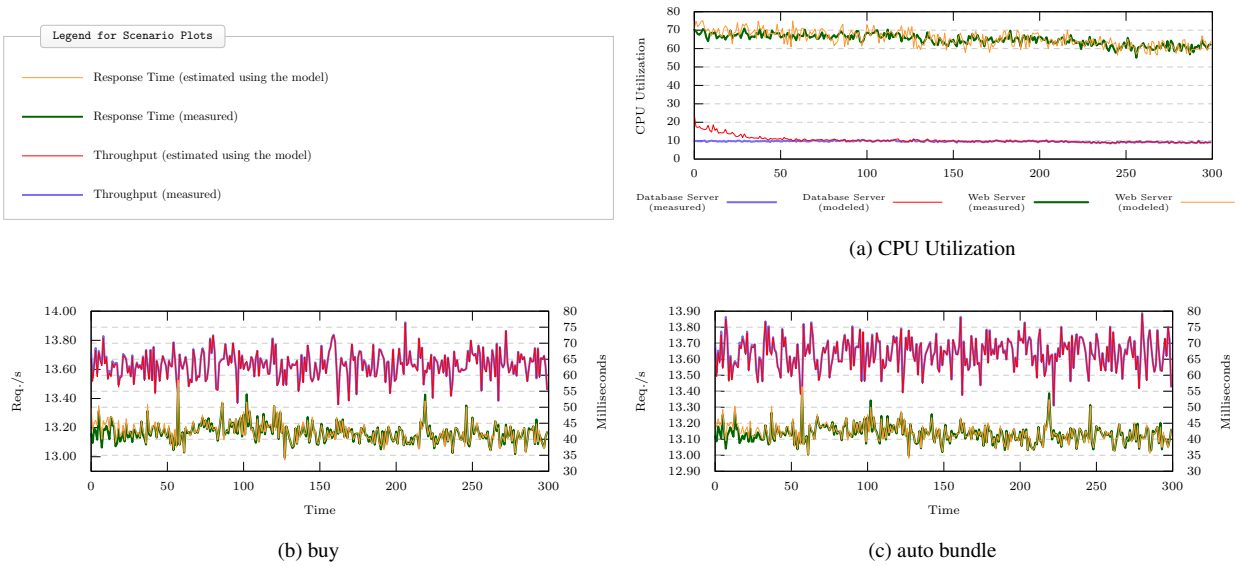
5.2 Experiment 2: Elasticity in the public cloud

This experiment examines the adaptive management algorithm's ability to provide elasticity when overall traffic increases and decreases. We use a simplified cost metric to calculate the cost of our EC2 deployment, and use the volume of the `pay` class of traffic as our benefit function. This is roughly analogous to prioritizing checkout activity over other activities. Application-level DoS attacks are not expected to generate traffic to this traffic class, because reaching the checkout page usually requires user interaction, a valid account, and valid credit card numbers[2]. The workload mix remained constant over the experiment, and therefore so did the cost efficiency metric.

Figure 5 shows the measurements obtained during this experiment. The workload generated for each scenario is captured (approximately) as the arrival rate (blue line in figures 5b–5g, on the left Y-axis). We started with a baseline workload; at iteration 50, we increased the workload. The adaptive manager added a new web server (purple line in figure 5a, on the right Y-axis). When the workload is increased, there is a brief spike in the CPU utilization metric for the web servers (red line in figure 5a, on the

[1] Initially, our framework used SNMP and JMX on the deployed VMs and application servers; however, these monitoring tools did not capture performance metrics for individual classes of traffic. Additionally, some values are measured incorrectly (such as CPU utilization) when running on a virtual machine in a cloud environment.

[2] Of course, our focus is the general approach and not optimal selection of the benefit function.



(a) CPU Utilization

(b) buy

(c) auto bundle

Figure 4: **Experiment #1.** Comparing estimated values to measured values for a selection of traffic classes for consistent workload.

left Y-axis), but also on the response time for each scenario (green line in figures 5b–5g, on right Y-axis). This violated the SLO, and caused the addition of a server because the cost efficiency metric remained within an acceptable range.

After workload increases at iteration 120 and again at 190, again a new server is added each time. Notice that after adding new servers, the CPU utilization for the web servers and the response time for the various classes of traffic have about the same values as before. This suggests that the number of servers added each time (estimated using the model) was appropriate to maintain SLOs.

At iteration 220, we dropped the workload sharply. The algorithm decided that two web servers could be safely removed. Again, the performance metrics remained acceptable following this removal.

The spikes in some of the measurements are largely due to delays in actuated new servers, leading to some backlog of requests and temporarily higher response times (off the graph, in some cases).

5.3 Experiment 3: Elasticity while mitigating DoS attacks

This experiment tests one of our key contributions: achieving elasticity to maintain SLOs for our desirable traffic, while detecting and redirecting undesirable traffic. To emulate a DoS attack, we dramatically increased the volume of traffic generated to one (or more) of the traffic classes. Our measurements from this experiment are shown in Figure 6.

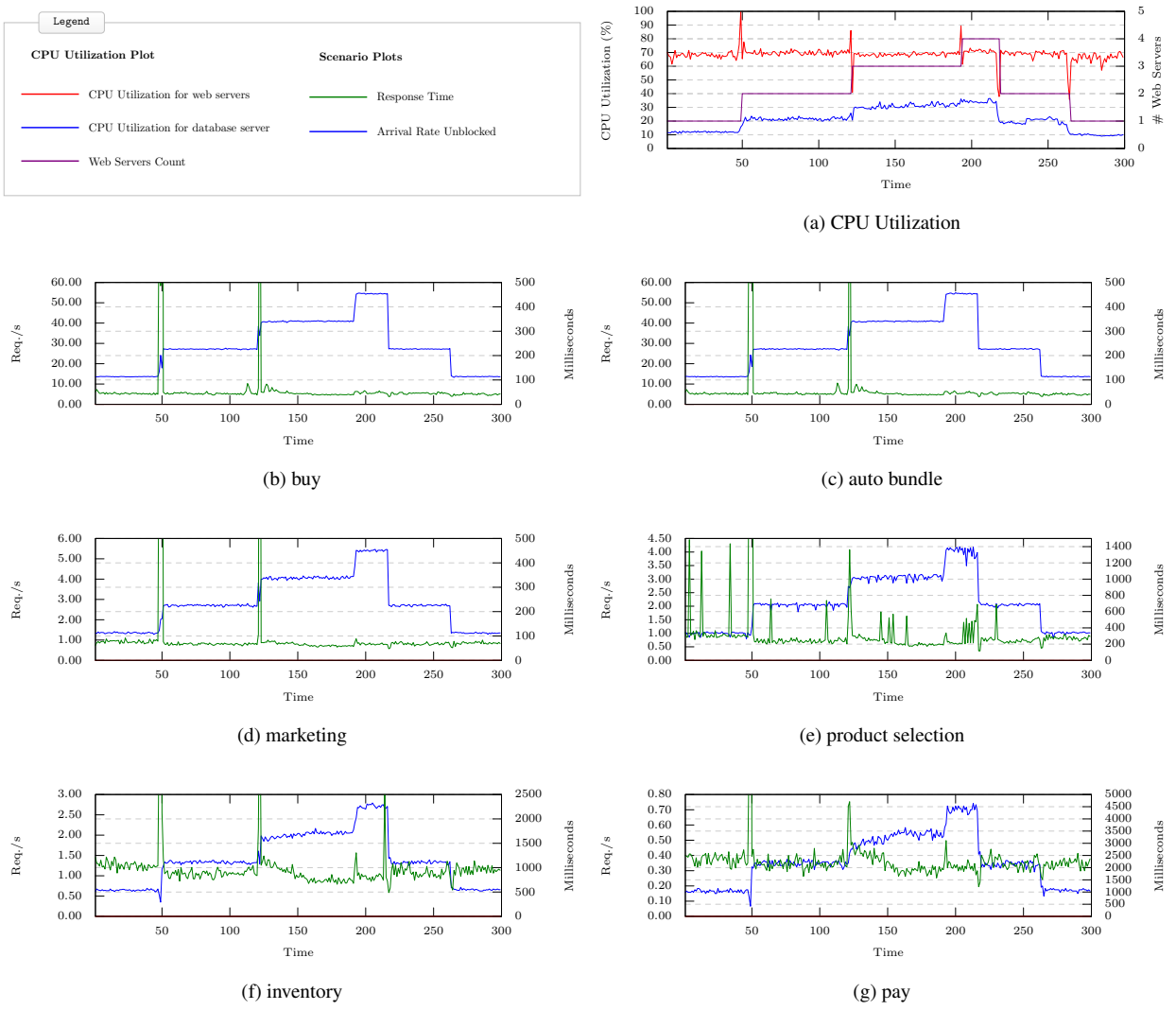
At iteration 4, we increase the workload across all traffic classes, and a new virtual machine is added. Performance degrades across all performance metrics in all classes, including the average CPU utilization. However, the addition of the new server restores performance metrics to acceptable levels.

At iteration 15 a first attack starts on a URL in the traffic class `marketing`. The arrival rate abruptly jumps from around 1.5 requests per second to close to 100 requests per second. The degradation in performance is immediate. Our algorithm provides the new metrics and workload levels to the LQN. The algorithm contemplates adding additional servers, but the increased cost is not offset by an increase in benefit, as the marketing traffic class does not generate revenue directly, and does not contribute to the benefit function. It instead determines it is necessary to redirect some traffic classes to a checkpoint and, after solving the model for various possible redirection schemes, determines (correctly) that the best course of action is to filter the requests on `marketing`.

A few iterations later (iteration 27), we emulated a second simultaneous attack on the `product selection` traffic class. Using a similar process, the algorithm once again identified the scenario under attack and redirected traffic to a checkpoint.

While these two attacks continued, the workload to the other classes increased for unrelated reasons. We can see response time (particularly for `inventory` and `pay`) increase. Once an SLO is violated, the algorithm decides two servers will be necessary to handle the continuing increase in desirable traffic (see purple line in figure 6a). This decision appears to be the correct one, as all performance metrics return to satisfactory levels (without being over provisioned).

At iteration 97, the first DoS attack (on `marketing`) is stopped, and the algorithm stops redirecting traffic for



(a) CPU Utilization

(b) buy

(c) auto bundle

(d) marketing

(e) product selection

(f) inventory

(g) pay

Figure 5: **Experiment #2.** Increasing and decreasing the workload resulted in the addition/removal of servers, while maintaining key performance metrics at acceptable levels.

that class of traffic. By iteration 120, the temporary increase in desirable traffic also resides, and at iteration 130 a web server is removed leaving the web cluster with three machines. Note there is still an ongoing DoS attack, and removing resources is counter-intuitive, but again the performance metrics suggest this was the correct decision as they are maintained at acceptable levels. When the second attack finishes, and the last redirection is halted, the system performance metrics stay within the SLOs. We do see an increase in some response times and had the experiment continued we expect the algorithm may have added an additional server.

This experiment shows that our algorithm is able to assure the elasticity of the web application and make good decisions to achieve SLOs and maintain cost efficiency even while the application is under one or more attacks.

5.4 Experiment 4: A limitation of the implementation

The previous experiments demonstrated the strength of our approach and our algorithm. However, there is a limitation: the reaction time is slower than is sometimes required to respond appropriately. This problem can be addressed by adding a statistical model that responds rapidly using a heuristic (as in [3]). The measurements from this experiment are shown in Figure 7.

When the first DoS attack starts on `marketing`, the system identified the problem and started redirecting traffic. At iteration 67, a second attack causes a severe and rapid degradation of performance metrics across all traffic classes and servers. Due to delays between iterations, the system is unprotected and the internal queues are filled. Our algorithm analyses the data and decides that two more classes need to be protected: `buy` and `auto`

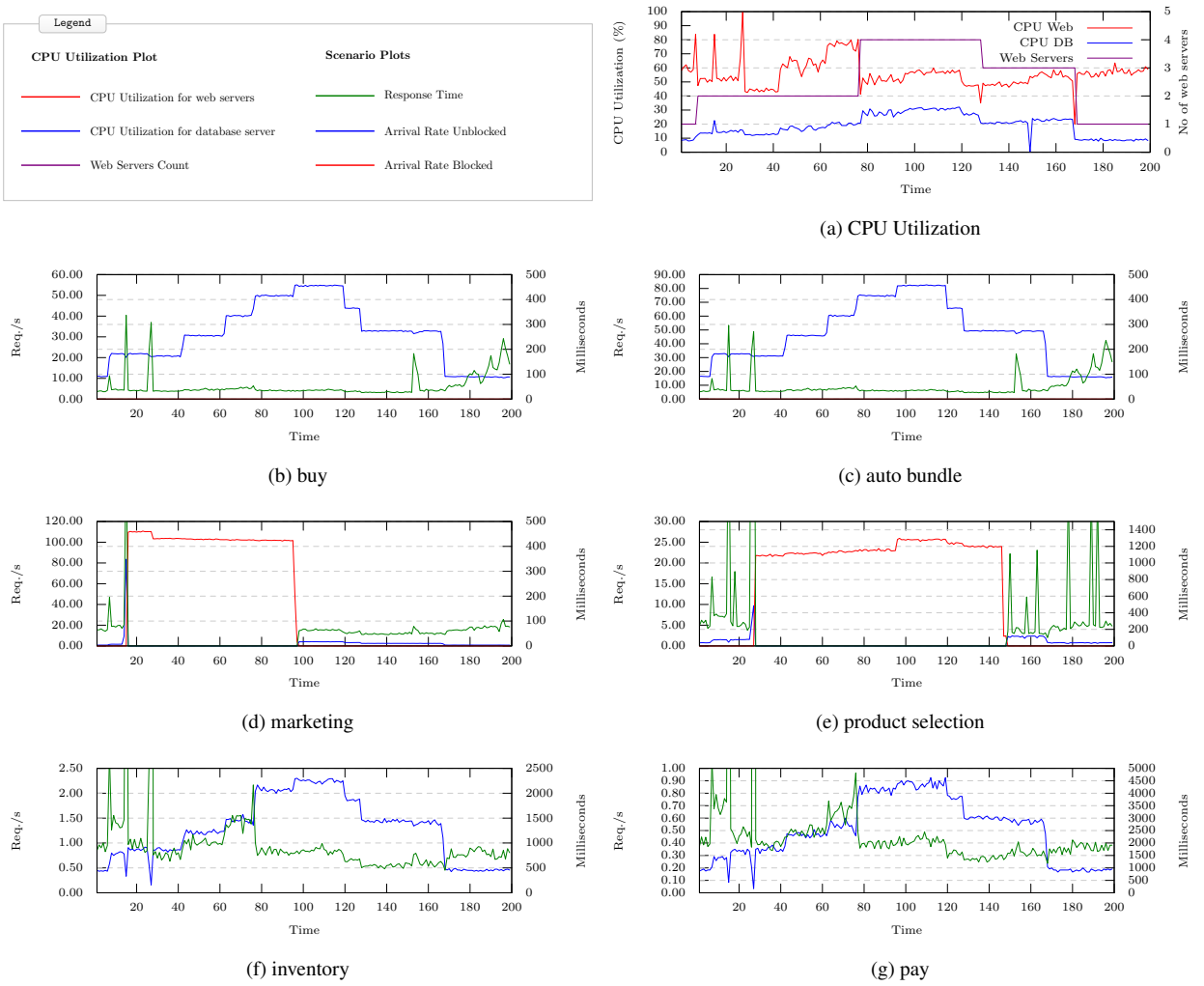


Figure 6: **Experiment #3.** Overlapping DoS attacks on two traffic classes; the algorithm mitigated these attacks while adjusting the number of VMs for the remaining workload.

`bundle`. The decision is correct in that it restores SLOs, but of course incorrect since no malicious traffic is targeting these classes.

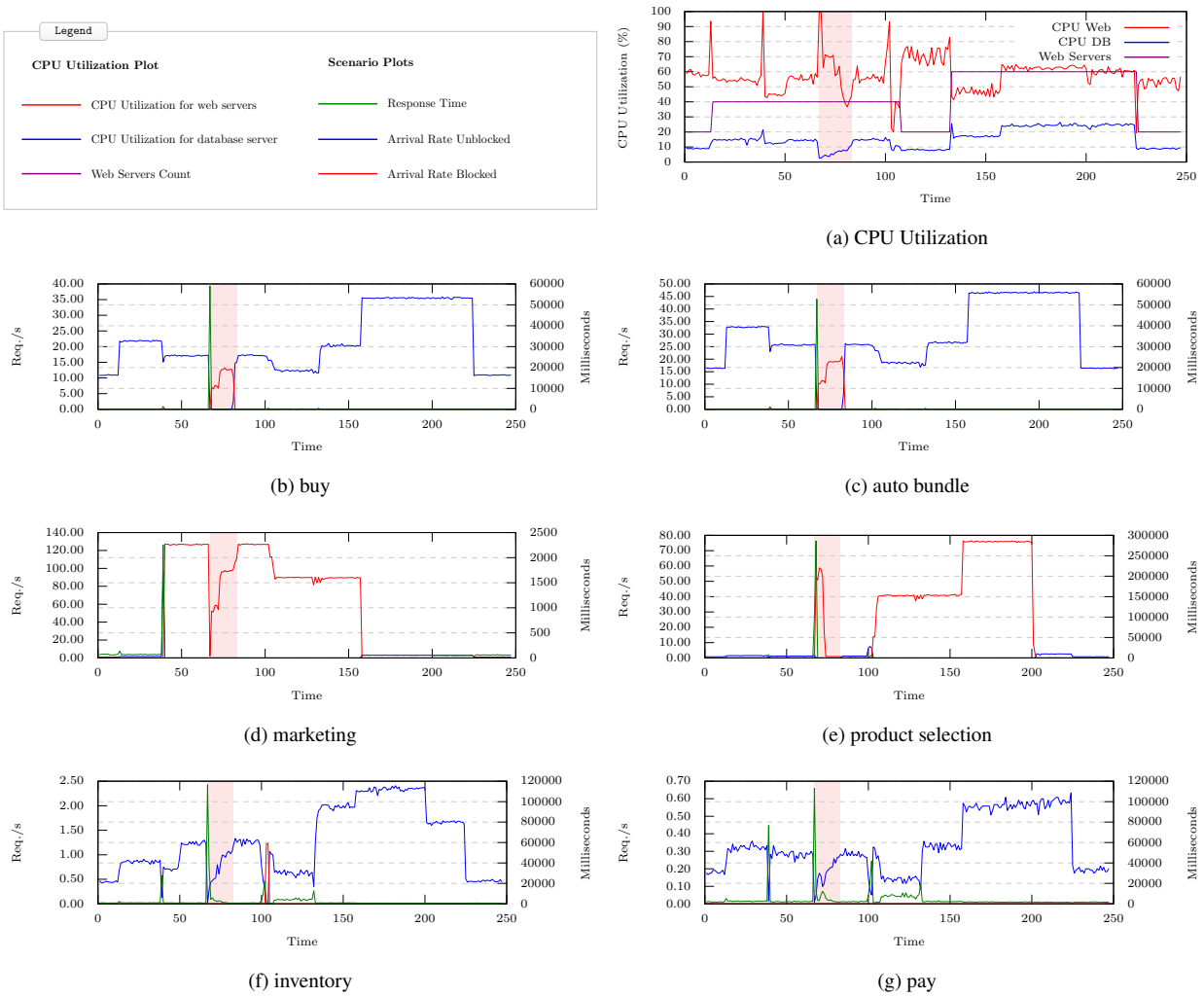
When the scenario under attack is resource-intensive, it will take multiple iterations to process the backlog of DoS requests that made it through before we started directing traffic. Although traffic to `inventory` and `pay` is not redirected, there is a significant drop in the arrival rate and an increase in response time. The drop in arrival rate is normal behaviour, because normal users will not make a new request to the server until they receive the response from their previous request.

6 Related Work

There are many approaches in achieving elasticity. Companies such as Amazon, Azure, RightScale, and Rackspace offer pre-defined rule-based autoscalers. The application owners manually define rules (often threshold based) for triggering scaling out/in actions. Then, at runtime, the autoscalers monitor the application performance metrics and evaluate them against the rules. When a rule condition becomes true, the system executes the rule's action such as adding or removing VM. Some researchers argue that specifying good threshold based rules is a challenging task [10, 29].

A potential weakness of pre-defined rule system is the thrashing effect when the system constantly adds or removes VM due to a fast changing workload. To address this problem, Iqbal et al. [13] combine rule-based scaling with statistical predictive models. Xiong et al. [31] demonstrated that an application analytic model could be used as an efficient method to identify the relationship between a number of required servers, workload and QoS.

Many researchers have designed and developed elastic algorithms without considering the cost factor; however, Han et al. [11] propose an elastic algorithm which considers both the cost and performance. An elastic al-



(a) CPU Utilization

(b) buy

(c) auto bundle

(d) marketing

(e) product selection

(f) inventory

(g) pay

Figure 7: **Experiment #4.** A DoS attack causes overcorrection due to a slow reaction, and additional classes are blocked.

gorithm will identify the application tier to scale in order to resolve the QoS issue while keeping the overall deployment cost as low as possible. A queuing analytic performance model is utilized for identification of the inefficient application tier.

The main weakness of the above approaches is that all user requests are considered desirable to the application owner. This may not be true in actual deployment environments. Many researchers and practitioners agree that DoS attacks are one of the biggest threats in the today's security landscape. Our novel approach distinguishes between desirable and undesirable traffic using cost efficiency metrics that consider not only the cost of the infrastructure, but also the business value of the workload.

7 Conclusion

This paper presented a model-driven adaptive management architecture and algorithm to scale a web application, mitigate a DoS attack, or both, based on an assessment of the business value of workload. The business value is measured through an efficiency metric as a ratio between the revenue and cost. The approach is enabled by a layered queuing network model previously used to model data centers but adapted for cloud. The model accurately predicts short-term cloud behavior, despite cloud variability over time. We evaluated our approach on Amazon EC2 and demonstrate the ability to horizontally scale a sample web application in response to an increase in legitimate traffic while mitigating multiple DoS attacks, achieving the established performance goal. We also showed the limitation of the approach which can be overcome through further work.

Acknowledgements

This research was supported by the SAVI Strategic Research Network (Smart Applications on Virtual Infrastructure), funded by NSERC (The Natural Sciences and Engineering Research Council of Canada) and by Connected Vehicles and Smart Transportation (CVST) funded by Ontario Research Fund.

References

[1] BACIGALUPO, D., VAN HEMERT, J., USMANI, A., DILLENBERGER, D., WILLS, G., AND JARVIS, S. Resource management of enterprise cloud systems using layered queuing and historical performance models. In *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on* (April 2010), pp. 1–8.

[2] BARNA, C., SHTERN, M., SMIT, M., TZERPOS, V., AND LITOIU, M. Model-based adaptive dos attack mitigation. In *ICSE Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)* (New York, NY, USA, 2012), SEAMS 2012, ACM, pp. 119–128.

[3] BARNA, C., SHTERN, M., SMIT, M., TZERPOS, V., AND LITOIU, M. Mitigating DoS attacks using performance model-driven adaptive algorithms. *Transactions on Autonomous and Adaptive Systems 9*, 1 (Mar. 2014), 3:1–3:26.

[4] BARRETT, E., HOWLEY, E., AND DUGGAN, J. Applying reinforcement learning towards automating resource allocation and application scalability in the cloud. *Concurrency and Computation: Practice and Experience 25*, 12 (2013), 1656–1674.

[5] CARON, E., RODERO-MERINO, L., DESPREZ, F., AND MURESAN, A. Auto-Scaling, Load Balancing and Monitoring in Commercial and Open-Source Clouds. Rapport de recherche RR-7857, INRIA, 2012.

[6] DOBBINS, R., MORALES, C., ANSTEE, D., ARRUDA, J., BIENKOWSKI, T., HOLLYMAN, M., LABOVITZ, C., NAZARIO, J., SEO, E., AND SHAH, R. Worldwide Infrastructure Security Report. Tech. rep., Arbor Networks, 2010.

[7] GALANTE, G., AND DE BONA, L. A survey on cloud computing elasticity. In *Utility and Cloud Computing (UCC), 2012 IEEE Fifth International Conference on* (2012), pp. 263–270.

[8] GARG, A., AND NARASIMHA REDDY, A. L. Mitigation of DoS attacks through QoS regulation. In *Quality of Service, 2002. 10^{th} IEEE International Workshop on* (Washington, DC, USA, 2002), IEEE Computer Society, pp. 45–53.

[9] GHANBARI, H., SIMMONS, B., LITOIU, M., BARNA, C., AND ISZLAI, G. Optimal autoscaling in a iaas cloud. In *Proceedings of the 9th International Conference on Autonomic Computing* (New York, NY, USA, 2012), ICAC '12, ACM, pp. 173–178.

[10] GHANBARI, H., SIMMONS, B., LITOIU, M., AND ISZLAI, G. Exploring alternative approaches to implement an elasticity policy. In *Proceedings of the 4th IEEE International Conference on Cloud Computing* (Washington DC, USA, 2011), IEEE.

[11] HAN, R., GHANEM, M. M., GUO, L., GUO, Y., AND OSMOND, M. Enabling cost-aware and adaptive elasticity of multi-tier cloud applications. *Future Generation Computer Systems 32* (2014), 82–98.

[12] IDZIOREK, J., AND TANNIAN, M. Exploiting cloud utility models for profit and ruin. In *IEEE International Conference on Cloud Computing* (2011), pp. 33–40.

[13] IQBAL, W., DAILEY, M. N., CARRERA, D., AND JANECEK, P. Adaptive resource provisioning for read intensive multi-tier applications in the cloud. *Future Generation Computer Systems 27*, 6 (2011), 871–879.

[14] KARGL, F., AND MAIER, J. Protecting web servers from distributed denial of service attacks, 2001.

[15] LI, J. Z., CHINNECK, J., WOODSIDE, M., LITOIU, M., AND ISZLAI, G. Performance model driven QoS guarantees and optimization in clouds. In *in Proceedings of Workshop on Software Engineering Challenges in Cloud Computing @ ICSE 2009* (2009).

[16] LONG, M., WU, C.-H. J., HUNG, J. Y., AND IRWIN, J. D. Mitigating performance degradation of network-based control systems under denial of service attacks. In *Proc. 30th Annual Conf. of IEEE Industrial Electronics Society IECON 2004* (Washington, DC, USA, 2004), vol. 3, IEEE Computer Society, pp. 2339–2342.

[17] MARSHALL, P., KEAHEY, K., AND FREEMAN, T. Elastic site: Using clouds to elastically extend site resources. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing* (Washington, DC, USA, 2010), CCGRID '10, IEEE Computer Society, pp. 43–52.

[18] MAURER, M., BRANDIC, I., AND SAKELLARIOU, R. Enacting slas in clouds using rules. In *Proceedings of the 17th International Conference on Parallel Processing - Volume Part I* (Berlin, Heidelberg, 2011), Euro-Par'11, Springer-Verlag, pp. 455–466.

[19] MENASCÉ, D. A. Simple analytic modeling of software contention. *SIGMETRICS Performance Evaluation Review 29*, 4 (2002), 24–30.

[20] MOREIN, W. G., STAVROU, A., COOK, D. L., KEROMYTIS, A. D., MISRA, V., AND RUBENSTEIN, D. Using graphic turing tests to counter automated DDoS attacks against web servers. In *Proceedings of the 10th ACM conference on Computer and communications security* (New York, NY, USA, 2003), CCS '03, ACM, pp. 8–19.

[21] ROLIA, J. A., AND SEVCIK, K. C. The method of layers. *IEEE Transactions on Software Engineering 21*, 8 (1995), 689–700.

[22] ROMAN, J., RADEK, B., RADEK, V., AND LIBOR, S. Launching distributed denial of service attacks by network protocol exploitation. In *Proceedings of the 2nd international conference on Applied informatics and computing theory* (Stevens Point, Wisconsin, USA, 2011), AICT'11, World Scientific and Engineering Academy and Society (WSEAS), pp. 210–216.

[23] SACHDEVA, M., SINGH, G., AND KUMAR, K. Deployment of Distributed Defense against DDoS Attacks in ISP Domain. *International Journal of Computer Applications 15*, 2 (February 2011), 25–31. Published by Foundation of Computer Science.

[24] SCHAD, J., DITTRICH, J., AND QUIANE-RUIZ, J.-A. Runtime measurements in the cloud: Observing, analyzing, and reducing variance. *Proceedings of the VLDB Endowment 3*, 1 (2010).

[25] SHTERN, M., SANDEL, R., LITOIU, M., BACHALO, C., AND THEODOROU, V. Towards mitigation of low and slow application ddos attacks. In *IEEE International Workshop on Software Defined Systems (SDS), IEEE International Conference on Cloud Engineering (IC2E)* (Accepted, 2014).

[26] SHTERN, M., SMIT, M., SIMMONS, B., AND LITOIU, M. A runtime cloud efficiency software quality metric. In *New Ideas and Emerging Results (NIER) track, Proc. of the 2014 Intl. Conference on Software Engineering (ICSE)* (Accepted, 2014).

[27] SMIT, M., SIMMONS, B., AND LITOIU, M. Distributed, application-level monitoring of heterogeneous clouds using stream processing. *Future Generation Computer Systems 29*, 8 (2013), 2103–2114.

[28] SMIT, M., AND STROULIA, E. Autonomic configuration adaptation based on simulation-generated state-transition models. In *Software Engineering and Advanced Applications (SEAA), 2011 37th EUROMICRO Conference on* (Aug 2011), pp. 175–179.

[29] SULEIMAN, B., AND VENUGOPAL, S. Modeling performance of elasticity rules for cloud-based applications. In *Enterprise Distributed Object Computing Conference (EDOC), 2013 17th IEEE International* (Sept 2013), pp. 201–206.

[30] URGAONKAR, B., SHENOY, P., CHANDRA, A., GOYAL, P., AND WOOD, T. Agile dynamic provisioning of multi-tier internet applications. *ACM Trans. Auton. Adapt. Syst. 3*, 1 (2008), 1:1–1:39.

[31] XIONG, K., AND PERROS, H. Service performance and analysis in cloud computing. In *Services-I, 2009 World Conference on* (2009), IEEE, pp. 693–700.

[32] ZHENG, T., YANG, J., WOODSIDE, M., LITOIU, M., AND ISZLAI, G. Tracking time-varying parameters in software systems with extended kalman filters. In *CASCON '05: Proceedings of the 2005 conference of the Centre for Advanced Studies on Collaborative research* (2005), IBM Press, pp. 334–345.

Integrating Adaptation Mechanisms Using Control Theory Centric Architecture Models: A Case Study

Filip Křikava
University Lille 1 / Inria

Philippe Collet
Université Nice Sophia Antipolis

Romain Rouvoy
University Lille 1 / Inria

Abstract

Control theory provides solid foundations for developing reliable and scalable feedback control for software systems. Although, feedback controllers have been acknowledged to efficiently solve common classes of problems, their adoption by state-of-the-art approaches for designing self-adaptation in legacy software systems remains limited and at best consists in *ad hoc* integrations, which are usually engineered manually.

In this paper, we revisit the *Znn.com* case study and we present an alternative implementation based on classical feedback controllers. We show how these controllers can be easily integrated into software systems through control theory centric architecture models and domain-specific modeling support. We also provide an assessment of the resulting properties, quality attributes and limitations.

1 Introduction

Feedback control is acknowledged as one of the viable solutions for self-adaptive software systems engineering [8, 31, 34]. It provides solid foundations and a systematic approach for designing reliable and robust adaptation mechanisms, *controllers*, which drive the system adaptation [3]. However, integrating such controllers into legacy software systems remains challenging [8, 10]. In particular, this requires selecting the appropriate target system outputs and control inputs (*touchpoints*), devising the actual controller design, and finally a software architecture integrating the controller into the target system [21]. As a matter of example, well-established feedback controllers for common and recurring problems (*e.g. Quality of Service* (QoS) management [2, 20] or performance guarantees [1, 3, 4]), are being integrated into target systems and tuned manually. Even though supporting tools, such as MATLAB, SIMULINK, or SYSWEAVER [35], provide code generation capabilities, the controller integration into the target system still requires an extensive handcrafting of a non-trivial code that results in significant accidental complexities. Moreover, these tools mostly target embedded real-time systems rather than distributed enterprise systems.

In this paper, we revisit the *Znn.com* case study [12], an acknowledged case study from the self-adaptive software systems community[1], and we describe an elegant solution integrating classical feedback controllers using control theory centric architecture models [27]. *Znn.com* is a web-based N-tier client-server system that models a news service provider like *cnn.com*. The main control objective is to make *Znn.com* to serve its content within acceptable response time and quality even in the event of traffic spikes caused by highly popular news by using *content adaptation* (*e.g.* serving reduced content quality). This paper contributes to demonstrate a systematic integration of a control theory based approach that addresses the *Znn.com* control objective.

Our solution is based on a technologically agnostic *Domain-Specific Modeling Language* (DSML) for defining *Feedback Control Loops* (FCLs). It supports composition, distribution and reflection, thereby enabling coordination and composition of multiple distributed FCLs using control schemes. It raises the level of abstraction at which the FCL architectures are defined, and a support is provided for automated implementation code synthesis and verification. The application to *Znn.com* enables us to demonstrate the model capabilities to progressively refine adaptation mechanisms, going from local content delivery adaptation using an existing and proven control algorithm [2, 3] to distributed content adaptation, and finally to adaptive control.

2 Related Work

IBM proposed MAPE-K decomposition of a FCL [24] which has become a widely referenced model for autonomic systems, followed by number of framework-based approaches [34]. *The Rainbow framework* [18] provides

[1] http://www.hpi.uni-potsdam.de/giese/public/selfadapt/exemplars/model-problem-znn-com

an architecture-based approach for self-adaptive software systems using utility theory for an optimal adaptation strategy selection. *The DYNAMICO model* [38] defines a fixed three-layers architecture with three FCLs for managing control objectives, target system adaptation and dynamic monitoring. *The StarMX framework* [6] designs self-managing Java-based applications using JMX for target system touchpoints and a policy-rule language for adaption engine. *The Zanshin Framework* [5] uses requirements engineering and goal models for self-adaptive software development.

The advantage of the above framework based solutions is that they provide an architecture basis of an application and therefore they can simplify its development. However, the adaptation mechanisms within these frameworks mostly use a simple, fixed threshold-based event-condition actions, not providing support for control theory based controllers. For example, both DYNAMICO and Zanshin implements *Znn.com* decision policies by using simple conditions such as `experiencedRespTime > MAX_RESPTIME` [11, p. 187]. As a result, the target system is more likely to experience instability due to oscillations (*e.g.* continuously enlisting and discharging servers). Furthermore, frameworks always impose the use of a specific technological stack. The level of abstraction and formal reasoning is also usually limited since the adaptation is an integral part of the implementation. Finally, except DYNAMICO, they are primarily designed for scenarios that can be solved by centralized control loop and do not allow hierarchical control schemes nor adaptive control as they do not support runtime modifications of adaptation strategies or thresholds.

The *model@run.time* approaches are using models to represent abstractions of running systems and MDE techniques for their adaptation at runtime [15]. For example, Vogel *et al.* [40] propose runtime executable megamodels with a language for adaptation logic modeling and a runtime interpreter. Similarly to our approach they also support hierarchically organization and FCL coordination, however, they present only a high-level overview of how the actual adaptations look like.

A lot of effort has been also invested in tools for engineering feedback control for real-time embedded systems. *Ptolemy II* [13] is an extensive framework for the simulation of concurrent actor-oriented systems allowing to combine heterogeneous models of computation. We follow a similar actor-oriented approach and our execution semantics is derived from Ptolemy push-pull model of computation (*cf.* Section 3.2). However, Ptolemy focus rather on simulation of the executable models and their transformations to the embedded systems. SIMULINK is an industry standard tool for developing feedback control targeting primarily embedded systems. SYSWEAVER extends SIMULINK code generation capabilities for distributed real-time systems.

3 Control Theory Centric Architecture Models

This section outlines our approach for integrating adaptation mechanisms into software systems through control theory centric architecture models. A detailed description is provided in [26].

3.1 Principles and Design Decisions

Generality (applicability to a wide range of target platforms and adaptation scenarios), *visibility* (explicit FCLs, their processes and interactions), and *composability* (fine-grained reusable elements representing the FCL processes) are all well-identified requirements for FCL engineering [8, 10, 32, 34]. In order to meet these requirements, we structure the approach around a DSML with an actor-oriented design. The key advantage of a DSML is the possibility to raise the level of abstraction at which the FCLs are described and directly use the FCL domain concepts. Moreover, DSMLs are particularly suitable for automated reasoning and implementation code synthesis [25]. Since FCLs are inherently concurrent, we choose an actor-oriented design [22] representing the FCL processes as message-passing actors. The actor model allows to implement FCLs without worrying about thread safety, it is scalable [19] and seamlessly supports remote distribution.

For illustration, we use the Apache overload control FCL (*cf.* Figure 1) from Hellerstain *et al.* [21, §4.6.2][2], which can be considered as a simple *Znn.com* adaptation mechanism. It adjusts the maximum number of simultaneous connections (MC) based on the difference between reference (MEM^*) and actual (MEM) memory usage.

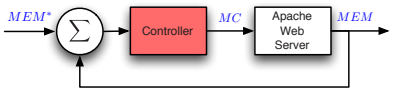


Figure 1: Apache overload control block diagram

3.2 Feedback Control Definition Language

Our approach is based on an actor-oriented component meta-model for representing FCLs abstractions, called *Feedback Control Definition Language* (FCDL) [27]. The components are actor-like entities called *Adaptive Elements* (AE) that are connected into hierarchically composed networks that form closed FCLs.

Syntax. An AE defines properties and input/output ports through which it communicates with other AEs using either data-driven (*push*) and demand-driven (*pull*) mode.

[2]For simplicity, we only use the case with one controller.

Once an AE receives a message, it executes its associated behavior whose result may or may not be sent further to the connected downstream elements which in turn will cause them to react and so on and so forth. An AE can be *passive*, *i.e.* triggered by a message, or *active*, *i.e.* triggered by an external event (*e.g.* a file modification). The ports and properties data values are statically typed and FCDL further supports parametric polymorphism. We recognize the following types of AE: *a sensor* (raw information collection), *an effector* (changes propagation), *a processor* (data processing and analyzing), and *a controller* (decision making). FCDL also contains a *composite* type that can be created from both atomic AEs and other composites. It can define ports, which are used to promote ports of the contained elements. Furthermore, a composite is also the primary unit of deployment.

Figure 2 shows an FCDL model implementing the FCL from Figure 1. The figure uses an informal FCDL graphical notation (a formal textual syntax is presented further in Section 4).

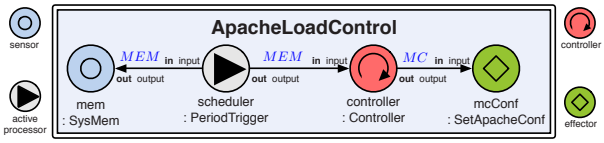


Figure 2: A FCDL model of Apache overload control

The `PeriodicTrigger` is an active processor. It periodically pulls memory utilization (*MEM*) from `SysMem` sensors and in turn pushes the value to the `Controller` that computes a new *MC* configuration to be applied by the `SetApacheConf` effector. The MEM^* value is modeled as a property of the controller.

Conceptually, each AE can be seen as a target system itself, and as such it can provide sensors and effectors enabling the AE reflection. This is a crucial feature permitting to hierarchically organize multiple FCL [36] in an uniform way and therefore realize complex control schemes from elementary building blocks.

Semantics. The execution semantics is based on the Ptolemy [13] push-pull model of computation [42]. We further adapt a notion of *Interaction Contracts* (IC) to precisely define allowed interactions of AE [9]. An IC specifies what ports activate an AE, what inputs might be pulled during AE execution, and what outputs might push results. For example, the IC associated with `PeriodicTrigger` is $\langle self; \Downarrow (\texttt{input}); \Uparrow (\texttt{output?}) \rangle$. It denotes an interaction caused by a *self* activation, pulling data from the `input` port and conditionally pushing data to the `output` port. ICs allow for asserting certain architectural properties (*e.g.*, consistency, determinacy, completeness) and they denote the type of the associate activation function making the generated source code both *prescriptive* (guiding developers) and *restrictive* (limiting developers to what the architecture allows).

4 Application to ZNN.COM

The main *Znn.com* control objective is content adaptation whereby the delivered content quality (*e.g.* degraded image quality) is reduced when the server is under heavy load. This has been well studied by Abdelzaher *et al.* [1–3], providing a control theoretic approach, which we integrate into *Znn.com* using FCDL.

4.1 Local Content Delivery Adaptation

The aim of the adaptation is to maintain web server load at a certain pre-set value. The server content is preprocessed and stored in M trees where each one offers the same content, but of a different quality and therefore size. At runtime, a given URL request, *e.g.* `photo.jpg`, is served from either `/full/photo.jpg` or `/degraded/photo.jpg` depending on the current load of the server. Since the resource utilization is proportional to the size of the content delivered, offering the content from the degraded tree helps to reduce the server load.

Controller Design. Abdelzaher *et al.* [2,3] proposes two controllers: a simple integral controller and a more sophisticated proportional integral controller. Due to the space limitations, in this paper we only consider the former one, however, from the software architecture perspective, the only difference between them is the type of AE that is instantiated. The focus of FCDL is to facilitate the controller integration into software system not to develop of the controller itself.

The controller input is the web server utilization $U = aR + bW$ that is periodically computed using request rate $R = \frac{\sum r}{t}$ and delivered bandwidth $W = \frac{\sum w}{t}$, where a and b are platform constants[3] and $\sum r, \sum w$ are the number of requests and the amount of bytes sent over some period of time t, respectively. The controller output is the severity of the adaptation action $G = G + K_I E = G + K_I(U^* - U)$ where K_I is the controller integral gain, U^* is the target utilization (set by a system administrator) and U is the observed utilization. It determines which content tree should be used ranging from $G = M$, servicing all requests using the highest quality content tree to $G = 0$ in which case all requests are rejected.

Architecture. Figure 3 shows one possible integration of the above controller into the target system using FCDL.

For the *decision-making* part we create an AE, `IController`, that implements a general integral controller. Once a new value (U) is pushed into its input, it computes and pushes the control input (G). Both the integral gain (K_I) and the reference input (U^*) are represented as the controller properties. The *monitoring part* periodically computes server utilization U. Both the R

[3] *cf.* Abdelzaher *et al.* [2,3]

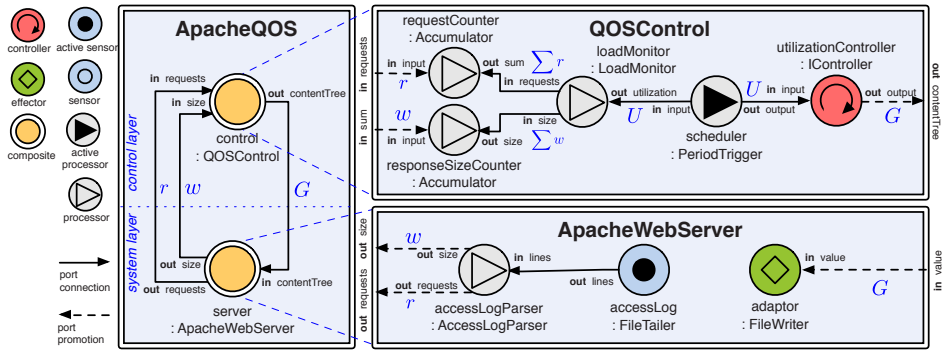


Figure 3: Apache content delivery control

and W can be obtained from Apache access log file. We create an active sensor, `FileTailer`, that activates every time a file content changes pushing out the modified part. The connected `AccessLogParser` extracts the number of requests r, the size of the responses w and pushes the values into the connected counters `requestCounter` and `responseSizeCounter`. To compute utilization U, the sum of requests $\sum r$ and response size $\sum w$ has to be converted into request rate R and bandwidth W—*i.e.*, the number of requests and sent bytes over certain time period t. We reuse the periodic trigger, which by pulling its input causes `LoadMonitor` to compute U using the accumulated $\sum r$, $\sum w$ sums. In the *reconfiguration part*, the `FileWriter` updates the web server URL rewrite rules reflecting the newly computed content tree.

To demonstrate composition, the presented elements are assembled into three composites `ApacheQOS`, `QOSControl` and `ApacheWebServer`, representing the main composite that will be deployed, the control, and the target system, respectively. This makes a clear separation of concerns and easy to switch from web server implementation to another.

Implementation. FCDL models are implemented in a domain-specific language called *Extended Feedback Control Definition Language* (XFCDL). It is a textual DSL for authoring FCDL models that further supports modularization and AE implementation using a Java-like expression language Xbase[4]. Listing 1 shows an excerpt[5] of the `IController` AE. Line 1 defines a new active polymorphic processor type with data type parameter T, followed by ports declaration (lines 2-4) and property definition (line 6). Line 7 specifies an IC and line 10 provides its implementation directly in Xbase.

```
1  active processor PeriodicTrigger<T> {
2    push in port output: T
3    pull in port input: T
4    self port selfport: long // self port for self-activation
5
6    property initialPeriod: Duration = 10.seconds
7    act activate(selfport; input; output?)
8
9    implementation xbase {
10     act activate { output.put(input.get) }
11   }
12 }
```

Listing 1: XFCDL code of `PeriodicTrigger` AE

4.2 Distributed Adaptation

Next, we extend the adaptation to cover distributed *Znn.com* deployment on a pool of replicated servers with a load balancer.

Controller Design. The distributed deployment consists of a server pool S with n servers and one load balancer. Each server S_i runs locally the previously developed `ApacheQOS` FCL computing its target content tree G_i. In order to maintain the highest QoS, the load balancer dynamically schedules the arriving requests to a server $s \in S$ that provides the least degraded content: $\texttt{content\_tree}(s) = \texttt{max}(\texttt{content\_tree}(S))$.

Architecture. Figure 4 depicts the FCL architecture representing the distributed control. The `LocalApacheQOS` runs at each of the server S_i, encapsulating the local `ApacheQOS` FCL. The `LoadBalancerControl` runs on the load balancer controlling the scheduler using the above equation.

The load balancer FCL first collects the content tree (G) status of all the participating servers using distributed publish/subscribe event bus. An advantage of using an event bus is that it does not need to be *a priori* aware of all the participating servers. In FCDL, an event bus is facilitated by two AEs: the publisher (`EventBusPublisher`) and the subscriber (`EventBusSubscriber`). We use key-value tuples of servers S_i (server hostname) with their corresponding content trees G_i. The G_i is obtained from a newly promoted `ApacheQOS` port `contentTree` so that the G is available from the outside. The pushed (S_i, G_i) entries are received by the `EventBusSubscriber` and aggregated using the `MapStore` AE, which is a map storage. The server with the highest G is selected by the `MapMaxKey` AE and consequently used to update the load balancer scheduling rules.

[4] A statically typed Java-like expression language `http://bit.ly/1mr36bt`

[5] The complete XFCDL code is available from the companion website `http://fikovnik.github.io/Actress/ICAC14.html`

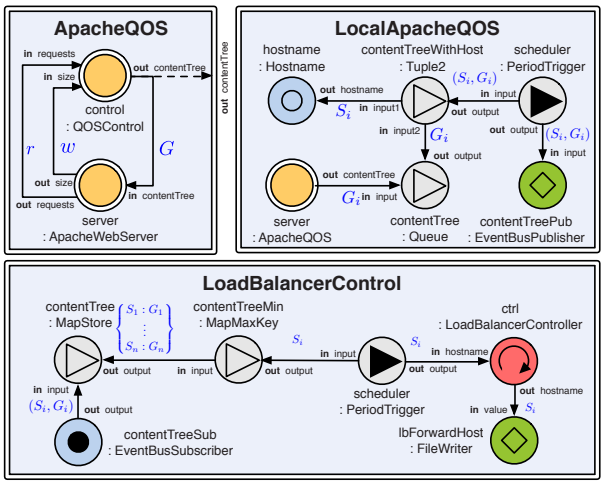


Figure 4: Distributed QoS Management Control FCLs

4.3 System Identification

Controllers for software systems are usually driven by *"black box"* models derived from experimental runs collecting data and statistical model constructions. An experimental run consists of observing the effect of control inputs on the measured outputs. In FCDL, this can be facilitated by designing an open loop architecture in which target system touchpoints are used to set control inputs and observe/log corresponding system outputs. For example, Figure 5 shows an architecture model for tuning the controller from Section 4.1 into an open loop that can exercise the system on a range of inputs and log its outputs. Instead of connecting a controller output into the `ApacheWebServer` content tree input, we connect it directly to a value generator, a discrete sine wave.

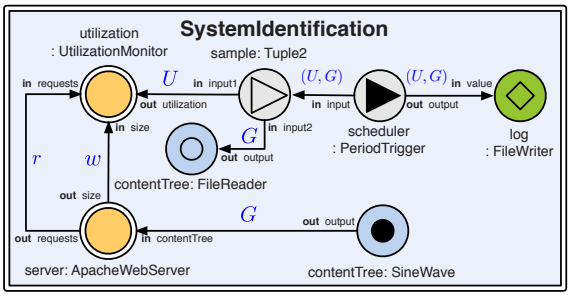


Figure 5: Apache content delivery control. The `UtilizationMonitor` contains the `requestCounter`, `responseSizeCounter` and `loadMonitor` elements from Figure 4.

4.4 Adaptive Control

An adaptive control improve FCL portability to load conditions and platform resource capacities that have not been anticipated during the system identification [4]. In FCDL, an adaptive control is facilitated by the model reflection. Figure 6 depicts an architecture of adaptive control for local content delivery adaptation FCL (*cf.* Section 4.1) that reuses part of the system identification developed above.

The aim is to perform an online profiling of the target system (relation between U and G), based on which we estimate the controller parameters (K_I). First the `IController` is extended with a provided effector to allow to change K_I at runtime. Next, we reuse the part of the architecture developed for the system identification and we create an `AdaptiveController` for the parameter estimation. It can be implemented using an adaptive controller as shown by Lu *et al.* [29] or by constructing a dynamic system model as proposed by Filieri *et al.* [37]. Finally, we encapsulate the corresponding elements into a new composite `AdaptiveControl` that can be placed on the top of the FCL developed in Section 4.1.

Adaptive control is one example of the FCDL reflection capabilities, which can also be used to design adaptive monitoring, or to organize multiple FCLs using various control schemes, such as hierarchical control.

5 Assessment and Discussion

In this section we assess our approach by discussing its properties and quality attributes. As such, the assessment is rather qualitative since we do not evaluate the controllers themselves.

Implementation. To facilitate the development using FCDL, we have implemented a prototype of a Java/EMF [14] based modeling environment called ACTRESS [27]. It provides support for FCDL modeling, verification (user defined constraints and temporal properties) and source code generation together with runtime platform.

Properties. The generality is addressed by using a fine-grained FCL decomposition which should be usable in any domain for various adaptation property. FCDL is built as a technologically-agnostic model and the Java based ACTRESS implementation provides only one technological solution. Generality is also obtained in adaptation scenarios, as they are captured at a conceptual level using the problem domain concepts, rather than the implementation concepts. The visibility property is tackled by having all the FCL processes represented as first-class entities with explicit interactions that are precisely guided by interaction contracts. Finally, we have shown that FCLs are composed from clearly structured fine-grained AEs using ICs to guide AE interactions and implementations (cf. Section 4.1).

Quality Attributes. Among many software quality attributes, the following are relevant for evaluating self-adaptive engineering approaches and have been already used by others, *e.g.* Asadollahi *et al.* [6].

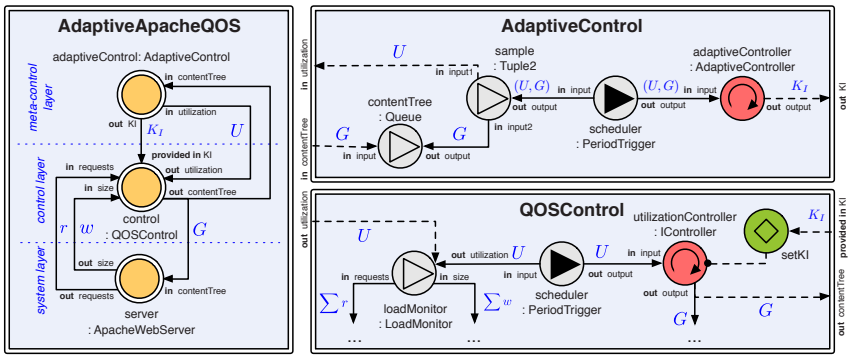


Figure 6: Adaptive control for Apache content delivery controller

– *Flexibility.* FCDL can represent both closed and open control loops and AE reflection allows for designing complex control schemes. Unlike most frameworks, FCDL does not dictate any system architecture nor any specific technology. Furthermore, it promotes separation of concerns in the sense that the FCL architecture and control mechanisms may be defined by control engineers while the technical/system-level processors and touchpoints may be implemented by software engineers. Next to the *Znn.com* case study, FCDL was also used to build overload control adaptation scenarios in the domain of high-throughput computing [26, §8.1].
– *Scalability.* The FCDL support for composition, polymorphic data types and ICs allowed us to incrementally refine the needed FCLs throughout the case study. These techniques are likely to allow for building larger models.
– *Usability.* Our approach relies on known concepts, as FCDL is using notions from control theory and component-based software engineering, xFCDL follows known concepts from Java, and the ACTRESS modeling environment is integrated in the Eclipse IDE, which might simplify adoption for the users already familiar with it. Furthermore, using on the actor-model simplifies AE implementation without the need to protect mutable states [19]. The implementation effort varies between 200-300 xFCDL lines of code per scenario[6].
– *Reusability.* There are two features that contributes to AE reusability: the FCDL support for data type polymorphism and the Xbase support for lambda expressions that allows to use functions types as properties. This results in higher-order polymorphic AEs definitions.
– *Extensibility.* FCDL and xFCDL are both defined using their respective EMF meta-models. Therefore, extending their core functionality is only possible by modifying the ACTRESS source code. On the other hand, thanks to MDE, it is possible to use the FCDL models and target different systems, providing new code generators, verification techniques and the like.
– *Performance.* The ACTRESS runtime is based on Akka[7] which with no AE deployed accounts for 1.5MB[8]. The memory overhead is about 400 bytes per actor instance with a possible throughput of 50 million messages per sec on a single machine[9]. The size and the CPU time of an AE is mostly affected by the amount of state it keeps and the complexity of its activation methods. However, the main potential performance issues are in the indirect load caused by the sensors and effectors.

6 Conclusions

While control theory provides solid foundations for designing self-adaptive systems, its mapping into implementation artifacts often results in the development of dedicated assets (*e.g.* code, models) which inevitably prevents their reuse and adoption at a larger scale. To overcome this limitation, we define FCDL, a domain-specific modeling language for integrating adaptation mechanisms into legacy software systems. We demonstrated its use on an implementation of local and distributed content delivery adaptation and distributed resource management. FCDL is a domain-specific and technologically-agnostic architecture model that provides an actor-based programming model.

Currently we are focusing in carrying more case studies, in particular targeting different self-adaptive properties and improvements such as support for distributed deployment and failure propagation. For future work, we intend to investigate adaptive feedback controllers and the principles of defensive programming in order to better control the execution of feedback control loops.

Acknowledgments. This work is partially supported by the Datalyse project `www.datalyse.fr`.

[6] The complete xFCDL code is available from the companion website `http://fikovnik.github.io/Actress/ICAC14.html`

[7] `http://akka.io`

[8] MacBook Pro 2.53 Ghz, 8GB RAM, Java 1.7_17 64bit, Akka 2.2

[9] `http://bit.ly/1gHM975`

References

[1] T. Abdelzaher. Modeling and performance control of Internet servers. In *39th IEEE Conference on Decision and Control*, volume 3, IEEE, 2000.

[2] T. Abdelzaher and N. Bhatti. Web server QoS management by adaptive content delivery. In *7th International Workshop on Quality of Service*, IWQoS, London, 1999. IEEE.

[3] T. Abdelzaher, K. Shin, and N. Bhatti. Performance guarantees for Web server end-systems: a control-theoretical approach. *IEEE Transactions on Parallel and Distributed Systems*, 13(1):80–96, 2002.

[4] T. Abdelzaher and J. Stankovic. Feedback Control Architecture and Design Methodology for Service Delay Guarantees in Web Servers. *IEEE Transactions on Parallel and Distributed Systems*, 17(9):1014–1027, Sept. 2006.

[5] K. Angelopoulos, V. E. Silva Souza, and J. Pimentel. Requirements and architectural approaches to adaptive software systems: A comparative study. In *8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS, IEEE, May 2013.

[6] R. Asadollahi, M. Salehie, and L. Tahvildari. StarMX: A framework for developing self-managing Java-based systems. In *2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, Ieee, May 2009.

[7] K. J. Astöm and B. Wittenmark. Adaptive Control, 1995.

[8] Y. Brun, G. Di Marzo Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, H. Müller, M. Pezzè, and M. Shaw. Engineering Self-Adaptive Systems Through Feedback Loops. *Software Engineering for Self-Adaptive Systems*, 2009.

[9] D. Cassou, E. Balland, C. Consel, and J. Lawall. Leveraging software architectures to guide and verify the development of sense/compute/control applications. In *33rd International Conference on Software Engineering*, ICSE, 2011.

[10] B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, G. Di Marzo Serugendo, S. Dustdar, A. Finkelstein, C. Gacek, K. Geihs, V. Grassi, G. Karsai, H. Kienle, J. Kramer, M. Litoiu, S. Malek, R. Mirandola, H. Müller, S. Park, M. Shaw, M. Tichy, M. Tivoli, D. Weyns, J. Whittle, R. Lemos, and Others. Software Engineering for Self-Adaptive Systems: A Research Roadmap. *Software Engineering for Self-Adaptive Systems*, 2009.

[11] S.-w. Cheng. *Rainbow: Cost-Effective Software*. PhD thesis, Carnegie Mellon University, 2008.

[12] S.-W. Cheng, D. Garlan, and B. Schmerl. Evaluating the effectiveness of the Rainbow self-adaptive system. In *4th ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS, IEEE, May 2009.

[13] J. Eker, J. Janneck, E. Lee, J. Ludvig, S. Neuendorffer, and S. Sachs. Taming heterogeneity - the Ptolemy approach. *IEEE*, 2003.

[14] Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework (2nd Edition). Addison-Wesley Professional (2008)

[15] R. B. France and B. Rumpe. Model-driven Development of Complex Software: A Research Roadmap. In *Future of Software Engineering*, FOSE, 2007.

[16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[17] N. Gandhi, J. Hellerstein, S. Parekh, and D. Tilbury. Using MIMO feedback control to enforce policies for interrelated metrics with application to the Apache Web server. In *EEE/IFIP Network Operations and Management Symposium.*, pages 219–234. IEEE, 2002.

[18] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste. Rainbow: architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, 2004.

[19] P. Haller and M. Odersky. Scala Actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2-3):202–220, Feb. 2009.

[20] T. He, J. a. Stankovic, M. Marley, C. Lu, Y. Lu, T. Abdelzaher, S. Son, and G. Tao. Feedback control-based dynamic resource management in distributed real-time systems. *Journal of Systems and Software*, 2007.

[21] J. Hellerstein, Y. Diao, S. Parekh, and D. Tilbury. *Feedback control of computing systems*. Wiley Online Library, 2004.

[22] C. Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8(3):323–364, June 1977.

[23] G. J. Holzmann. *Spin Model Checker*. Addison-Wesley Professional, 1. edition edition, 2003.

[24] IBM. An Architectural Blueprint for Autonomic Computing, 4. edition. Technical report, IBM, 2006.

[25] S. Kelly and J.-P. Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley-IEEE, 2008.

[26] F. Krikava. *Domain-Specific Modeling Language for Self-Adaptive Software System Architectures*. PhD thesis, University of Nice Sophia-Antipolis, 2013.

[27] F. Krikava, P. Collet, and R. France. ACTRESS: Domain-Specific Modeling of Self-Adaptive Software Architectures. In *Symposium on Applied Computing (SAC), track on Dependable and Adaptive Distributed Systems (DADS)*, 2014.

[28] E. A. Lee. The Problem with Threads. *Computer*, 2006.

[29] Y. Lu, T. Abdelzaher, C. Lu, and G. Tao. An adaptive control framework for QoS guarantees and its application to differentiated caching. In *10th International Workshop on Quality of Service*, IWQoS, IEEE, 2002.

[30] M. Luckey, B. Nagel, C. Gerth, and G. Engels. Adapt cases. In *6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS, page 30, New York, New York, USA, 2011. ACM Press.

[31] M. Maggio, H. Hoffmann, M. D. Santambrogio, A. Agarwal, and A. Leva. Decision making in autonomic computing systems. In *8th ACM international conference on Autonomic computing*, ICAC, 2011.

[32] H. Müller, M. Pezzè, and M. Shaw. Visibility of control in adaptive systems. In *Proceedings of the 2nd international workshop on Ultra-large-scale software-intensive systems*, ULSSIS, 2008.

[33] A. J. Ramirez and B. H. C. Cheng. Design patterns for developing dynamically adaptive systems. In *2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS, 2010.

[34] M. Salehie and L. Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 4(2):1–42, 2009.

[35] D. Niz, G. Bhatia and R. Rajkumar. Model-Based Development of Embedded Systems: The SysWeaver Approach. *12th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2006.

[36] J. Kephart and D. Chess. The Vision of Autonomic Computing. *Computer 36(1)*, 2003.

[37] A. Filieri, H. Hoffmann and M. Maggio. Automated Design of Self-Adaptive Software with Control-Theoretical Formal Guarantees. *Proc. 36th International Conference on Software Engineering*, 2014.

[38] G. Tamura, N. M. Villegas, H. A. Muller, L. Duchien, and L. Seinturier. Improving context-awareness in self-adaptation using the DYNAMICO reference model. In *8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS, 2013.

[39] N. M. Villegas, H. A. Müller, G. Tamura, L. Duchien, and R. Casallas. A framework for evaluating quality-driven self-adaptive software systems. In *6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, volume 1 of *SEAM*, page 80, New York, New York, USA, 2011. ACM Press.

[40] T. Vogel and H. Giese. A Language for Feedback Loops in Self-Adaptive Systems: Executable Runtime Megamodels. In *7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, number 3 in SEAMS, IEEE, June 2012.

[41] K. Wu, D. J. Lilja, and H. Bai. The Applicability of Adaptive Control Theory to QoS Design: Limitations and Solutions. In *19th International Parallel and Distributed Processing Symposium*, IPDPS, pages 272b–272b. IEEE, 2005.

[42] Y. Zhao. A Model of Computation with Push and Pull Processing. Technical report, Technical Memorandum UCB/ERL M03/51, University of California, Berkeley, 2003.

ShuttleDB: Database-Aware Elasticity in the Cloud

Sean Barker [1,*], *Yun Chi* [2,*], *Hakan Hacıgümüş* [3], *Prashant Shenoy* [1], *Emmanuel Cecchet* [1]

[1] *School of Computer Science, University of Massachusetts Amherst, Amherst, Massachusetts, USA*

{sbarker,shenoy,cecchet}@cs.umass.edu

[2] *Square Inc., San Francisco, California, USA*

layunchi@gmail.com

[3] *NEC Laboratories America, Cupertino, California, USA*

hakan@nec-labs.com

Abstract

Motivated by the growing popularity of database-as-a-service clouds, this paper presents ShuttleDB, a holistic approach enabling flexible, automated elasticity of database tenants in the cloud. We first propose a database-aware live migration and replication method designed to work with off-the-shelf databases without any database engine modifications. We then combine these database-aware techniques with VM-level mechanisms to implement a flexible elasticity approach that can achieve efficient scale up, scale out, or scale back for diverse tenants with fluctuating workloads. Our experimental evaluation of the ShuttleDB prototype shows that by applying migration and replication techniques at the tenant level, automated elasticity can be achieved both intra- and inter-datacenter in a database agnostic way. We further show that ShuttleDB can reduce the time and data transfer needed for elasticity by 80% or more compared to tenant-oblivious approaches.

1 Introduction

In recent years, online applications have increasingly migrated to cloud platforms, which use data centers to provide computing and storage resources to these applications. Databases are no exception to this trend, and many services have been built on the idea of the "database cloud." Examples include Amazon RDS [1] and Google Cloud SQL [19], which expose cloud-based relational databases to client applications, and Salesforce, which employs a shared database used by many independent customers. There are numerous advantages to the cloud-based model, such as the pay-as-you-go model, where resources are billed and paid for on a fine-grain usage basis, and flexible resource allocation, where computing and storage can be dynamically increased or decreased based on an application's changing workload needs.

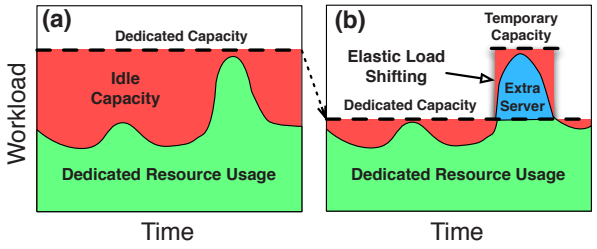


Figure 1: Provisioning for peak demand (left) may result in lower utilization and higher costs versus an elastic approach (right) requiring fewer dedicated resources.

The key to realizing many of these benefits is *elasticity* – the ability of the cloud platform to adjust an application's resource allotment on the fly when responding to long-term workload growth, seasonal variations, or sudden load spikes. Since workload peaks tend to be transient in nature, a priori static provisioning for the peak results in substantial waste as resources sit unused at non-peak times—typical server utilization in real data centers has been estimated at only 5-20% [4]. An example of this issue is shown in Figure 1. Figure 1(a) shows an example workload serviced by traditional dedicated resources. However, if the system is elastic, as shown in Figure 1(b), we can simply shift load to a new server during peak demand, and thus reduce the dedicated resources needed to service the workload.

Database Elasticity. Despite their widespread use, databases present some of the greatest difficulties in supporting elasticity. These difficulties largely stem from the specific requirements of relational databases, such as transactions and ACID compliance, and have led some to label the SQL database as the "Achilles heel of cloud elasticity" [20]. While most modern databases support clustering and replication (e.g., products such as MySQL Cluster), these systems are seldom designed to dynamically grow or shrink, and often introduce configuration,

*This work was performed while the author was at NEC Labs America. This research was supported in part by NSF grant CNS-1117221 and a gift from NEC Labs America.

management, or performance overhead [15].

While it is possible to encapsulate databases into virtual machine containers and use "standard" VM-based cloud elasticity mechanisms such as VM migration or VM replication (e.g., Amazon's auto-scaling [2]), the approach does not work well for many database cloud scenarios. For example, shared hosting scenarios (e.g., Salesforce cloud) are designed to collocate large numbers of (small) database tenants on single servers, which is not well suited to a VM-per-tenant elasticity model. If multiple tenants share single VMs to counteract the impact of hosting many VMs, then elasticity of the server is limited when using VM-level techniques, since individual tenants can no longer be migrated or replicated. VM-level mechanisms may also be unnecessarily heavyweight [24] for a "standard" application such as a database server that may not require significant customization of the underlying system.

Alternatives to VM-based black-box elasticity for databases include NoSQL systems (e.g., key-value stores such as BigTable [10] or Dynamo [16]), or augmenting the database engine itself to support elasticity (e.g., Zephyr [18], Albatross [14], and RemusDB [23]). However, these approaches introduce significant added complexity within the database itself and may change the behavior observed by client applications (e.g., the possibility of transaction failures during migrations [18]). In some ways, however, multitenant databases are well suited to application-level elasticity; for example, each database tenant may be viewed as a lightweight container, in the same way that a single VM is the unit of elasticity in VM migration.

Ideally, database elasticity should be based on migration and replication—like VM elasticity—and be transparent to the application and database engine, just as VM elasticity is transparent to the application and the OS.

Contributions. In this paper, we present ShuttleDB, a flexible system combining virtual machine elasticity with lower-level, database-aware elasticity to provide efficient database elasticity both within and across cloud data centers. Our primary contributions are threefold:

1. We examine the dichotomy between high-level VM migration and low-level DB-aware migration to decide *when* each approach is more appropriate. In particular, we identify two primary system dimensions determining what type of elasticity is appropriate—tenant type/size (i.e., collocated vs dedicated) and network type (i.e., LAN vs WAN). On this basis, for each database tenant that requires to be elastically scaled, ShuttleDB determines (a) whether to use high-level VM elasticity or low-level DB-aware elasticity, and (b) whether to scale up, scale out or scale back.

2. We present a specific technique for database-aware live migration, allowing migration of individual database tenants among servers without incurring the full cost of virtual machine migration. In contrast to most existing work, our technique requires no changes to the database engine, relying only on the presence of standard hot backup tools, and can be done live without any database down time. Additionally, performance optimizations enable efficient operation across wide-area networks.

3. We present a prototype implementation of ShuttleDB as a elasticity middleware that uses an off-the-shelf DBMS and virtualization platform. We empirically evaluate our system and demonstrate that it automatically achieves efficient elasticity under a variety of system conditions. In particular, we find that it provides tenant migration with minimal client workload delays and can reduce the time and data transfer needed for elasticity by 80% or more compared to tenant-oblivious approaches.

2 Database Clouds and Elasticity

Database Clouds. We assume the environment of a cloud providing a "database as a service" to its customers. In a database cloud, each customer rents a database from the cloud and manages it as a normal relational database, while the cloud is responsible for ensuring high performance. In particular, we assume that the cloud has a pool of physical, virtualized servers spread across one or more data centers. Each server houses one or more virtual machines (VMs), and each VM in turn houses one or more database tenants. In our work, a *tenant* is defined simply as an independent consumer of resources — most likely a single user or customer servicing a particular application. Multiple tenants may reside on individual servers so as to maximize server utilization and minimize hardware costs.

Small tenants (those with little data and/or small workloads) may be co-located within a single VM to avoid incurring the memory overhead of housing many VMs with individual OSes. Within a VM, all collocated tenants are assumed to share the same database process, which has been advocated in order to maximize resource sharing [13]. As a illustrative example, we executed a simple workload across 10 tenants on a server, first sharing an instance between all tenants (single-process), then with 10 separate processes with equal resource shares (multi-process). Transaction latencies as we scale the aggregate workload in both cases are shown in Figure 2. As the server becomes heavily loaded, we see that the multi-process model quickly falls behind, eventually showing 3x higher latency than the single-process model.

Large tenants, on the other hand, are housed in a dedicated virtual machine, which enables them to use all of the CPU and storage resources allocated to the VM. This model is analogous to EC2 compute clouds or shared hosting scenarios such as Salesforce [26].

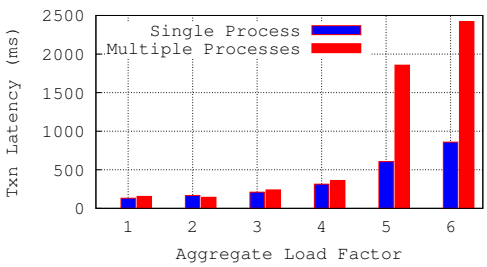


Figure 2: Single-process multitenancy outperforms multi-process multitenancy under load.

Tenant workloads are dynamic and can exhibit temporal variations or sudden spikes. In response to such variations, an important function of the cloud is to *elastically* scale the resources available to tenants. The mechanisms for providing this elasticity can be divided into two primary categories: VM-level or database-level approaches.

VM-level mechanisms: In the simplest case, when a tenant is about to experience an overload, its virtual machine can be migrated to a larger physical server and the VM can be allocated additional resources as needed. Such VM migration can be performed on-demand, transparently, and without noticeable downtime [11]. Although widely supported by virtualization platforms, however, VM migration is is wasteful for small tenants. Since many small tenants share a single virtual machine, moving the underlying VM and its disk moves *all* resident tenants, even if only one tenant is experiencing the increased workload. The simplicity of VM migration is, however, useful for large tenants housed in dedicated VMs – in this case, although the migration moves both the OS and the database, the extra overhead of moving the OS state gets increasingly amortized as the database size grows. VM mechanisms may also be used to replicate tenants onto different servers. In this case, a snapshot of the virtual machine disk is taken and copied over to the new server, which is started as a new VM instance containing a replica of the tenant Scale out via replication is best suited for large database tenants where the capacity required to service the tenant workload exceeds that of a single physical server.

Despite the simplicity and application-agnostic nature of VM-level elasticity, the VM-based approach is not without drawbacks. First, since VM migration and replication are black-box techniques, we cannot exploit useful application-specific properties, such as the presence of a database query log (which allows for *query shipping* instead of only *data shipping* as done in VM migration). Second, since most VM migration designs assume a shared, network-attached storage system, the target scenario is shipping *memory* state rather than disk state as in a database. Recent versions of Xen support migration of both VM memory and disk state in shared-nothing servers, but this still assumes a LAN environment. WAN-oriented extensions to VM migration have been proposed [29] but are not yet natively supported by off-the-shelf VM platforms.

	Dedicated Tenant (large)	**Colo Tenant** (small)
LAN scale-up	VM migrate	DB migrate
LAN scale-out	DB or VM replicate	
WAN scale-up	DB migrate	DB migrate
WAN scale-out	DB replicate	

Table 1: Elasticity mechanisms in ShuttleDB are intelligently chosen on tenant size and network characteristics.

Database-level mechanisms: A different approach is to implement elasticity at the application (i.e., database) level. Many such systems have been previously proposed [14, 18, 23], but operate largely by modifying the internals of the database itself. Such modifications complicate usage by end-users, as the operational guarantees of the database may change—e.g., the possibility of unpredictable transaction failures during migration in [14]. To be practical, database enhancement should retain transactional (ACID) semantics when implementing migration. Similarly, traditional master-slave replication allows us to implement a basic form of scale-out elasticity by adding or removing slave instances. However, many replication environments are oriented towards heavily manual configuration, and simply moving a database to a new server without downtime often involves more overhead than necessary if using replication.

Database-level elasticity has several advantages when used in cloud environments. Since replication and migration can be performed on a per-tenant basis, it is particularly useful for small tenants that share a virtual machine (as overloaded tenants can be individually migrated). For large tenants, these differences are less important and the benefits of database-level elasticity may not outweigh the downsides of added complexity. For cross-data center WAN elasticity, however, database-level mechanisms are still preferable due to the current limitations of VM-level mechanisms over WAN migration.

ShuttleDB approach. From the previous discussion, it follows that neither VM-level nor database-level elasticity works well in all scenarios. Consequently, ShuttleDB incorporates both VM-level and DB-level elasticity and automatically chooses the "best" elasticity mechanism for each elastic operation on a given tenant (e.g, VM-level or DB-level, and scale-out or scale-up). In addition, ShuttleDB provides its own technique for database live migration that does not require any modification to the internals of a database and works with most off-the-shelf database platforms. We also show how this approach can be used to implement database replication in multi-tenant master-slave settings.

The ShuttleDB approach is summarized in Table 1. For small, co-located tenants within a VM, scale up is the best elasticity option, since tenant requirements are less than the capacity of a single server. Further, database-level mechanisms are more efficient since they enable a single tenant to be migrated independently of others, while a VM-level mechanism would need to move the entire VM and all resident tenants. For large dedicated tenants, the choice depends on other factors. If the tenant is still smaller than a single machine, scaling up by migration to a larger machine is a feasible option. Within a LAN, VM migration is the simplest approach. When the tenant requirements exceed the capacity of a single server, it must be scaled out by replicating onto multiple machines. Within a LAN, this can be achieved by either VM- or DB-replication. Across cloud data centers, however, DB-level migration or replication is preferable to VM-level mechanisms, due to the premium placed on bandwidth usage in such scenarios.

3 Database-Aware Elasticity

Elasticity requires both migration for scaling up and replication for scaling out. Replication is actually a specific case of migration where the original database does not have to be stopped at the end of migration. Virtual machine migration is built-in to modern hypervisors, and as such is relatively simple to employ. Database-aware migration, on the other hand, is not part of off-the-shelf DBMSes. To address this issue, ShuttleDB provides its own database migration technique, which also serves as the building block for replication elasticity. Our technique draws inspiration from live VM migration [11] and database migration [6] methods, but has important differences from both. While VM migration uses "black box" data shipping to transfer state to the target machine, our DB migration technique combines query log shipping and data shipping for greater efficiency. "Process-level" techniques used in previous work such as [6] migrate *all* database state managed by a database server process, while our method can perform "tenant-level" migration where individual tenant databases within a database server can be live migrated. ShuttleDB's elasticity protocol is 'live', with minimal downtime regardless of data size, and is fully transparent to clients of the database. Our database-agnostic migration protocol is described below, while our database-specific prototype implementation is detailed in Section 5.

3.1 Migration Protocol

DB-aware migration in ShuttleDB employs a three-phase database migration protocol, which is shown in Figure 3. For a given migration, we have three logical machines to consider: the source server on which the migrating tenant currently resides, the target server to which the tenant is migrating, and the client(s) currently issuing requests to the migrating tenant.

Phase 1 – Hot Backup. In the first phase, we create a *live snapshot* of the tenant database by employing an off-the-shelf hot backup tool and streaming the resulting backup image to the target server. Suitable hot backup tools are available for most well-known database systems (MySQL, Postgres, Oracle, etc.). Since the tenant may have a large amount of data, taking this snapshot may take minutes to hours. However, as the server is not blocked during this period, it continues to service all client workloads as usual. At the completion of the first phase, the target server contains a copy of the tenant database up to some position in the binary log.

Phase 2 – Live Deltas. Once phase 1 completes, a consistent snapshot of the tenant exists on the destination server, but may be out of date, since the local server continues to execute queries during the backup. If replication has to be performed, the slave replica can be started right away at the destination, letting the database replication mechanisms bring the replica up to date.

In case of migration, the second phase proceeds as a series of *delta rounds* in which the source server replays queries from the binary log and streams them across the network to bring the target server up to date. This is analogous to memory deltas used in VM migration, but applied to a query log instead of the contents of memory.

Let p_s be the log position of the source server and p_t be the log position of the target server. At the start of each round, the target server sends p_t to the source. The source then reads from p_t to p_s in the log and sends this delta to the target, which is applied to the target database after filtering queries not pertaining to the migrating tenant. Once the delta is applied to the target, the next round begins by again sending p_t to the source.

Delta rounds continue until the duration of the most recent round is either (a) less than a small threshold (e.g., a few seconds) or (b) greater than the round before it. This approach guarantees termination of phase 2. In typical circumstances, each delta round is shorter than the last (since the number of write queries in the next delta round is proportional to the duration of the previous round), which will ultimately result in satisfying condition (a). If this is not the case – i.e., delta rounds are getting longer – then this is an indication that the target server is actually falling further behind the source and will trigger condition (b). In either case, migration proceeds to phase 3.

Phase 3 – Handover Delta. Once delta rounds are completed, the two copies of the database are nearly in sync, and we are ready to hand off the tenant workload from the source to the target. To do this, a final *handover delta* is performed to complete migration. The lo-

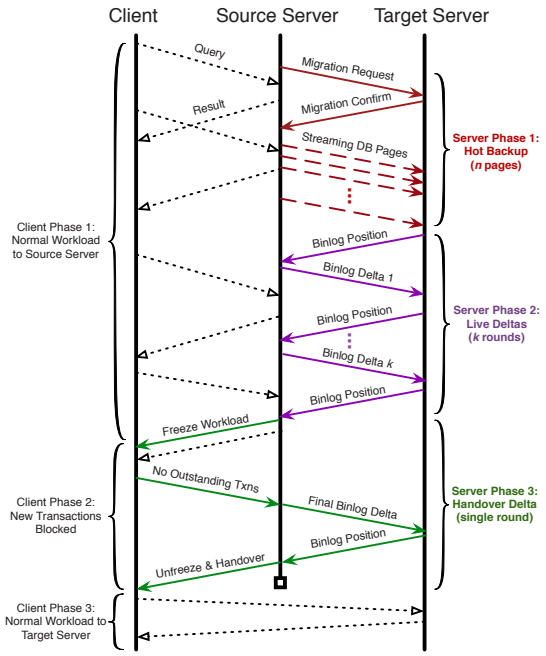


Figure 3: Three-phase live migration protocol for database tenants used in ShuttleDB.

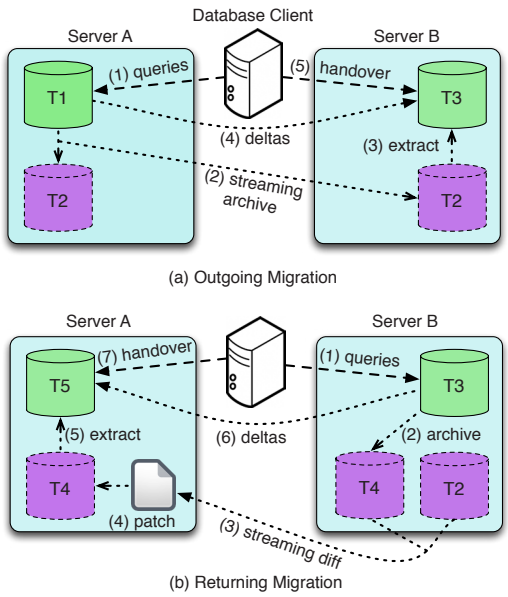


Figure 4: Saving a tenant archive during migration allows transferring small diffs instead of the full tenant database during subsequent migrations.

cal server first freezes the tenant workload by queueing all new incoming transactions directed at the migration tenant (but without blocking existing transactions), then waits to finish servicing the tenant's outstanding transactions prior to the freeze. Once all such transactions are completed, the local server sends a final delta, bringing the target server fully in sync with the source (since new transactions are frozen in the meantime). Once the final delta is applied, previously queued transactions are forwarded to the target server, clients are redirected to the new server, and the tenant workload is unfrozen. This completes the handover and switches the 'authoritative' tenant copy to the target server. Note that while phase 3 necessarily imposes a degree of effective downtime while the final delta is applied, this duration is *not* dependent on the size of the database and is typically a second or two at most, as demonstrated in Section 6. Downtime may be longer if phase 2 was terminated by lengthening delta rounds, but this case is unlikely except with extremely write-intensive workloads.

To adjust this migration protocol for replication, we do not need to freeze the workload at the master. Instead, the slave connects to the master and receives a continuous stream of updates from the master. Queries can be directed to the new slave as soon as it has caught up.

3.2 Performance Optimizations

We employ two notable performance optimizations during the migration process. First, we consider the fact that since migration itself imposes some degree of overhead (e.g., streaming the migrating tenant), local performance may be degraded. To address this issue, we employee an automated technique previously proposed [6] that automatically and dynamically rate-limits migration (depending on performance) to avoid excessive interference. Second, we consider the case when a migration is performed to a server containing an old version of the tenant (e.g., when migrating to a server containing a backup or returning from a cloud server following a temporary workload spike). Here, we do not need to migrate the *entire* tenant, but can transmit only the (small) delta between the old tenant copy and the up-to-date version.

This process, which we term 'delta migration', is illustrated in Figures 4(a) and (b) for an outgoing and returning migration, respectively. During the initial hot snapshot (phase 1) of the outgoing migration, we save a copy of the snapshot on both the source server (a local copy) in addition to the usual streaming copy to the target server. The remainder of phases 2 and 3 then proceed as normal. During phase 1 of the return migration, however, rather than streaming the database back to the source server, we re-use the older snapshot already present locally. We then generate a patch from the original database snapshot and stream only this patch back to the source. Since the source still has a copy of the original snapshot, it then applies the patch to generate the up-to-date snapshot, then proceeds phases 2 and 3 of bursting as usual.

In performing this optimization, we are able to skip the most expensive part of migration (the full data stream in phase 1). The net result is a major reduction in the

amount of network data that must be transferred to migrate between machines. This reduction in network traffic is of particular interest when migrating over the wide-area, e.g., between a local cluster and Amazon EC2, since such transfers are likely to be over relatively low-bandwidth links.

Note that pre-copying can also be employed to reactively spawn slave replicas in case of a sudden load spike. Previously terminated replicas can be quickly restarted by sending the small diff patch containing the updates that occurred since they were stopped.

4 Automated Elasticity

ShuttleDB employs an intelligent algorithm that combines VM-level elasticity with the database-level techniques described in the previous section to automatically scale the capacity of each tenant as needed by the workload. The algorithm involves four key steps: (i) *when* to invoke the scaling algorithm, (ii) *who* (i.e., which tenant(s)) to choose for optimization, (iii) *where* to migrate or replicate the tenant(s), and (iv) *which* mechanisms to use for scaling: i.e., scale-up, scale-out or scale-back and whether to use VM-level or DB-level techniques. The algorithm involves the following steps:

Step 1: **When** *to initiate elastic scaling:* ShuttleDB monitors the current query latency of tenants in the database server (computed as a smoothed average over a sliding window) and also tracks the resource usages at the underlying virtual machine to determine when to initiate the scaling algorithm. ShuttleDB uses an upper threshold on latency and resource utilization as well as a lower threshold on these values to initiate scale-up/out and scale-back, respectively. Further, in addition to *reactively* triggering the algorithm when the thresholds are breached, it is also possible to use time series based load forecasting that uses past trend history to predict future values and use these predictions to *proactively* initiate the algorithm. Currently, we use a standard ARIMA time-series forecasting method, which smooths the observed latencies and utilization over recent time periods to predict future values [28].

Step 2: **Which** *tenants to choose for scaling:* In most cases, the tenant that is experiencing the overload (or is about to, as per predictions) is chosen for scaling (or for scaling back if it is under-loaded and below the low threshold). However the choice of which tenants to choose may not always be staightforwad. In certain shared co-location scenarios, for example, no single tenant may be experiencing an overload but each tenants may be experiencing small increases in load so that they all collectively exceed the higher threshold. A more interesting scenario is one where one tenant is overloaded, but it may be *cheaper to move out a different tenant* and give the freed resources to the overloaded tenant. For example, if two tenants equally share a VM's CPU and memory and have database sizes of 5GB and 10GB, and if latter experiences overload, it is cheaper to move the first tenant and give all of the VM's resources to the latter. To intelligently choose the "correct" tenants, ShuttleDB performs a simple cost-benefit analysis. The *cost* of moving a tenant is estimated as the amount of disk state (and possibly memory state when using VM-level migrations) that must be transferred. The *benefit* of moving a tenant is the amount of load it offloads to a different server (measured as CPU or disk load, depending on the bottleneck resource on that VM). ShuttleDB greedily chooses the tenants with the greatest benefit to cost ratio – that is, those that offload the most load at the lowest data transfer cost.

Step 3: **Where** *to move a tenant:* Whenever possible ShuttleDB attempts to move tenants to servers in the same cloud data center. In scenarios where local resources are stressed, ShuttleDB will then choose to move tenants to servers in the nearest cloud site. Note that such WAN-level migrations or replication must be done carefully since there will be an impact on the front-end tiers of the application, which may experience WAN latencies if the backend tenant moves to a different site. Typically such moves would be done in consultation with the elasticity mechanisms of the front-tier as well, but such coordinated elasticity mechanisms are beyond the scope of this paper, and here we assume that the scaling of database cloud is done independently.[1] Typically for scale out and scale up, ShuttleDB chooses any server with sufficient idle resources. Scale back for shared tenants involves a consolidation and ShuttleDB chooses a virtual machine that hosts other shared tenants but has sufficient resources to house more.

Step 4: **Which** *mechanisms to choose:* The final step involves determining which elasticity mechanism to choose. Table 1 depicts the preferred DB or VM-level mechanism used in each case. For shared (small) tenants, scaling up is the preferred option and DB-level mechanisms are used to move only the desired tenants and avoid needless data copying. The only scenario where scale out is used for a small tenant is when it experiences a very large workload growth - in this case, a DB migration is first performed to extract the tenant out of the shared VM and it is given its own VM, which is then replicated, like in the large dedicated tenant case. For large dedicated tenants, ShuttleDB attempts to scale up (migrate to bigger server) when possible and then uses scale out (replicate to other servers) when no single server can service the incoming workload. LAN mechanisms are always preferred over WAN.

[1] Amazon's S3 storage also performs such geographic replication independently, although latency issues are more critical for databases.

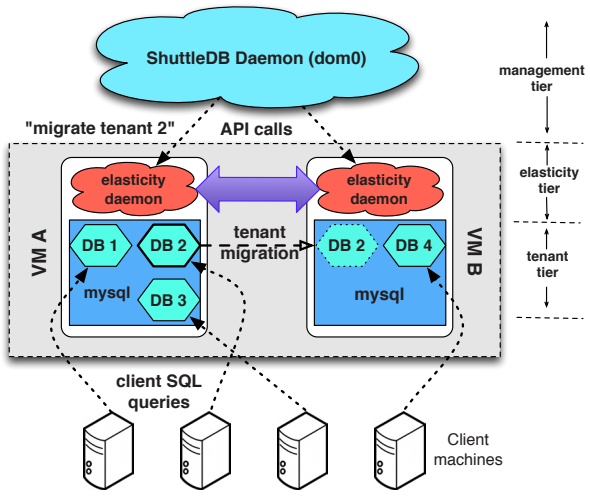


Figure 5: ShuttleDB consists of three tiers: a database/tenant tier (using an off-the-shelf DBMS), a database elasticity tier (using VM or DB-aware migration), and a top-level manager (e.g., to facilitate elasticity).

5 Prototype Implementation

Our current implementation of ShuttleDB uses a three-tier design as depicted in Figure 5. End-users (or application servers) talking to the databases interact with the lowest level of the system, which are standard database processes running within VMs (e.g., `mysqld`). Above the database servers is the elasticity layer, which provides the automated mechanisms to dynamically grow, shrink, or relocate the resources allocated to any given database tenant, using either VM-level or DB-level elasticity. At the highest level is the elastic database manager, which employs the simple API exposed by the elasticity layer to manage server and tenant workloads. The manager may implement simple or sophisticated algorithms for automatically administrating a database server cluster. While many database managers are possible, we present an example in Section 6 for automating the process of 'cloud bursting', in which tenants are migrated to remote servers to alleviate workload spikes.

Each physical server in our prototype runs a single instance of the ShuttleDB daemon on the domain0 VM, which runs Xen Cloud Platform 1.6 on top of CentOS 6 to manage all VMs on the server. Multiple daemons communicate in a peer-to-peer fashion to facilitate migrations. Separate daemons runs within each domainU VM to manage the database-aware elasticity layer.

VM elasticity. To allow for shared-nothing live migration of virtual machines, we make use of the Storage XenMotion feature, which allows for moving disk, memory, and virtual devices to a remote host. To clone an active VM, we snapshot the VM (a live operation), then export the snapshot to a new VM on the local or remote host. We then start the cloned VM and update its network settings to result in a suitable VM for replication.

DB elasticity. We implemented our technique for database-aware migration on top of MySQL, using the Percona XtraDB [25] database engine (effectively InnoDB with a few useful extensions pertaining to taking backups) provided by the Percona Server package. The database elasticity implementation is loosely coupled from the database engine itself, as follows. For migration phase 1 (streaming backups), we use a hot backup tool (Percona `xtrabackup` [30] in the current prototype) to snapshot and extract the data for the migrating tenant, which is compressed and streamed across the network to the target server. On completion, the target server imports the tenant database into the already running database server (this functionality is provided both by XtraDB and bleeding-edge versions of InnoDB [5]). We perform phase 2 (live deltas) by reading from the database transaction log and streaming updates to the target server. The queries executed since the initial snapshot or last delta round are filtered to remove extraneous queries pertaining to non-migrating tenants, then shipped to the target server and executed. For simplicity, once the handover of phase 3 begins, we employ a client-side proxy to freeze the workload and temporarily queue transactions, then release those transactions to the target server once the handover completes. This approach avoids having to make any modifications to the database engine itself.

Delta migrations. Our implementation handles the delta migration optimization described in Section 3.2 at a disk block level. During the initial outgoing migration, the local copy of the tenant is saved as a single binary archive file, which is also transferred to the target server. When the return migration is initiated, the database patch is simply generated as a disk-level patch between the original archive file and the up-to-date archive file using `rdiff`. Since this approach is oblivious to any actual data formats used by the database, it relies on block-level similarities between the archives to generate a compact diff – however, our experiments in Section 6 demonstrate that this is generally sufficient and results in a small patch file. One issue we encountered during testing was workload interference resulting from the archiving process, since saving a local copy of the archive required substantial I/O resources. To counteract this, we added the option to dedicate either a spare disk or a RAM-based volume for archiving operations. Local archiving may also be disabled entirely, which removes much of the I/O overhead of migration but prevents use of the delta migration optimization – effectively trading off between disk and network resource consumption.

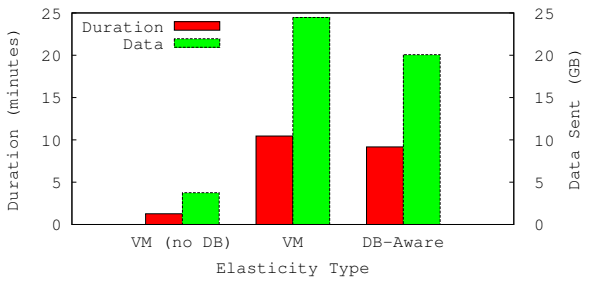


Figure 6: LAN migration of a large (dedicated) tenant.

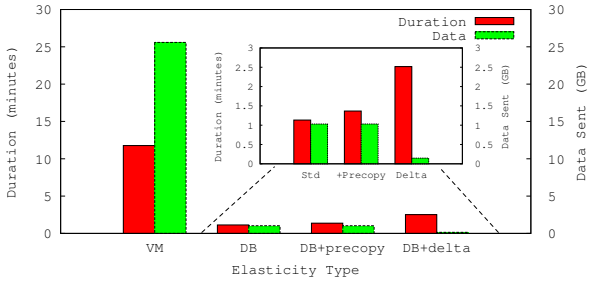


Figure 7: LAN migration of a small (collocated) tenant.

6 Experimental Evaluation

To evaluate our prototype, we first consider the individual elasticity mechanisms employed by ShuttleDB (from Table 1), then consider two scenarios demonstrating ShuttleDB's utility – first, in automating database cloud bursting, and second, in leveraging DB-aware elasticity to increase the efficiency of VM replication.

We use a heavily modified version of the Yahoo Cloud Serving Benchmark (YCSB) [12] to generate the workload for our system. While YCSB was originally designed exclusively for key-value stores, we begin with an extended, transactional version used in prior work [17, 14, 6] and further extend it to generate a closed workload with Poisson-distributed arrival times. Each workload consists of replaying a trace of workload 'intensities', which determine the number of transactions issued to the tenant per time unit.

6.1 LAN Elasticity

We first compare the efficiency of VM and DB-aware migration when operating over a LAN, in order to substantiate our earlier arguments about when each technique is preferable. We configure a VM with 30 GB of storage and 2 GB of RAM. The size of the base system is roughly 1.6 GB, while all additional space is used by the database server. We first consider moving a large (i.e., dedicated) database tenant by configuring a 20 GB tenant and moving it to a second server while servicing a workload. Figure 6 shows the duration of migration for 3 cases: a no-tenant baseline (i.e., only the OS), VM-based tenant migration, and DB-aware tenant migration. We see that the benefit of employing DB-aware elasticity in this case is minimal and likely does not justify the added migration complexity versus simply using VM elasticity.

Next, we configure the VM with twenty 1 GB tenants instead of a single 20 GB tenant, and evaluate four scenarios: VM migration, DB-aware migration, DB-aware migration with precopying (i.e., preparation for future delta migrations), and DB-aware delta migration (i.e., migrating to a server with a local precopy). In each scenario, we transfer a **single** tenant from the shared server to a separate, dedicated server. In this case, as shown in Figure 7, the differences are striking. VM migration, which simply transfers the entire VM, results in nearly 20x slower migration than DB-aware migration. Adding precopying to DB-aware migration adds a small amount of overhead, but still greatly outperforms VM elasticity. Finally, using a previous precopy to perform a delta migration takes somewhat longer on account of processing the database delta, but results in transferring less than 200 MB total of data – roughly one fifth of the already reduced amount in the base DB-aware case.

Result: *With large tenants, the simplicity of VM migration is preferable. With small tenants, however, DB-aware elasticity greatly outperforms VM elasticity. The use of precopying can even further reduce the amount of network data required.*

6.2 DB-Aware Live Migration

Next, we evaluate the 'liveness' of our DB-aware migration technique by considering the amount of downtime incurred. We configured two ShuttleDB servers over a LAN, and a single tenant with 1.5 GB of data servicing 80 read and 20 write queries per second. We then moved the tenant from the source server to the target using DB-aware elasticity, observing the transaction latencies shown in Figure 8. Migration begins at event (a), at which point ShuttleDB begins streaming the tenant to the target server (phase 1). As seen, this operation has little to no visible impact on tenant performance. Less than 3 minutes later, at event (b), the initial streaming copy is completed, and the target server begins preparing the copy to service the workload. Once complete, two copies of the tenant are running, and the original copy begins applying deltas to the new copy (phase 2). Delta rounds take roughly 15 seconds, at which point the workload is frozen and the handover (phase 3) is performed.

The inset of Figure 8 shows a plot of transaction latencies over time around the time of the handover. Latency just following the handover spikes, owing to the frozen workload (during which transactions are queued

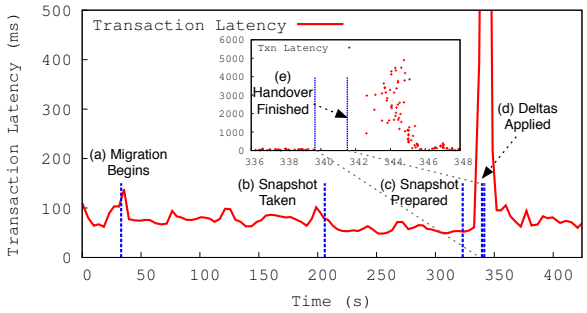


Figure 8: Live migration of a tenant servicing 10 transactions (100 queries) per second.

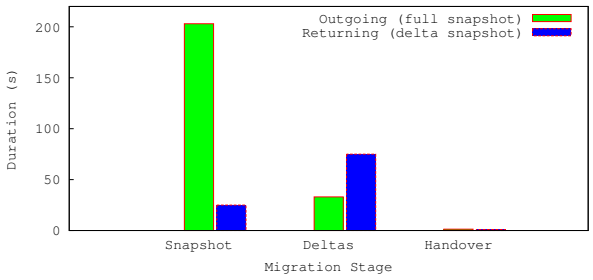


Figure 9: DB-aware delta migrations significantly reduce bandwidth and time requirements across wide-area networks.

at the source). However, this period lasts only 2 seconds, after which queued transactions are released to the target. Here, the queued transactions are delayed by an average of 2-3 seconds, but only 25 transactions in total are affected, and the entire duration of possible delays lasts less than 5 seconds. Finally, we note that this result is conservative, since the migrating tenant is servicing many queries during the handover; less active tenants will experience a shorter handover period, and thus will observe lesser delays. As seen, the only notable workload impact occurs during this handover period, whose duration is dependent *only* on the write workload intensity, and is not related to data size. Migrating a larger tenant simply extends the harmless duration of phase 1.

Result: *DB-aware elasticity in ShuttleDB provides robust migration capabilities with near-zero downtime and minimal query delays, even for highly active tenants.*

6.3 Wide-Area Elasticity

Elasticity between data centers (i.e., over a WAN) is challenging due to lower bandwidth and higher latencies. To evaluate this scenario, we configured a source server on the west coast of the US with ten 512 MB database tenants handling 50 queries per second, and a target server in an Amazon EC2 data center located on the east coast of the US. As shown in Figure 9, we then shifted one of the ten tenants to the EC2 server using DB-aware elasticity, then back from the EC2 server using a delta migration after waiting several minutes. As shown, nearly 90% of the elapsed time of the initial migration is spent transferring the initial snapshot (unsurprising given the limited available bandwidth). Importantly, the entire period in which the workload is **not** serviced (during the handover phase) lasts only a single second.

On the returning migration, we again observe the effectiveness of DB-aware delta migrations. While processing deltas increases the time spent in phase 2, the decrease in phase 1 more than compensates, and the total migration duration on the return is less than half that of the outgoing migration. Furthermore, the amount of data transferred during the delta migration is only 97 MB, as compared to 776 MB during the outgoing migration – over an 87% reduction.

We could not easily perform a companion experiment using VM elasticity, as public data centers such as EC2 do not expose their hypervisor infrastructure. Despite this, however, we would not expect VM elasticity to perform well given the limitations demonstrated in the previous experiment. Moreover, over a lower-bandwidth network such as a WAN, we would expect the differences to be even more significant than before.

Result: *ShuttleDB elasticity can effectively span multiple networks and data centers and minimizes the data transfers necessary.*

6.4 Defusing a Hotspot

Here, we present end-to-end experiments demonstrating how ShuttleDB responds to a server hotspot by migrating and replicating tenants. We use the World Cup soccer trace [3] to generate the workload for our experiments. This trace contains an end-to-end workload hotspot, starting with a stable (low) arrival rate, rising to a peak, then falling back to the baseline.

We configured our local server with 10 tenants, each with 1 GB of data. The query arrival rate of a tenant is driven by the world cup trace, while the other tenants run a standard arrival rate – initially, all arrival rates are identical. We set the bursting threshold latencies to 300 ms (upper) and 200 ms (lower). The multi-query transactions executed by all tenants include a mix of read, write, sort, and join operations, with specific arrival times given according to a Poisson distribution. Finally, the duration of the world cup trace is scaled to 90 minutes.

First, we run the workloads with ShuttleDB disabled, to observe the effects of the hotspot on tenant performance. We then re-run the same experiment with tenant migration on LAN and WAN. All the results are summarized in Figure 10. The increase in workload around $t = 50$, results in a sharp latency increase that exceeds

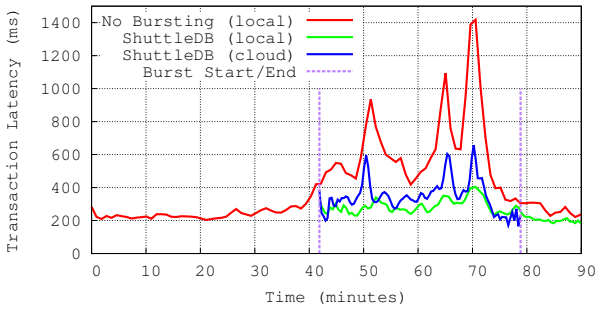


Figure 10: ShuttleDB automatically migrates tenants to mitigate the impact of a workload spike.

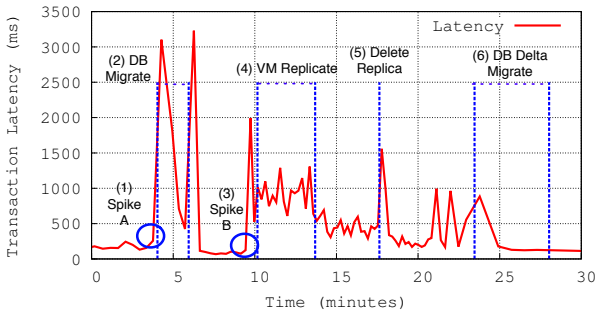


Figure 11: Database-aware elasticity with two workload spikes. First spike handled with tenant migration, and second spike with replication before scaling back.

4x baseline performance without ShuttleDB. When running with ShuttleDB, however, a migration is initiated around $t = 70$, once the ARIMA prediction determines that latency will exceed the threshold value of 300 ms. Whether the tenant is migrated to another database on the same LAN or over WAN, the latency remains within reasonable bounds with an average latency increase of 100ms for the LAN case. Once the peak has passed and latency begins to fall to normal levels (around $t = 150$), the tenant is returned to the local server.

In a second experiment, shown in Figure 11, we put the tenant under a series of two workload spikes, exceeding the capacity of the multitenant server. Following the spikes, the workload on the tenant gradually reduces to normal. To address the workload spikes, we provisioned two spare servers for ShuttleDB to use.

After the first spike, ShuttleDB migrates the single tenant out of the multitenant server to the first dedicated server, stabilizing performance. Note that although latency spikes briefly immediately following this migration due to a cold cache, this issue may be mitigated by executing read queries on both machines to warm the target cache prior to the handover. After the second spike, latency increases yet again, and so ShuttleDB replicates the entire VM to the second spare server. However, note that the primary reason we can effectively employ VM replication is *because* of the migration, which extracts the single tenant. Following the spikes and gradual decrease of the workload, the database is able to scale back, first by deleting the VM replica, then by performing a DB-aware delta migration back to the original server.

Result: *Migration and replication can be combined in ShuttleDB to maximize elasticity in multitenant servers.*

7 Related Work

Elastic cloud platforms have been proposed for many useful applications, such as video streaming services [32] and medical image registration [21]. The general concept of 'cloud bursting' [7, 22] has become popular as a way to merge existing infrastructure with newly available cloud resources. Systems such as Dolly [9] have considered cloud systems for databases through the use of cost models governing database provisioning.

Live migration has been extensively studied in the context of virtual machines [11, 8, 29], where the key challenge is migrating a dynamic memory image with minimal downtime. Live migration has been extended to the domain of databases in the context of both shared-nothing systems [17, 18] and systems with networked attached storage [14]. Our own prior work has addressed performance interference when migrating databases [6].

Multitenant databases have also attracted significant attention due to the rise of cloud computing, at varying levels of multitenancy [17, 27, 31]. Prior work has demonstrated that purely VM-based multitenancy may result in high overhead and low tenant consolidation [13], a conclusion supported by our own studies.

8 Conclusions

In this paper, we presented techniques to implement database-aware elasticity in multi-tenant database clouds. We proposed a database-aware live migration and replication approach that is designed to work with common off-the-shelf databases without requiring any database engine modifications. ShuttleDB combines database-aware techniques with VM-level mechanisms to implement a flexible approach to achieving efficient scale up, scale out or scale back for diverse scenarios ranging from different tenants sizes to inter- and intra-data center elasticity. We implemented a prototype of ShuttleDB and experimentally demonstrated the benefits of ShuttleDB's database-aware elasticity mechanisms and intelligent elasticity algorithm. As future work, we plan to study the interplay between ShuttleDB's decision making and the elasticity mechanisms employed by other application tiers.

References

[1] AMAZON. Amazon relational database service. http://aws.amazon.com/rds/, 2013.

[2] AMAZON. Ec2 auto scaling. http://aws.amazon.com/autoscaling/, 2013.

[3] ARLITT, M., AND JIN, T. A workload characterization study of the 1998 world cup web site. *Network, IEEE 14*, 3 (2000), 30–37.

[4] ARMBRUST, M., FOX, A., GRIFFITH, R., JOSEPH, A. D., KATZ, R. H., KONWINSKI, A., LEE, G., PATTERSON, D. A., RABKIN, A., STOICA, I., AND ZAHARIA, M. A view of cloud computing. *Commun. ACM 53*, 4 (2010), 50–58.

[5] BAINS, S. Innodb transportable tablespaces. http://blogs.innodb.com/wp/2012/04/innodb-transportable-tablespaces/, 2012.

[6] BARKER, S., CHI, Y., MOON, H. J., HACIGÜMÜŞ, H., AND SHENOY, P. 'cut me some slack': latency-aware live migration for databases. In *EDBT* (2012).

[7] BARR, J. Cloudbursting—hybrid application hosting. http://aws.typepad.com/aws/2008/08/cloudbursting-.html, 2008.

[8] BRADFORD, R., KOTSOVINOS, E., FELDMANN, A., AND SCHIÖBERG, H. Live wide-area migration of virtual machines including local persistent state. In *VEE* (2007).

[9] CECCHET, E., SINGH, R., SHARMA, U., AND SHENOY, P. J. Dolly: virtualization-driven database provisioning for the cloud. In *VEE* (2011).

[10] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: a distributed storage system for structured data. In *OSDI* (2006).

[11] CLARK, C., FRASER, K., HAND, S., HANSEN, J. G., JUL, E., LIMPACH, C., PRATT, I., AND WARFIELD, A. Live migration of virtual machines. In *NSDI* (2005).

[12] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with ycsb. In *SoCC* (2010).

[13] CURINO, C., JONES, E. P. C., MADDEN, S., AND BALAKRISHNAN, H. Workload-aware database monitoring and consolidation. In *SIGMOD* (2011).

[14] DAS, S., NISHIMURA, S., AGRAWAL, D., AND ABBADI, A. E. Albatross: Lightweight elasticity in shared storage databases for the cloud using live data migration. *PVLDB 4*, 8 (2011), 494–505.

[15] DATASTAX. Why migrate from mysql to cassandra. White paper, Datastax, 2012.

[16] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: amazon's highly available key-value store. In *SOSP* (2007).

[17] ELMORE, A. J., DAS, S., AGRAWAL, D., AND EL ABBADI, A. Who's driving this cloud? towards efficient migration for elastic and autonomic multitenant databases. Tech. Rep. CS-2010-05, UCSB, 2010.

[18] ELMORE, A. J., DAS, S., AGRAWAL, D., AND EL ABBADI, A. Zephyr: live migration in shared nothing databases for elastic cloud platforms. In *SIGMOD* (2011).

[19] GOOGLE. Google cloud sql. https://developers.google.com/cloud-sql/, 2013.

[20] HOGAN, M. Cloud elasticity and databases. http://scaledb.blogspot.com/2011/08/cloud-elasticity-databases.html, 2011.

[21] KIM, H., PARASHAR, M., FORAN, D. J., AND YANG, L. Investigating the use of autonomic cloudbursts for high-throughput medical image registration. In *GRID* (2009).

[22] MELL, P., AND GRANCE, T. The NIST definition of cloud computing, September 2011.

[23] MINHAS, U. F., RAJAGOPALAN, S., CULLY, B., ABOULNAGA, A., SALEM, K., AND WARFIELD, A. Remusdb: transparent high availability for database systems. *VLDB J. 22*, 1 (2013), 29–45.

[24] OSMAN, S., SUBHRAVETI, D., SU, G., AND NIEH, J. The design and implementation of zap: a system for migrating computing environments. *SIGOPS Oper. Syst. Rev.* (Dec. 2002).

[25] PERCONA. The percona xtradb storage engine. http://www.percona.com/docs/wiki/Percona-XtraDB:start, 2013.

[26] SALESFORCE. Salesforce crm and cloud computing. http://salesforce.com, 2013.

[27] SOROR, A. A., MINHAS, U. F., ABOULNAGA, A., SALEM, K., KOKOSIELIS, P., AND KAMATH, S. Automatic virtual machine configuration for database workloads. *ACM Trans. Database Syst. 35*, 1 (Feb. 2010), 7:1–7:47.

[28] WEI, W. W. S. *Time series analysis - univariate and multivariate methods*. Addison-Wesley, 1989.

[29] WOOD, T., RAMAKRISHNAN, K. K., SHENOY, P., AND VAN DER MERWE, J. Cloudnet: dynamic pooling of cloud resources by live wan migration of virtual machines. *SIGPLAN Not.* (2011).

[30] Percona XtraBackup. http://www.percona.com/software/percona-xtrabackup/, 2013.

[31] XIONG, P., CHI, Y., ZHU, S., MOON, H. J., PU, C., AND HACIGÜMÜŞ, H. Intelligent management of virtualized resources for database systems in cloud environment. In *ICDE* (2011).

[32] ZHANG, H., JIANG, G., YOSHIHIRA, K., CHEN, H., AND SAXENA, A. Intelligent workload factoring for a hybrid cloud computing model. In *SERVICES* (2009).

Matrix: Achieving Predictable Virtual Machine Performance in the Clouds

Ron C. Chiang*, Jinho Hwang+, H. Howie Huang*, and Timothy Wood*
The George Washington University and IBM T.J. Watson Research Center+*

Abstract

The success of cloud computing builds largely upon on-demand supply of virtual machines (VMs) that provide the abstraction of a physical machine on shared resources. Unfortunately, despite recent advances in virtualization technology, there still exists an unpredictable performance gap between the real and desired performance. The main contributing factors include contention to the shared physical resources among co-located VMs, limited control of VM allocation, as well as lack of knowledge on the performance of a specific VM out of tens of VM types offered by public cloud providers. In this work, we propose Matrix, a novel performance and resource management system that ensures the desired performance of an application achieved on a VM. To this end, Matrix utilizes machine learning methods - clustering models with probability estimates - to predict the performance of new workloads in a virtualized environment, choose a suitable VM type, and dynamically adjust the resource configuration of a virtual machine on the fly. The evaluations on a private cloud, and two public clouds (Rackspace and Amazon EC2) show that for an extensive set of cloud applications, Matrix is able to estimate application performance with average 90% accuracy. In addition, Matrix can deliver the target performance within 3% variance, and do so with the best cost-efficiency in most cases.

1 Introduction

In private and public clouds, the so-called Infrastructure as a Service (IaaS) model offers on-demand creation of virtual machines (VMs) for different users and applications, and enables dynamic management of VMs for maximizing resource utilization in the data centers. Ideally, a VM shall have three properties: 1) *efficiency*, where a significant portion of the program runs without any intervention from the hypervisor that manages the VMs; 2) *resource control* that prevents any program from gaining the full control of the system resources; and 3) *equivalence*, where any program running in a VM "performs in a manner indistinguishable" from an equivalent real machine [34]. Although virtualization technology has been improved greatly (its pervasive use in cloud computing is strong evidence), we have not yet achieved the vision of "an efficient, isolated duplicate of a real machine", that is, a VM shall be able to provide the performance close to the desired one.

Take a real world example, before buying a new tablet computer from an online retailer, one may shop a local store like BestBuy to test drive and compare various products. Nevertheless, any product that the customer eventually receives from the online retailer will be the same as what is presented locally. Unfortunately, when one purchases a VM in the cloud, little guarantee is provided to ensure an application hosted by the VM would keep the desired performance, not even mentioning to achieve the best cost-efficiency.

In this paper, we propose the concept of *Relative Performance* (RP) as the "equivalence" metric that measures the ratio between the desired performance and that of running in a VM. For a workload w, the RP can be formally defined as

$$RP_w = \frac{P_{VM}}{P_d}, \tag{1}$$

where P_{VM} is the performance of the workload w when running on a VM, and P_d is the desired performance. The performance is workload dependent and can be measured as the runtime (e.g., sequence alignment), throughput (e.g., video streaming), latency (e.g., webpage serving), etc. The RP that is equal to one means that the workload delivers the desired performance on the VM. The goal of Matrix is to deliver the desired performance while minimizing the resource cost.

In a cloud, many factors such as limited control of VM allocation and competition from co-located VMs to shared resources (e.g., CPU and I/O devices) contribute to hard-to-predict VM performance. To illustrate the problems on expected performance and operating cost, we run three benchmarks ranging from I/O intensive, memory intensive to CPU intensive workloads, both locally and on Amazon EC2. There are two local physical machines in this test: $PM1$ has a 2.93 GHz Intel Core2 Duo processor and 4 GB memory, and $PM2$ has a 3 GHz Intel Pentium4 processor with 2 GB memory. The desired performance P_{d1} and P_{d2} are the performance of running a given benchmark on $PM1$ and $PM2$ respectively. Fig. 1 shows the RPs (in runtime/latency for three benchmarks) for P_{d1} and P_{d2} on four EC2 instances[1]. Our tests show that the RP for these three benchmarks can vary dramatically from 18% of the target performance to more than three times. Clearly, it is challenging to know ahead of time for each application which VM instance provides a good tradeoff between the cost and

[1]For Amazon EC2 instances, $m1.small$ type equips with 1.7 GB memory and 1 EC2 Compute Unit priced at six cents per hour, $m1.medium$ 3.75 GB memory and 2 Compute Units at 12 cents per hour, $m1.large$ 7.5 GB memory and 4 Compute Units at 24 cents per hour, and the $t1.micro$ has the smallest amount of memory (613 MB) and CPU resource.

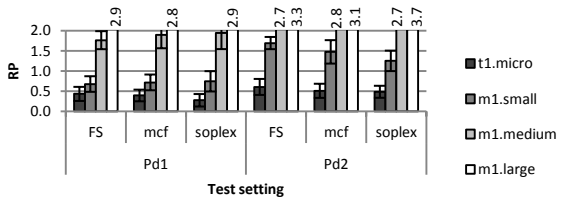


Figure 1: The performance for various EC2 instances ranges from 27% to 3.7 times of the desired performance P_{d1} and P_{d2}. Each column shows an average of ten runs

performance. Benchmarking an application in the cloud may alleviate the problem, but it becomes cumbersome as public cloud providers offer dozens of VM types.

In this work, we propose a performance and resource management system, *Matrix*, that targets at delivering predictable VM performance with the best cost-efficiency. To achieve this goal, Matrix utilizes clustering models with probability estimates to predict the performance of new workloads in a virtualized environment, chooses a suitable VM type, and dynamically adjusts the resource configuration of a VM on the fly.

The first contribution is that *Matrix can predict accurately how a new workload will perform on different cloud VM instances*. To this end, Matrix first constructs performance models of a set of representative workloads that define common application "genes". Given performance models for these applications, we leverage the support vector clustering (SVC) to quickly classify a new workload, using soft boundary probability estimates to infer its "gene" composition. A number of studies [28, 46, 22, 45] have worked on service-level agreement (SLA), performance prediction, and anomaly detection in virtualized environments. The major differences of Matrix lie in the understanding of the dynamic relationship between resource allocation and the new workload performance.

The second contribution is that *Matrix allocates VM resource to application in a way that minimizes the cost while achieving good performance*. To this end, Matrix applies an approximate optimization algorithm and makes use of the characteristics of the kernel functions of support vector machine (SVM) to find the optimized resource allocation. More specifically, the support vector regression (SVR) is used to develop our RP models. By exploiting gene composition knowledge, Matrix can do so without knowing a priori application information within guest VMs.

Third, *Matrix is able to handle different cloud environments and applications.* We conduct a large set of experiments with real cloud applications and ranging from a single machine, a local cluster, and a virtual cluster, to evaluate Matrix on both our private cloud and the public cloud of Amazon EC2 and Rackspace.

In this work, we present three use cases of Matrix:

- **Automatic VM configuration**. Matrix can adapt VM settings to the changes in workload, while maintaining a desired performance and achieving good cost-efficiency in the cloud.
- **VM instance recommendation**. With workload performance models, Matrix recommends the VM instance that is best suited for specific applications.
- **Cloud provider recommendation**. Given a new application, Matrix can also help users to choose an appropriate VM from different cloud providers.

2 Related Work

Performance Modeling and Analysis has been extensively studied, both in non-virtualized environments [29, 44], and virtualized environments [16, 36, 21, 54, 7]. There are also performance models which target specific applications or system components. For example, Li et. al [25] model the performance of parallel matrix multiplication in virtualized environments, and Watson et al. build probability distribution models of response time and CPU allocations in virtualized environments [50]. While we share the same idea on exploiting machine learning techniques, we further explore the ability of classification with probability estimates to model the performance of new workloads.

Automatic Resource Configuration is an important issue in parallel and distributed systems [24, 38, 15] and performance monitoring tools [23]. Similarly, various machine learning techniques have shown promising results for VM provision and configuration, e.g., clustering [35], classification [26], reinforcement learning [37]. Also, several works have focused on minimizing operation cost, for example, Niehörster et al. [31] applies fuzzy control at runtime, and Kingfisher [41] formulates the problem as an integer linear program (ILP) and implements a heuristic ILP solver. Most related to our work are several existing resource configuration frameworks such as DejaVu [48], JustRunIt [55], and [43]. The key differences of Matrix lie in a comprehensive framework to predict and maintain the desired performance of a new workload while minimizing the operating cost. While DejaVu also handles new applications and adapts resources to suit new demands, DejaVu uses dedicated sandbox machines to clone and profile VMs. In contrast, Matrix utilizes representative models to construct new workload's model in an online fashion. Many works aim to predict resource requirements for cloud applications, e.g., [13, 14, 18, 49, 51]. Most of them either focus on single application or ignore the cost-efficiency. On the other hand, Matrix is able to adapt to new applications and minimize the operating cost. Also, Matrix deals with the problem of multi-cloud resource management, which is shown to be critical in [5]. Performance interference in virtualized environments is another critical barrier to

provide predictable performance. DeepDive [32] utilizes mathematical models and clustering techniques to detect interference. DeepDive requires comparing the performance from VM clones in dedicated machines. Similar to [7, 19, 30], Matrix removes this need by including the interference factors into the performance models.

3 Matrix Architecture

The goal of Matrix is to predict and configure VMs in an automatic manner so that the applications running within the VMs would achieve the performance with a close vicinity of a specific one. We present the architecture of Matrix in Fig. 2. On the left, Matrix builds both clustering and RP models of representative workloads and this task is done offline. There are three steps in this phase: 1) profiling the training set of representative workloads (presented in Sec. 3.1); 2) tuning the SVM parameters to find the best model configuration; and 3) training the classifier and the basic RP models, for later use of the online module (Sec. 3.2 and 3.3). This offline training stage builds RP models from our generic benchmarks, but it can be repeated periodically to include data from newly added workloads.

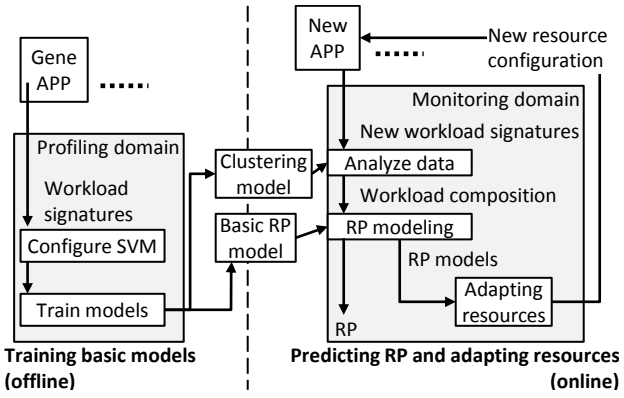


Figure 2: Matrix Architecture

When a new application is moved to the cloud, Matrix requires only the workload signature when running on its current infrastructure, which could be either physical or virtual machines. As shown in the right hand side of Fig. 2, Matrix can classify these workload signatures compared to the previously trained models. Then, the system calculates a runtime RP model based on adjusted performance estimates and outputs the predicted RP to the resource allocation module. Next, Matrix will search for the VM configurations with the minimum cost to maintain a desired performance (Sec. 3.4). To provide automatic resource management, we formulate an optimization problem with nonlinear inequality constraints. For fast response time, Matrix utilizes the Lagrange multipliers to provide an approximate solution and a bound to the minimum resource cost.

3.1 Workload Signatures

Matrix first must profile a set of workload "genes" that indicate how different types of applications will perform when moved into cloud platforms.

A group of representative applications are firstly selected as "genes" to construct an expert system. Our selection principle, similar to [2], is to have the reference workloads as diverse as possible - the resulting collection shall cover from CPU-intensive to data-intensive, and their problem sizes also shall vary from small to large data volumes. Table 1 summarizes the representative applications selected from a few widely used benchmark suites, e.g., FileBench [27], SysBench [20], SPEC2006 [10], PARSEC[1], and Cloud9 [4, 8]. Note that while this set of applications is not optimal by all means, they provide, as we will see in evaluations, a good basis for RP modeling. We leave the exploration of different gene applications as future work.

Table 1: Summary of representative applications

Name	Description
video server	serving a set of video files
web server	retrieving web contents and updating log files
file server	a mixture of various file I/O operations
OLTP	query and update database tables
mcf	running simplex algorithm
hmmer	pattern searching of gene database
soplex	linear program solver
canneal	evolutionary algorithm
DS01 to DS15	15 distributed data serving workloads
C01 to C15	15 parallel CPU-intensive workloads

For parallel application, we select a training set that consists of 15 data-intensive workloads (DS01 to DS15) and 15 CPU-intensive workloads (C01 to C15). The first five DS series workloads run Apache Cassandra, a distributed key-value store, with read/write ratios of 100/0, 75/25, 50/50, 25/75, and 0/100 where the record popularity is in uniform distribution. For DS6 to DS10, they access Cassandra with same read/write ratios but in the Zipfian distribution of record popularity. For the number 11 to 15 training workloads, they share the same pattern and order of read/write ratios in both the first and the second five groups, but the record popularity is in the latest distribution. The last 15 representative applications in the training set are CPU-intensive parallel workloads from Cloud9, a scalable parallel software testing service. The training set for CPU-intensive parallel workloads are randomly selected out of 98 different utility traces from the GNU CoreUtils 6.10 for running Cloud9.

For a basic signature, we take the arithmetic means of three system parameters - CPU utilization, the amount of data read and written per second. Since it is insufficient to use the mean alone to represent a workload when there is a large variability in the observed data, we choose the coefficient of variation (C.O.V) as part of

the signatures to describe the variability. As prior work [17, 2] has already shown that the resource allocation of VMs greatly affects the observed system parameters, we include the number of VCPUs and the size of memory in the workload signatures because these two parameters are frequently used knobs for tuning VM performance. Furthermore, we also take into account the interference from co-located VMs. For simplicity, all workload signatures from other VMs are summed up as one background VM and included in the modeling process.

Dealing with applications running on multiple machines poses more challenges. The traffic in and out of each node is critical to data-intensive applications' performance. The number of nodes is also important for modeling workload concurrency. In other words, Matrix needs to scale resources horizontally (increasing and decreasing the number of nodes), as well as vertically (scaling up and down resources on each node). Thus, Matrix includes the amount of data flow of each node and the number of nodes in a cluster as additional parameters when modeling an application performance on a set of machines.

3.2 Clustering Method

Matrix needs a workload classifier to identify new workloads that are running in the guest VMs. Most of previous works use a "hard" classifier. That is, the classifier outputs a certain workload without ambiguity. That method, however, provides little help when dealing with new workload, which can be very different from any workload in the training set. To address this problem, we explore "soft" classifiers in this work, which have soft boundary and output probability estimates of being each component in the model. These probability estimates can be utilized as weights to infer the "gene" composition of new workloads. Specifically, we utilize a multiclass SVC with likelihoods provided by a pairwise coupling method [6]. We use a rigorous procedure to tune and train classifiers. Our classifiers are built as follows:

Data Scaling avoids the features in larger numeric ranges dominating those in smaller ranges. In addition, scaling data into a restricted range can avoid numerical difficulties during the kernel value calculation [6]. We scale each attribute in the range of $[0, 1]$.

Parameter Selection: Choosing the optimal parameter values is a critical step in the SVC design. The grid search method is a common practice in finding the best configuration of a SVC. That is, the parameter selection is usually done by varying parameters and comparing either estimates of generalization error or some other related performance measure [11]. When the search approaches a grid point, it calculates the value of ten-fold cross validation (CV). In order to save the searching time, the search firstly starts with a loose grid to identify regions with good CV values. Then, the search uses a finer grid to further approach the best configuration. We conduct the grid search on the following parameters: 1) SVC types: C-SVC [3] and ν-SVC [40, 39]. 2) Kernel functions: Polynomial, sigmoid, and Gaussian radial basis function (RBF). 3) Constraint violation cost C to avoid overfitting. $C \in \mathbb{R}^+$. 4) Kernel width coefficient γ, which affects the model smoothness. $\gamma \in \mathbb{R}^+$. 5) Variable ν in ν-SVC provides an upper bound on training errors $\nu \in (0, 1]$.

Training: Once the best parameter configuration is decided, the final classifier is trained by using the best configuration with the whole training data.

In terms of SVC types and kernel functions, the grid searching results suggest that ν-SVC with RBF kernel outperforms other classifiers and kernel functions. Therefore, Matrix uses ν-SVC with RBF kernel as the classifier. ν-SVC has been proved to provide an upper bound on the fraction of training errors and a lower bound of the fraction of support vectors.

3.3 Performance Modeling

The performance modeling has two main procedures: **1) Constructing the building block:** Matrix utilizes the SVR to construct the basic RP models of each training application. A popular version of SVR is ν-SVR [40, 39]. Matrix uses ν-SVR with the RBF kernel for the basic RP modeling because the grid searching results suggest it is better than others. **2) Generating the performance model:** The performance modeling of representative workloads completes one part of the story. Our goal is to capture new workloads' RP models in an online fashion.

Suppose there are n representative workloads w_i, $i \in \{1, \ldots, n\}$. The corresponding performance models are $f_i(R)$, where $r_j = \{x \in \mathbb{R} \mid 0 \leq x \leq 1\}$ and $R = \{r_1, \ldots, r_m\}$ are resource configurations and system statistics, $j \in \{1, \ldots, m\}$. Because all performance models are built by SVR, the performance models can be represented as: $f_i(R) = \underline{\theta} \cdot \phi(r_i) + \theta_0$, where $\phi(r_i)$ are kernel functions, θ_0 is the offset vector to the origin, and $\underline{\theta}$ is the separating vector.

Our classifier then analyzes a new workload w_{new} and generates an output $\{p_1, \ldots, p_n\}$, where p_i are the probability estimates of being workload w_i, $i \in \{1, \ldots, n\}$. The final performance model of workload w_{new} is

$$f_{new}(R) = \sum_{i=1}^{n} p_i \cdot f_i(R), \quad \text{where} \sum_{i=1}^{n} p_i = 1. \tag{2}$$

In other words, the likelihood p_i acts as a weight to control the fraction of f_i in the final model f_{new}.

3.4 Automatic Resource Configuration

Once we obtain a performance model of a new workload w_{new}, configuration module starts to find the minimum allocation for keeping the desired performance.

Let C_j be the cost of resource j, $j \in \{1, \ldots, m\}$. Resources are, e.g., the memory size and the number of VCPUs and VMs. r_j is the ratio of resource j on a physical server that is allocated to the VM. We formulate the resource configuration problem as an optimization one with a nonlinear equality constraint:

$$\begin{aligned} \underset{R}{\text{minimize}} \quad & F_c(R) = \sum_{j=1}^{m} C_j \times r_j \\ \text{subject to} \quad & f_{new}(R) = \sum_{i=1}^{n} p_i \cdot f_i(R) = 1, \\ & \sum_{i=1}^{n} p_i = 1, \\ & r_j = \{x \in \mathbb{R} | 0 \leq x \leq 1\}, \\ & i \in \{1, \ldots, n\}, j \in \{1, \ldots, m\} \end{aligned}$$

Because both the objective and constraint function are continuously differentiable[2], we utilize the Lagrange algorithm for solving this problem.

Note that the above problem is formulated under the assumption that r_j is the ratio of resource j on a physical server that is allocated to the VM. However, real systems usually can not partition resources at an arbitrary granularity. For example, the memory allocation for VMs is usually done in the unit of one megabyte. If a system has 2 GB memory, the finest possible R_j values will be $\{1/2000, 2/2000, \ldots, 2000/2000\}$. As a result, the system would not be able to use the optimal resource configuration R^*. Instead, the system needs to take $(\lceil r_1^* \rceil, \ldots, \lceil r_m^* \rceil)$ as the resource configuration, where the ceiling operation of r_i here is defined as taking the smallest value r_i' in the finest possible granularity, such that $r_i' \geq r_i$. Let the granularity of resource i be d_i, $i \in \{1, \ldots, m\}$. In other words, the miss allocation on resource i is at most d_i. Therefore, the upper bound on the extra resource allocation cost is $\sum_{i=1}^{m} C_i \times d_i$.

4 Implementation

We have implemented and tested Matrix on both a local private cloud and two public clouds, namely Amazon EC2 and Rackspace cloud servers. Fig. 3 summarizes the work flow of the prototype.

The preparing data block includes parsing, formatting, and scaling collected traces. The clustering model and RP models are previously built by the training set offline. The Matrix online module is controlled by a Linux bash shell script combined with a SVM module written in C and an optimization problem solver in MATLAB. The tasks of this online module are to 1) collect traces, 2) analyze workload compositions, and 3) predict current RP and suggest a configuration to obtain desired performance with less cost.

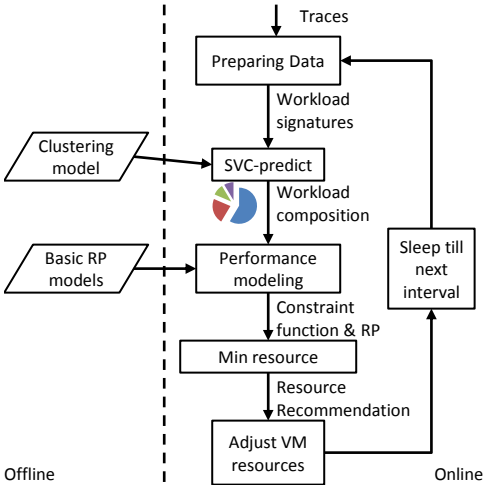


Figure 3: Matrix prototype

The online module is running as a background process in the host domain, which collects workload signatures of VMs every second by using *xentop*. At every minute, a parser will parse collected data and scale all values in the range of $[0, 1]$. The online module then feeds the scaled trace and clustering model to the *SVC-predict* module which outputs the workload composition in possibilities of representative workloads. These probability estimates along with the basic RP models become the running workload's performance model (Eq. 2). Then, Eq. 2 is served as the constraint function of the optimization problem in Sec. 3.4. Finally, the online module adjusts resource allocations and repeats the same procedure for the next interval.

There are three main differences between the two Matrix prototypes on private and public clouds. First, Matrix in public clouds can not use *xentop* to collect traces because we have no access to the host domain. Instead, we run *top* and *iostat* in every guest domain to collect traces. Second, Matrix can not arbitrarily adjust resources of an instance in the public cloud. And, instance types can only be changed when it is not running. To address this problem, we adapt Xen-blanket [53] nested virtualization for some tests.

Prototype performance. The measured running time from parsing collected trace to output the minimum resource recommendation is around 0.6 second where the optimization solver takes about 70% in the whole process. As future work, the running time of the online module can be further reduced by implementing the solver in

[2]One of the properties of the RBF kernel.

native system without using MATLAB. In addition, Matrix may also be integrated with Monalytics [23] to reduce overheads.

5 Evaluations

Testing scenarios. We evaluate Matrix in three scenarios: a single machine, a cluster of physical machines, and of VMs. Three virtualized environments are used in our experiments: local Xen virtualized servers, Amazon EC2 instances[3] and Rackspace cloud servers[4]. We label Rackspace cloud servers from smallest to the largest as RS1 to RS7. For example, RS1 has 1 VCPU and 512 MB memory and RS7 has 8 VCPUs and 30 GB memory. All tests on public clouds are conducted for at least 30 runs, with multiple batches that run at different times of day and on various weekdays and weekends.

In Sec. 5.1, we start the experiments with the single machine case, which aims to accommodate the testing applications in a VM such that the workloads perform closely to the desired one. We mainly use $PM1$, which is described in Sec. 1, as the target performance. We have two local servers for hosting VMs: $VS1$ and $VS2$ are two six-core Intel Xeon CPUs at 2.67 GHz and 2 GHz, and with 24 and 32 GB memory, respectively. Both machines are running Linux 2.6.32, Xen 4.0, and NFS over a Gigabit Ethernet.

In Sec. 5.2, Matrix aims to accommodate the testing applications in a set of VMs such that the workloads perform closely to the desired one. We use a four-node physical cluster (PC) as the target performance, each of which has a 1.80 GHz Intel Atom CPU D525 (two physical cores with hyper-threading) and four GB memory connected on a Gigabit Ethernet. In the local private cloud, we use the $VS2$ to host a virtualized cluster (VC). Similar to the single machine case in Sec. 5.1, the public VCs are hosted on the Amazon EC2 and Rackspace.

In Sec. 5.3, Matrix targets at accommodating the testing applications in a set of VMs in public clouds such that the workloads perform closely to the desired one in a local cloud. We use VCs of 32 and 64 VMs in a local cloud as the target performance, and study how to configure VCs in Amazon EC2 and Rackspace cloud servers to achieve similar performance. Each VM has one VCPU and 1.5 GB memory. This way, we examine the feasibility of migrating a VC from a private to public cloud while providing the desired performance with minimized cost.

Cloud applications that are used in this work consist of *Cloudstone*, a performance measurement framework for Web 2.0 [42]; *Wikipedia* with Database dumps from Wikimedia foundation [52] and real request traces from the Wikibench web site [47]; *Darwin*, an open source version of Apple's QuickTime video streaming server; *Cloud9* makes use of cloud resources to provide a high-quality on-demand software testing service; and *YCSB (Yahoo! Cloud Serving Benchmark)*, a performance measurement framework for cloud serving systems [9].

For YCSB, the experiments use two core workloads: YCSB1 and YCSB2, both send requests following a Zipfian distribution. The major difference between YCSB1 and YCSB2 is the read:write ratio: YCSB1 is an update heavy workload with the read:write ratio of 50:50, and YCSB2 reproduces a read mostly workload with the read:write ratio of 95:5. Note that after Sec. 5.2, YCSB1 and YCSB2 are served from multiple nodes. In addition, YCSB3, YCSB4, and YCSB5 will be added into the testing set as well. YCSB3 is a 100% read workload. 95% requests of YCSB4 are read operations and mostly work on the latest records. 95% requests of YCSB5 are also read operations but it scans within 100 records.

Evaluation metrics. We use three metrics to evaluate the performance of Matrix. To measure the accuracy of the models, we define the prediction accuracy as $1-(|predicted\ value\ -\ actual\ value|/actual\ value)$. That is, the closer to 1 the better.

The goal of Matrix is to achieve a desired VM performance with minimum cost. To this end, we define two additional metrics: the RP-Cost product (RPC) as $|RP-1|\cdot(VM\ Cost)$, and the Performance Per Cost (PPC) as $RP/VM\ Cost$. In this test, we measure the cost for purchasing instances on public clouds in dollars. For RPC, a smaller value is preferred as it indicates small performance difference and cost, and for PPC, a larger value is better because of indicating better performance for the same cost.

5.1 Single Machine Case

Model Composition. We first present how Matrix analyzes applications and composes performance models. Fig. 4 demonstrates the snapshots taken by Matrix while

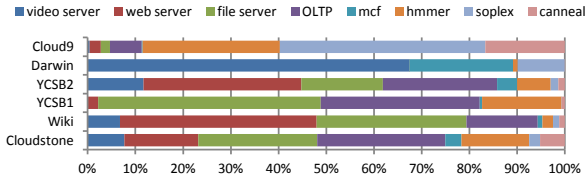


Figure 4: Application composition examples

applications are running. Let's take Darwin as an example. Darwin is about 67% like *video server*, 22% like *mcf*, 10% like *soplex*, and the possibilities to be others are very small. Although Darwin is a video streaming server, it is not 100% like the *video server* from the FileBench in the representatives. The reason is that the *video server* only emulates I/O operations and omits many CPU tasks on a video streaming server, which can be captured by

[3] A full list of Amazon EC2 instance types and prices can be found at http://www.ec2instances.info/

[4] A full list of Rackspace cloud servers can be found at http://www.rackspace.com/cloud/servers/.

Matrix with suggestion of including *mcf* and *soplex* as part of the Darwin's workload signature. Therefore, Darwin's estimated performance by the composition in Fig. 4 will be $0.67 \cdot f_{video\ server} + 0.22 \cdot f_{mcf} + 0.1 \cdot f_{soplex} + ...$ (Recall Eq. 2). Similarly, the sample composition of YCSB1 has a large portion of *file server*, *OLTP*, and *hmmer*. Note that these are just sample snapshots, and the composition ratio depends on the workload intensity and datasets, and may change over time.

Model Accuracy. We examine Matrix's accuracy on predicting new workloads' RP across different settings on our local VMs, the Amazon EC2 instances, and the Rackspace cloud servers. To train the RP models on the local VMs, we run the training set on $PM1$ and VMs for the RPs and training data. We collect 1,000 data points for each training workload's performance model. Each data point is generated by running the workload with a uniformly randomly configured thread (worker) count (2 to 32), working set size (5 to 32 GB), and resource allocation (1 to 8 VCPUs and 1 to 8 GB memory). Because hardware heterogeneity potentially affects performance of cloud applications [12, 33], Matrix also trains models for working on Amazon and Rackspace, instead of simply using those trained on the local VMs. The training process on the public clouds is almost identical to the one on local VMs, except the part of dynamically configuring resources. Because we can not arbitrarily adjust resources on the public clouds, the training data are collected from running them on each instance type for 100 times. Note that Matrix needs only a one time training process for modeling the gene workloads in the VMs.

For the tests on local VMs, we run each configuration for five times, five minutes per run. In Fig. 5, each column shows the average prediction accuracy and standard deviation of 30 runs (five runs for six testing applications). The same testing process is repeated on the Amazon and Rackspace at three different times and days. Thus, the public cloud results are averages of 90 runs.

Most of prediction accuracies are higher than 85% and the average value across all cases is 90.15%. The local VM tests on $VS2$ have a slightly higher accuracy (91.1%) than those on $VS1$ do (90%). On the Amazon EC2, t1.micro has the lowest prediction accuracy due to big variances on its performance. In general, larger instance types are more stable and usually lead to higher accuracies. The experiments on Rackspace also show that larger instances tend to have higher accuracy. Given the same instance type, HVM instances have lower accuracy than paravirtualized VMs, partly due to virtualization overheads. The average prediction accuracies across all Amazon and Rackspace instance types are 89.8% and 90.3% respectively.

All results pass the two-sample t-tests and are stable across all test environments. Note that we also conduct the same tests on the training set. The results show that the training applications can be identified correctly over 95% and their performance estimations have accuracy higher than 94% across all training applications.

Automatic Resource Configuration. Here, we assume a user wants to keep a desired performance of a YCSB VM with the minimum resources allocated. We run YCSB2 for one hour and change workload intensities every ten minutes. In the first ten minutes, two threads work on two millions records; The workload intensity is increased to four threads and eight millions records in the second period; eight threads and 16 millions records in the third period; Then, workload intensity is decreased to four threads and 16 millions records in the fourth period; two threads and 16 millions records in the fifth period; two threads and two millions records in the last ten minutes. Fig. 6a shows the corresponding resources and RPs as the workload intensity changes. Over the hour, the average resource savings are 37% on CPU and 55% on memory, when compared to a baseline VM which keeps using two VCPUs and four GB memory to imitate $PM1$'s setting. The average performance is 1.06 (closer to the target value) compared to 1.56 provided by the baseline VM.

In Amazon EC2, we can only change the type of an instance when it is not running. As a workaround, we use the Xen-blanket (nested virtualization) in an Amazon EC2 HVM instance (m3.2xlarge). In the one hour test, the average resource saving is about 5% on memory, compared to a baseline VM which keeps using one VCPUs and two GB memory. There is no resource saving numbers for CPU because the minimum VCPU number is one in this test. The average RP shown in Fig. 6b is about 0.95 compared to the one of 0.83 by the baseline VM. In other words, with the ability of adjusting resources to accommodate demands, Matrix can keep a desired performance with as few resources as possible.

Choosing instances among cloud providers. In this test, Matrix is used to recommend instances for running a certain workload as close to the desired performance as possible. The light, medium, and heavy workloads used here are defined as 4, 16, and 32 threads (or workers) with 8, 16, and 32 GB working set respectively. We conduct the same tests on Amazon EC2 and Rackspace cloud servers. Then, we list the most recommended instance types in Table 2 such that running certain workloads would be close to the desired performance with less cost. If the recommended instances on both sides have the same price, e.g., RS2 vs. m1.small, the one provides a higher RP will be selected. For the light workload intensity, RS3 is the most recommended type to use, which has the same price as m1.medium at $0.12 per hour. RS3 is chosen because it provides higher RP with the same price. The performance of YCSB work-

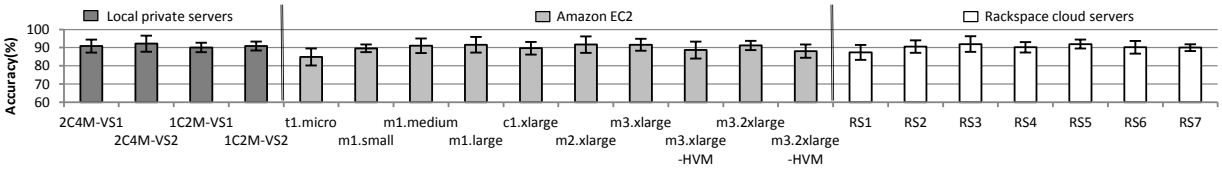
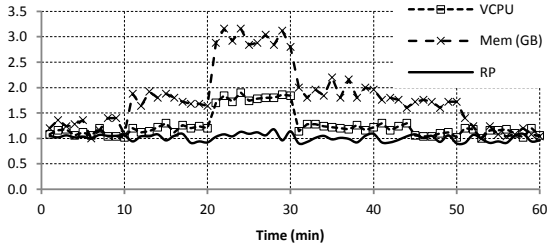
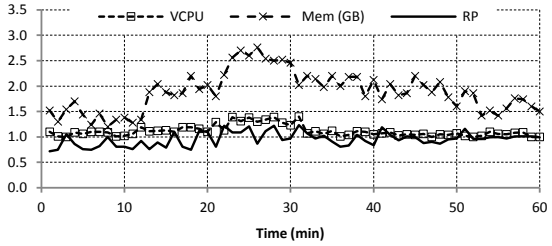


Figure 5: Accuracies on predicting performance. The labels aCbM-VSc on the leftmost four columns mean these tests are done on a VM with a VCPU and b GB memory hosted by our local machine VSc. The rightmost seven labels, RS1 to RS7, represent Rackspace instances from the smallest to the biggest one. Other labels represent Amazon instance types used



(a) Local



(b) Amazon EC2

Figure 6: RP changes as resources and workload intensity change. Intensities are changed every ten minutes

load is sensitive to the heap size because it affects the amount of cached contents and the frequency of flushing the cached requests in Cassandra. This effect would be more obvious if there are more write operations. Therefore, the recommendation for light YCSB1 is m1.small against RS2 because its memory space is larger.

Table 2: Most recommended instance types for running certain workloads with desired performance and less cost

Applications	Light	Medium	Heavy
Cloudstone	RS3	RS3	m1.large
Wiki	RS3	m1.medium	m1.large
YCSB1	m1.small	m1.medium	m1.medium
YCSB2	RS2	m1.medium	m1.medium
Darwin	RS3	RS3	m1.medium
Cloud9	RS3	RS3	RS3

For the medium workload intensity, the recommended Rackspace instances for YCSB1 and YCSB2 are both RS4, where the recommended Amazon instances are m1.medium. Although RS4 provides higher performance than m1.medium for these workloads, RS4 is more expensive and its RPs here are more than one than m1.medium. Therefore, the recommended instances for medium YCSB1 and YCSB2 are both m1.medium. For the rest of the applications with medium workload intensity, we mostly select the one with higher RP between RS3 and m1.medium.

For the heavy workload intensity, Cloudstone and Wiki choose m1.large against RS4 because of the higher performance with the same price. The situation for the heavy YCSB is the same as its medium case. The case of Darwin chooses m1.medium because Darwin does not need more CPU cores but more memory would be helpful. On the other hand, the heavy Cloud9 desires more CPU cores than memory. Thus, the heavy Cloud9 chooses RS3 over m1.medium.

Choosing the right instance types to minimize cost and optimize performance for a certain workload requires sophisticated analysis on application and platform characteristics. Such processes could be very time consuming without the help of Matrix.

5.2 Multi-Machine Case

Many cloud applications are designed to work on multiple computers and communicate via a network. In this section, we first start the tests on a local VC. For profiling the system under different resource configurations, the number of VMs in a VC ranges from one, two, four to eight; the VCPU numbers on one VM is varied from one to four; and the size of memory on one VM is also varied from one to four GB. In other words, we have 64 VC settings in terms of the VM numbers, VCPUs, and memory sizes. We assume all VMs in a cluster are identical and leave the heterogeneous or asymmetric clusters as future work. We collect required profiling statistics from five runs of each representative application on all 64 VC settings. In order to capture the dynamics of various workload intensities, training applications will be uniformly randomly configured with thread/worker numbers from 2 to 128 and working set sizes from 20 to 100 GB in each run, in total 9,600 data points.

For our tests on the public cloud, the instance types included as the Amazon VC instances for training and testing are *t1.micro*, *m1.small*, *m1.medium*, *m1.large*, *m1.xlarge*, and *m2.xlarge*. Similar to the local VC test, the number of VMs in an Amazon VC ranges from one, two, four to eight. Thus, we have 24 VC settings on EC2. The workload intensity is changed for profiling in the

same way as it is in profiling local VCs. We also profiled VCs on Rackspace. The instance types used are RS1 to RS5. The rest processes and settings on Rackspace are similar to what we did on Amazon.

Prediction Accuracy. We first explore the accuracies on predicting RPs at clusters with different VM types and various numbers of VMs. Because of the space limit, we omit some figures. In general, the mean accuracy across all cases is 90.18% with a standard deviation of 2.55, where the mean accuracies on Amazon and Rackspace are 90.05% and 90.3% respectively.

From the accuracy tests, we found that Matrix has relatively good prediction accuracies on some applications, e.g., YCSB3 and YCSB4. Take the YCSB3, a read only testing workload in Zipfian distribution, as an example. Matrix effectively identifies this as an 100% read workload with an over 95% possibility. Among three possible distributions for pure read requests, Matrix recognizes this workload has an over 75% possibility to follow Zipfian distribution. This contributes to a relatively high accuracy for YCSB3.

To further analyze influences of representative applications in the training set, we remove five workloads at a time from the training set. Then, all models are rebuilt from the new training set. Next, we examine the accuracies on predicting RPs of applications in the testing set on a four-VM cluster whose VMs identically have four VCPUs and four GB memory. This test is repeated three times and the average accuracies are reported in Fig. 7. We remove CPU-intensive training applications first, the YCSB5 shows larger degradation than the others in the beginning because it consumes more CPU in scanning records when processing requests. When we start to remove data-intensive training workloads (the training set size is less than 15), all three testing applications drop dramatically. When we reduce the training size from ten to five, YCSB1 and YCSB5 both drop more than 20% because key genes (the 50/50 and 100/0 workload in the Zipfian distribution for YCSB1 and YCSB5 respectively) are removed. YCSB4 holds higher than the others at the training size of five because the 100/0 workload in the latest distribution, which represents most of the YCBS4, is still kept in the final five.

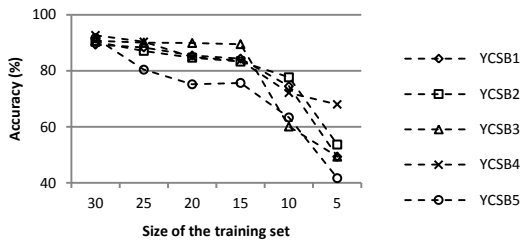


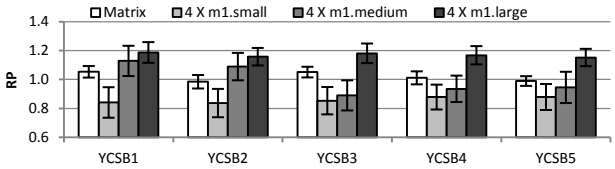
Figure 7: Accuracies on predicting RP decrease as the size of training set shrinks

VC in Private Cloud. We also verified automatically configuring a VC's resources to maintain the desired performance. The test is similar to the one in Fig. 6a, but has more dimensions in resources, e.g., number of machines, and workloads, e.g., changing workload types. Due to the space limit, we omit some figures of this test. In brief, Matrix tracks workload activities closely and is able to change VM configuration quickly and keep RPs on track.

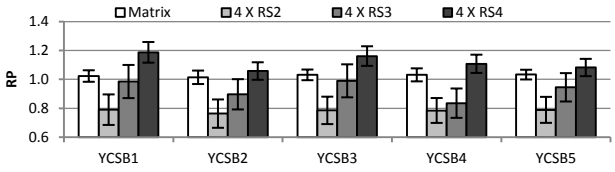
VC in Public Clouds. Here we will only change the type of instance. We did not use Xen-blanket here due to concern over the overhead of nested virtualization. In this case, we run the tests in three steps: 1) Each application in the testing set is executed for ten minutes on a VC with a randomly uniformly selected type and the number of VMs from one to eight. 2) Matrix collects required system statistics, and recommends a configuration. 3) The same application then runs on a cluster of instances closest to the recommended configuration. We repeat the above steps at the weekday daytime, the weekday nighttime, and the weekend. In addition, we change workload intensity from light, medium, and heavy for each testing application.

Fig. 8 shows the average RPs and standard deviations when we re-run testing cases with the recommended configurations as well as three fixed size VCs. Each column shows the average RP of 45 runs. Fig. 8a and Fig. 8b are results from Amazon and Rackspace respectively. All the RPs from Matrix spread between 0.88 and 1.16 with the mean of 1.02 across all cases. As it is shown in Fig. 8, using configurations suggested by Matrix makes the average RPs closer to one and smaller in variance than using the three static configurations. It leads to a low average RP value of 0.82 when using $4 \times m1.small$ or $4 \times RS2$ all the time because the medium and the heavy workloads are too intensive for it. In general, Matrix uses $4 \times m1.small$ or $4 \times RS2$ at light workloads but uses more powerful instances when workload is heavier. The average RP of all $4 \times m1.medium$ and $4 \times RS3$ cases is close to one but its standard deviation is 0.1, which is more than twice of the one of Matrix (0.04). The large variance in RPs of the $4 \times m1.medium$ and $4 \times RS2$ case comes from over-provisioning at the light workload, inadequacy at the heavy one, and the difference in the workload mix, even at the appropriate intensity. For example, Matrix uses $3 \times m1.medium$ for YCSB1 and $2 \times m1.large$ for YCSB5 at the medium workload which makes RPs closer to one than the $4 \times m1.medium$ does. When the workload is heavy, Matrix uses $2 \times m1.large$ or $2 \times RS4$ most of the time. Thus, although statically using $4 \times m1.large$ and $4 \times RS4$ has small variance values, the average RP in this case increases to 1.14.

Cost Efficiency. Here we examine the RPC and PPC values to see the cost-efficiency of each configuration.



(a) Amazon



(b) Rackspace

Figure 8: RPs when using Matrix and three static cluster settings on Amazon and Rackspace

To ease comparison, the RPC and PPC values in Table 3 are normalized to Matrix's . A smaller RPC means more cost-efficient while keeping desired RPs, and on the other hand, a higher PPC means more RP can be achieved for the same cost. In both the tests on Amazon and Rackspace, Matrix outperforms other static settings in both metrics.

Table 3: Cost efficiency (RPC and PPC) of Matrix and three static configurations on Amazon and Rackspace respectively

Amazon EC2

	Matrix	$4 \times m1.small$	$4 \times m1.medium$	$4 \times m1.large$
RPC	1.00	24.00	20.41	143.02
PPC	1.00	0.84	0.47	0.33

Rackspace cloud servers

	Matrix	$4 \times RS2$	$4 \times RS3$	$4 \times RS4$
RPC	1.00	25.33	18.67	90.54
PPC	1.00	0.78	0.68	0.52

5.3 Private to Public Cloud

In this case, we evaluate the case of migrating a virtual cluster from private to public cloud. We make the number of VMs per cluster larger than previous tests in order to test the scalability. Because our Rackspace account has a limitation on memory size at 64 GB, the results of public cloud here are all obtained from the Amazon EC2. We use 32- and 64-VM local VCs (VC32 and VC64) in this case. These local VCs are hosted on four $VS2$ servers. Each VM in one local VC has one VCPU and 1.5 GB memory, and each $VS2$ hosts 16 VMs. The training procedure on EC2 is almost the same as the previous one in Sec. 5.2, except that we extend the number of VMs to 32 and 64 in the procedure. We then verify the prediction accuracies in Amazon VCs. Because of the space limit, we omit some figures of this test. The average accuracy across different clusters is 0.89 with the standard deviation of 0.03.

We also make Matrix to recommend EC2 configurations comparable to the 32- and 64-VM local VCs for running the light, medium, and heavy testing workloads, which have 8, 32, and 64 threads and 80, 160, and 320 GB working set size respectively. We run each testing application and intensity for 30 times on a VC with $32 \times m1.xlarge$ instances for Matrix to find the matched configurations. In general, Matrix mostly uses $30 \times m1.medium$, $24 \times m1.large$, and $20 \times m1.xlarge$ instances for the VC32 at the light, medium, and heavy workloads respectively. When the cluster size increases from 32 to 64, Matrix makes the EC2 cluster to use more instances correspondingly. The configuration for the light workload is changed from $30 \times m1.medium$ to $64 \times m1.medium$. The configurations for the medium and heavy workloads become $44 \times m1.large$ and $36 \times m1.xlarge$ respectively. Using the suggested configurations gives average RPs of 1.02 with the standard deviations of 0.07 across different workload intensities. The RPs for all the cases spread between 0.88 and 1.16 with the mean of 1.03.

We also verified the PPC and RPC values of Matrix in this test. Due to the space limit, we omit a table here. According to the RPC values, Matrix costs much less than the static EC2 VC settings, especially at the VC64 tests. Further, Matrix demonstrates better PPC values than the static settings, which indicates a good performance-cost efficiency. The PPC values also show that using powerful instances may be not cost-efficient although they do provide better performance.

6 Conclusion

In this paper, we have presented Matrix, a performance prediction and resource management system. Matrix utilizes clustering methods with probability estimates to classify new cloud workloads and provide advice about what instance types will offer the desired performance level and the lowest cost. Matrix uses machine learning techniques and an approximation algorithm to build performance models, and uses them for managing on-line resources when applications are moved to the cloud. We demonstrated that our models have high accuracy, even when transitioning a distributed application from a cluster of physical machines to a set of cloud VMs. Matrix helps to keep a desired performance in the cloud while minimizing the operating cost.

As future work, Matrix may be extended to study the mapping from a local storage to a cloud one, such as the Amazon EBS. The cost model in Matrix could be more complete by including the charge on data usage. Also, we may expand the load balancing ability of Matrix to handle heterogeneous or asymmetric cluster machines and workload intensities.

References

[1] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[2] J. L. Bonebakker. Finding representative workloads for computer system design. Technical report, Mountain View, CA, USA, 2007.

[3] B. E. Boser, I. M. Guyon, and V. N. Vapnik. A training algorithm for optimal margin classifiers. In *Proceedings of the fifth annual workshop on Computational learning theory*, COLT '92, pages 144–152, New York, NY, USA, 1992. ACM.

[4] S. Bucur, V. Ureche, C. Zamfir, and G. Candea. Parallel symbolic execution for automated real-world software testing. In *Proceedings of the sixth conference on Computer systems*, EuroSys '11, pages 183–198, New York, NY, USA, 2011. ACM.

[5] R. Buyya, J. Broberg, and A. M. Goscinski. *Cloud Computing Principles and Paradigms*. Wiley Publishing, 2011.

[6] C.-C. Chang and C.-J. Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27, 2011. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

[7] R. C. Chiang and H. H. Huang. Tracon: interference-aware scheduling for data-intensive applications in virtualized environments. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 47:1–47:12, New York, NY, USA, 2011. ACM.

[8] L. Ciortea, C. Zamfir, S. Bucur, V. Chipounov, and G. Candea. Cloud9: a software testing service. *SIGOPS Oper. Syst. Rev.*, 43(4):5–10, Jan. 2010.

[9] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[10] S. P. E. Corporation. Spec cpu2006. `http://www.spec.org/cpu2006/`.

[11] K. Duan, S. Keerthi, and A. N. Poo. Evaluation of simple performance measures for tuning svm hyperparameters. *Neurocomputing*, 51(0):41 – 59, 2003.

[12] B. Farley, A. Juels, V. Varadarajan, T. Ristenpart, K. D. Bowers, and M. M. Swift. More for your money: exploiting performance heterogeneity in public clouds. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 20:1–20:14, New York, NY, USA, 2012. ACM.

[13] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: Guaranteed job latency in data parallel clusters. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 99–112, 2012.

[14] H. Herodotou, F. Dong, and S. Babu. No one (cluster) size fits all: Automatic cluster sizing for data-intensive analytics. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, SOCC '11, pages 18:1–18:14, 2011.

[15] Z. Hill, J. Rowanhill, A. Nguyen-Tuong, G. Wasson, J. Knight, J. Basney, and M. Humphrey. Meeting virtual organization performance goals through adaptive grid reconfiguration. In *Grid Computing, 2007 8th IEEE/ACM International Conference on*, pages 177–184, 2007.

[16] A. Iosup, S. Ostermann, M. Yigitbasi, R. Prodan, T. Fahringer, and D. H. J. Epema. Performance analysis of cloud computing services for many-tasks scientific computing. *Parallel and Distributed Systems, IEEE Transactions on*, 22(6):931–945, 2011.

[17] R. K. Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley, 1 edition, Apr. 1991.

[18] V. Jalaparti, H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Bridging the tenant-provider gap in cloud services. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 10:1–10:14, 2012.

[19] Y. Koh, R. Knauerhase, P. Brett, M. Bowman, Z. Wen, and C. Pu. An analysis of performance interference effects in virtual environments. In *In Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2007.

[20] A. Kopytov. Sysbench. `http://sysbench.sourceforge.net/index.html`.

[21] S. Kundu, R. Rangaswami, K. Dutta, and M. Zhao. Application performance modeling in a virtualized environment. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1 –10, 2010.

[22] S. Kundu, R. Rangaswami, A. Gulati, M. Zhao, and K. Dutta. Modeling virtualized applications using machine learning techniques. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*, VEE '12, pages 3–14, New York, NY, USA, 2012. ACM.

[23] M. Kutare, G. Eisenhauer, C. Wang, K. Schwan, V. Talwar, and M. Wolf. Monalytics: online monitoring and analytics for managing large scale data centers. In *Proceedings of the 7th international conference on Autonomic computing*, ICAC '10, pages 141–150, New York, NY, USA, 2010. ACM.

[24] S. Lacour, C. Perez, and T. Priol. Generic application description model: toward automatic deployment of applications on computational grids. In *Grid Computing, 2005. The 6th IEEE/ACM International Workshop on*, pages 4 pp.–, 2005.

[25] H. Li, G. Fox, and J. Qiu. Performance model for parallel matrix multiplication with dryad: Dataflow graph runtime. In *Cloud and Green Computing (CGC), 2012 Second International Conference on*, pages 675–683, 2012.

[26] M. Maurer, I. Brandic, and R. Sakellariou. Self-adaptive and resource-efficient sla enactment for cloud computing infrastructures. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 368–375, 2012.

[27] R. McDougall, J. Crase, and S. Debnath. Filebench. `http://sourceforge.net/projects/filebench/`.

[28] A. Menon, J. R. Santos, Y. Turner, G. J. Janakiraman, and W. Zwaenepoel. Diagnosing performance overheads in the xen virtual machine environment. In *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, VEE '05, pages 13–23, New York, NY, USA, 2005. ACM.

[29] Y. Nakajima, Y. Aida, M. Sato, and O. Tatebe. Performance evaluation of data management layer by data sharing patterns for grid rpc applications. In *Euro-Par 2008 Parallel Processing*, volume 5168 of *Lecture Notes in Computer Science*, pages 554–564. Springer Berlin Heidelberg, 2008.

[30] R. Nathuji, A. Kansal, and A. Ghaffarkhah. Q-clouds: managing performance interference effects for qos-aware clouds. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 237–250, New York, NY, USA, 2010. ACM.

[31] O. Niehörster, A. Brinkmann, A. Keller, C. Kleineweber, J. Krüger, and J. Simon. Cost-aware and slo-fulfilling software as a service. *Journal of Grid Computing*, 10:553–577, 2012.

[32] D. Novaković, N. Vasić, S. Novaković, D. Kostić, and R. Bianchini. Deepdive: Transparently identifying and managing performance interference in virtualized environments. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 219–230, 2013.

[33] Z. Ou, H. Zhuang, J. K. Nurminen, A. Ylä-Jääski, and P. Hui. Exploiting hardware heterogeneity within the same instance type of amazon ec2. In *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*, HotCloud'12, pages 4–4, Berkeley, CA, USA, 2012. USENIX Association.

[34] G. J. Popek and R. P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17:412–421, July 1974.

[35] A. Quiroz, H. Kim, M. Parashar, N. Gnanasambandam, and N. Sharma. Towards autonomic workload provisioning for enterprise grids and clouds. In *Grid Computing, 2009 10th IEEE/ACM International Conference on*, pages 50–57, 2009.

[36] L. Ramakrishnan, R. S. Canon, K. Muriki, I. Sakrejda, and N. J. Wright. Evaluating interconnect and virtualization performance for high performance computing. In *Proceedings of the second international workshop on Performance modeling, benchmarking and simulation of high performance computing systems*, PMBS '11, pages 1–2, New York, NY, USA, 2011. ACM.

[37] J. Rao, X. Bu, C.-Z. Xu, L. Wang, and G. Yin. Vconf: a reinforcement learning approach to virtual machines auto-configuration. In *Proceedings of the 6th international conference on Autonomic computing*, ICAC '09, pages 137–146, New York, NY, USA, 2009. ACM.

[38] P. Ruth, J. Rhee, D. Xu, R. Kennell, and S. Goasguen. Autonomic live adaptation of virtual computational environments in a multi-domain infrastructure. In *Autonomic Computing, 2006. ICAC '06. IEEE International Conference on*, pages 5–14, 2006.

[39] B. Schölkopf, J. C. Platt, J. C. Shawe-Taylor, A. J. Smola, and R. C. Williamson. Estimating the support of a high-dimensional distribution. *Neural Comput.*, 13(7):1443–1471, July 2001.

[40] B. Schölkopf, A. J. Smola, R. C. Williamson, and P. L. Bartlett. New support vector algorithms. *Neural Comput.*, 12(5):1207–1245, May 2000.

[41] U. Sharma, P. Shenoy, S. Sahu, and A. Shaikh. A cost-aware elasticity provisioning system for the cloud. In *Distributed Computing Systems (ICDCS), 2011 31st International Conference on*, pages 559 –570, june 2011.

[42] W. Sobel, S. Subramanyam, A. Sucharitakul, J. Nguyen, H. Wong, A. Klepchukov, S. Patil, A. Fox, and D. Patterson. Cloudstone: Multi-platform, multi-language benchmark and measurement tools for web 2.0. In *The first workshop on Cloud Computing and its Applications*, CCA '08, 2008.

[43] A. A. Soror, U. F. Minhas, A. Aboulnaga, K. Salem, P. Kokosielis, and S. Kamath. Automatic virtual machine configuration for database workloads. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 953–966, New York, NY, USA, 2008. ACM.

[44] C. Stewart, T. Kelly, and A. Zhang. Exploiting nonstationarity for performance prediction. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 31–44, New York, NY, USA, 2007. ACM.

[45] Y. Tan, H. Nguyen, Z. Shen, X. Gu, C. Venkatramani, and D. Rajan. Prepare: Predictive performance anomaly prevention for virtualized cloud systems. In *Distributed Computing Systems (ICDCS), 2012 IEEE 32nd International Conference on*, pages 285 –294, june 2012.

[46] O. Tickoo, R. Iyer, R. Illikkal, and D. Newell. Modeling virtual machine performance: challenges and approaches. *SIGMETRICS Perform. Eval. Rev.*, 37(3):55–60, Jan. 2010.

[47] G. Urdaneta, G. Pierre, and M. van Steen. Wikipedia workload analysis for decentralized hosting. *Elsevier Computer Networks*, 53(11):1830–1845, July 2009. `http://www.globule.org/publi/WWADH_comnet2009.html`.

[48] N. Vasić, D. Novaković, S. Miučin, D. Kostić, and R. Bianchini. Dejavu: accelerating resource allocation in virtualized environments. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '12, pages 423–436, New York, NY, USA, 2012. ACM.

[49] A. Verma, L. Cherkasova, and R. H. Campbell. Aria: Automatic resource inference and allocation for mapreduce environments. In *Proceedings of the 8th ACM International Conference on Autonomic Computing*, ICAC '11, pages 235–244, 2011.

[50] B. J. Watson, M. Marwah, D. Gmach, Y. Chen, M. Arlitt, and Z. Wang. Probabilistic performance modeling of virtualized resource allocation. In *Proceedings of the 7th international conference on Autonomic computing*, ICAC '10, pages 99–108, New York, NY, USA, 2010. ACM.

[51] A. Wieder, P. Bhatotia, A. Post, and R. Rodrigues. Orchestrating the deployment of computations in the cloud with conductor. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation*, pages 367–381. USENIX, 2012.

[52] Wikimedia Foundation. Wikipedia:Database download. `http://en.wikipedia.org/wiki/Wikipedia:Database_download`.

[53] D. Williams, H. Jamjoom, and H. Weatherspoon. The xen-blanket: virtualize once, run everywhere. In *Proceedings of the 7th ACM european conference on Computer Systems*, EuroSys '12, pages 113–126, New York, NY, USA, 2012.

[54] T. Wood, L. Cherkasova, K. Ozonat, and P. Shenoy. Profiling and modeling resource usage of virtualized applications. In *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '08, pages 366–387, New York, NY, USA, 2008. Springer-Verlag New York, Inc.

[55] W. Zheng, R. Bianchini, G. J. Janakiraman, J. R. Santos, and Y. Turner. Justrunit: experiment-based management of virtualized data centers. In *Proceedings of the 2009 conference on USENIX Annual technical conference*, USENIX'09, pages 18–18, Berkeley, CA, USA, 2009. USENIX Association.

Adaptive, Model-driven Autoscaling for Cloud Applications

Anshul Gandhi, Parijat Dube, Alexei Karve, Andrzej Kochut, Li Zhang
IBM Research

Abstract

Applications with a dynamic workload demand need access to a flexible infrastructure to meet performance guarantees and minimize resource costs. While cloud computing provides the elasticity to scale the infrastructure on demand, cloud service providers lack control and visibility of user space applications, making it difficult to accurately scale the underlying infrastructure. Thus, the burden of scaling falls on the user.

In this paper, we propose a new cloud service, Dependable Compute Cloud (DC2), that automatically scales the infrastructure to meet the user-specified performance requirements. DC2 employs Kalman filtering to automatically learn the (possibly changing) system parameters for each application, allowing it to proactively scale the infrastructure to meet performance guarantees. DC2 is designed for the cloud - it is application-agnostic and does not require any offline application profiling or benchmarking. Our implementation results on OpenStack using a multi-tier application under a range of workload traces demonstrate the robustness and superiority of DC2 over existing rule-based approaches.

1 Introduction

With the advent of cloud computing, many application owners have started moving their deployments into the cloud. Cloud computing offers many benefits over traditional physical deployments including lower infrastructure costs and elastic resource allocation. These benefits are especially advantageous for applications with a dynamic workload demand. Such applications can be deployed in the cloud based on the current demand, and the deployment can be scaled dynamically in response to changing workload demand.

While cloud computing is a very promising option for application owners, it is not easy to take full advantage of the benefits of the cloud. Specifically, while cloud computing offers flexible resource allocation, it is up to the customer (application owner) to leverage the flexible infrastructure. That is, the user must decide when and how to scale the application deployment to meet the changing workload demand.

Dynamically sizing a deployment is challenging for many reasons (see, for example, the recent survey paper [15]). From the perspective of the user, who is also the application owner, some of the specific hurdles that complicate the dynamic sizing of the application are: (i) Requires expert knowledge about the dynamics of the application, including the service requirements of the application at each tier, and (ii) Requires sophisticated modeling expertise to determine when and how to resize the deployment. For small and medium businesses (SMB), which comprise the targeted customer base for many cloud service providers (CSPs) [8,24], these hurdles are non-trivial to overcome. SMB users would much rather contract a cloud service that manages their dynamic sizing than invest in employing a team of experts. The purpose of this research is to provide this exact service - an application-agnostic cloud offering that will automatically, and dynamically, resize user applications to meet performance requirements in a cost-effective manner.

Many CSPs today offer monitoring services to users (not necessarily for free) for tracking resource usage. While such monitoring services provide valuable information, the user still requires expert knowledge about the application and the performance modeling expertise to convert the monitored information into scaling actions.

Some CSPs also offer rule-based triggers to help users scale their applications. These rule-based triggers allow the users to specify some conditions on the monitored metrics which, when met, will trigger a pre-defined scaling action. Even with the help of rule-based triggers, however, the burden of determining the threshold conditions for the metrics still rests with the user. For example, in order to use a CPU utilization based trigger for scaling, the user must determine the CPU threshold at which to trigger scale-up and scale-down, and the number of instances to scale-up and scale-down.

Note that CSPs cannot gather all the necessary application-level statistics without intruding into the user-space application. Given the lack of control and visibility into the application, CSPs *cannot* leverage most of the existing work (see, for example, [7,25,26]) on dynamic scaling of applications since these works typically require access to the application for measurement and profiling purposes. Further, most of the existing work is *not* application-agnostic, which is a requirement for a practical cloud service. A detailed discussion of the related work can be found in Section 5

We propose a completely automated cloud service, Dependable Compute Cloud (DC2), that proactively and dynamically scales the application deployment based on user-specified performance requirements. DC2 leverages resource-level and application-level statistics to infer the

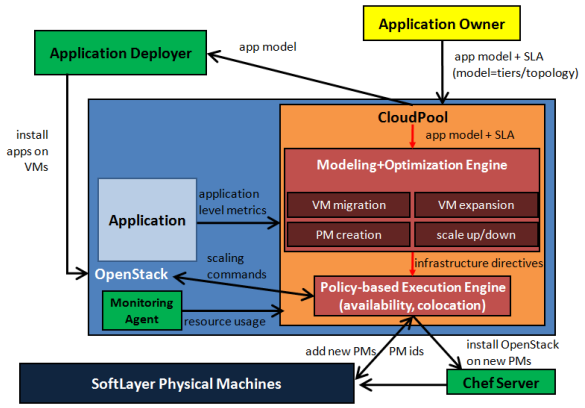


Figure 1: System architecture for DC2.

underlying system parameters of the application(s), and determines the required scaling actions to meet the performance goals in a cost-effective manner. A detailed discussion of our approach is presented in Section 2, along with details of our implementation.

At the heart of DC2 lies the modeling and execution engine that internalizes the monitored statistics and infers the necessary system parameters. While this engine can employ any grey-box or black-box modeling approach, in this paper we use Kalman filtering to infer the system parameters. Kalman filtering is a robust feedback control algorithm that combines monitoring information with a user-specified system model to create accurate estimations of the system state. In this paper we employ Kalman filtering by specifying a generic queueing-theoretic model (details of our modeling engine can be found in Section 3). Fortunately, since Kalman filtering leverages monitored statistics to come up with estimations, the underlying system model need not be accurate, as is often the case when using queueing theory (or any other mathematical modeling technique) to model complex systems.

We evaluate DC2 via implementation on OpenStack. We employ the three-tier bidding benchmark, RUBiS, as the user application and experiment with various workload traces. Our results demonstrate that DC2 successfully scales the application in response to changing workload demand without any user input and without any offline profiling. Importantly, we compare with existing rule-based triggers and show that DC2 is superior to such approaches. A detailed evaluation of our DC2 implementation is presented in Section 4.

2 Implementation

Figure 1 shows the proposed system architecture for the DC2 service environment. The Application Owner (customer) is responsible for providing the initial deployment model consisting of the multi-tier topology for the application (in the form of a graph or a configuration file) and the performance SLA requirements. The Application Deployer customizes the image for deployment and ties up the endpoints for the application during installation and configuration. We leverage Chef [18] to automate the installation of software on VMs during boot. We use OpenStack [17] as the underlying scalable cloud operating system. The VMs for the application are created on an OpenStack managed private cloud deployment on SoftLayer [23]. The CloudPool component in Figure 1 is a logical entity that models the application and issues the directives (such as VM scale up/down) required to maintain the performance SLA for the application. The Monitoring Agent is responsible for retrieving the resource-level metrics from the hypervisor and the application-level metrics from the application. The Modeling + Optimization Engine (described in detail in Section 3) takes as input the monitored metrics and outputs a list of directives indicating the addition or removal of VMs, migration of VMs, or a change in the resources allocated to VMs. These directives are passed on to the Policy-based Execution Engine that issues commands to OpenStack API, that in turn performs the scaling.

2.1 Application

We use the open source multi-tier application, RUBiS [2], for our experiments. RUBiS is an auction site prototype modeled after eBay.com supporting different classes of web requests such as bid, browse, buy, etc. Our RUBiS implementation employs Apache as the frontend web server, Tomcat as the Java servlets container, and MySQL as the backend database. In our experiments we focus on scaling the Tomcat application tier. We employ RUBiS's benchmarking tool to generate load by defining sessions consisting of a sequence of requests. The think time between requests is exponentially distributed with a mean of 1 second. We fix the number of clients for each experiment and vary the load by dynamically changing the composition of the workload mix.

2.2 Experimental setup

We employ multiple hypervisors with 8 CPU cores and 8 GB of memory each. The Apache and MySQL tiers are each hosted on a 4 CPU VM. The Tomcat application tier is hosted on multiple 2 CPU VMs. The provisioning time for a new Tomcat VM is about 30-40 seconds. Once the new VM is online, our automated scripts configure the JDBC with the IP address of the MySQL database and update the load balancer on Apache web server to include the new Tomcat VM.

2.3 Monitoring agent

We use virt-top (part of the libvirt [1] package) to collect VM CPU utilization statistics from each hypervisor periodically. For the application-level metrics, we periodically analyze the request URLs directed at the RU-

BiS application to compute the request rate and response time. Note that the user can choose to provide these metrics to us directly (for example, using a REST call). The monitoring interval is set to 10s. The collected statistics are then provided as input to the modeling engine.

2.4 Execution engine

The execution engine is primarily responsible for issuing commands for VM scaling based on the scaling directives received from the modeling engine. For robustness, the execution engine issues the VM scaling commands to OpenStack only after two successive scaling directives from the modeling engine. The execution engine is also responsible for placing the new VMs on specific hypervisors. We use host aggregates (which are essentially logical cloud partitions) to place the Apache and MySQL VMs on one hypervisor and Tomcat VMs on a different set of hypervisors.

3 Modeling

The modeling engine lies at the heart of our DC2 approach. We use a queueing-network model to *approximate* our multi-tier cloud application. However, since we cannot access the user application to derive the parameters of our model, we use a Kalman filtering technique to infer these unobservable parameters. We now describe our queueing model and Kalman filtering technique, followed by an analysis of our modeling engine, and finally, an explanation of how our modeling engine determines the required scaling actions for SLA compliance.

3.1 Queueing-network model

Figure 2 shows a queueing-network model of a generic three-tier system with each tier representing a collection of homogeneous servers. We assume that the load at each tier is distributed uniformly across all the servers in that tier. The system is driven by a workload consisting of i distinct request classes, each class being characterized by its arrival rate, λ_i, and end-to-end response time, R_i. Let n_j be the number of servers at tier j. With homogeneous servers and perfect load-balancing, the arrival rate of requests at any server in tier j is $\lambda_{ij} := \lambda_i/n_j$. Since servers at a tier are identical, for ease of analysis, we model each tier as a single representative server. With some abuse of terminology, we refer to the representative server at tier j as tier j. Let $u_j \in [0,1)$ be the utilization of tier j. The background utilization of tier j is denoted by u_{0j}, and models the resource utilization due to other jobs (not related to our workload) running on that tier. The end-to-end network latency for a class i request is denoted by d_i. Let $S_{ij}(\geq 0)$ denote the average service time of a class i request at tier j. Assuming we have Poisson arrivals and a processor-sharing policy at each server, the stationary distribution of the queueing network is known to have a

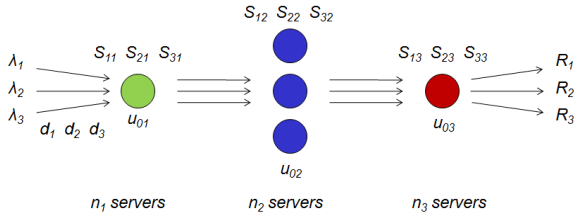


Figure 2: Queueing model for our system.

product-form [28], for any general distribution of service time at servers. Under the product-form assumption, we have the following analytical results from queueing theory:

$$u_j = u_{0j} + \sum_i \lambda_{ij} S_{ij}, \quad \forall j \tag{1}$$

$$R_i = d_i + \sum_j \frac{S_{ij}}{1-u_j}, \quad \forall i \tag{2}$$

While u_j, R_i and λ_i, $\forall i, j$, can be monitored easily and are thus observable, the parameters S_{ij}, u_{0j}, and d_i are non-trivial to measure and are thus unobservable. While existing work on auto-scaling (see, for example, [25,26]) typically obtains these values by directly accessing or modifying the application software (for example, by parsing the log files at each tier), our proposed application-agnostic cloud service *cannot* encroach the user's application space. Instead, we employ a parameter estimation technique, Kalman filtering (see Section 3.2 below), to derive estimates for the unobservable parameters. Further, since the system parameters can dynamically change during runtime, we employ the Kalman filter as an on-line parameter estimator to continually adapt our parameter estimates.

It is important to note that while the product-form is shown to be a reasonable assumption for tiered web services [5], we only use it as an approximation for our complex system. By employing the Kalman filter to leverage the actual monitored values, we minimize our dependence on the approximation.

3.2 Kalman filtering

For a three-class, three-tier system (i.e., $i = j = 3$), let $\mathbf{z} := (u_1, u_2, u_3, R_1, R_2, R_3)^T = \mathbf{h}(\mathbf{x})$ and $\mathbf{x} = (u_{01}, u_{02}, u_{03}, d_1, d_2, d_3, S_{11}, S_{21}, S_{31}, S_{12}, S_{22}, S_{32}, S_{13}, S_{23}, S_{33})^T$. Note that $\mathbf{z}$ is a 6-dimensional vector whereas $\mathbf{x}$ is a 15-dimensional vector. The problem is to determine the unobservable parameters $\mathbf{x}$ from measured values of $\mathbf{z}$ and $\lambda = (\lambda_1, \lambda_2, \lambda_3)$.

We use Kalman filtering to estimate the unobservable parameters (for a detailed explanation of Kalman filters, we refer the reader to [22]). The dynamic evolution of system parameters can be described through the following

Kalman filtering equations [22]:

$$\begin{aligned} \text{System State} \quad & \mathbf{x}(t) = \mathbf{F}(t)\mathbf{x}(t-1) + \mathbf{w}(t), \\ \text{Measurement Model} \quad & \mathbf{z}(t) = \mathbf{H}(t)\mathbf{x}(t) + \mathbf{v}(t), \end{aligned}$$

where $\mathbf{F}(t)$ is the state transition model and $\mathbf{H}(t)$ is the observation model mapping the true state space into the observed state space. In our case, $\mathbf{F}(t), \forall t$, is the identity matrix. The variables $\mathbf{w}(t) \sim \mathcal{N}(0, \mathcal{Q}(t))$ and $\mathbf{v}(t) \sim \mathcal{N}(0, \mathcal{R}(t))$ are process noise and measurement noise which are assumed to be zero-mean, multi-variate Normal distributions with covariance matrices $\mathcal{Q}(t)$ and $\mathcal{R}(t)$ respectively. The matrices $\mathcal{Q}(t)$ and $\mathcal{R}(t)$ are not directly measurable but can be tuned via best practices [14].

Since the measurement model $\mathbf{z}$ is a non-linear function of the system state $\mathbf{x}$ (see Eqns. (1) and (2)), we use the Extended Kalman filer [22] with $\mathbf{H}(t) = \left[\frac{\partial \mathbf{h}}{\partial \mathbf{x}}\right]_{\mathbf{x}(t)}$, which for our model is a 6×15 matrix with $\mathbf{H}(t)_{ij} = \left[\frac{\partial \mathbf{h_i}}{\partial \mathbf{x_j}}\right]_{\mathbf{x}(t)}$.

Since $\mathbf{x}(t)$ is not known at time t, we estimate it by $\hat{\mathbf{x}}(t|t-1)$, which is the *a priori* estimate of $x(t)$ given all the history up to time $t-1$. The state of the filter is described by two variables $\hat{\mathbf{x}}(t|t)$ and $\mathbf{P}(t|t)$, where $\hat{\mathbf{x}}(t|t)$ is the *a posteriori* estimate of state at time t and $\mathbf{P}(t|t)$ is the *a posteriori* error covariance matrix which is a measure of the estimated accuracy of the system state.

The Kalman filter has two phases: Predict and Update. In the predict phase, *a priori* estimates of state and error matrix are calculated. In the update phase, these estimates are refined using the current observation to get *a posteriori* estimates of state and error matrix. The filter model for the predict and update phases for our 3-class, 3-tier model is given by:

Predict:

$$\begin{aligned} \hat{\mathbf{x}}(t|t-1) &= \mathbf{F}(t)\hat{\mathbf{x}}(t-1|t-1) \\ \mathbf{P}(t|t-1) &= \mathbf{F}(t)\mathbf{P}(t-1|t-1)\mathbf{F}^T(t) + \mathcal{Q}(t) \end{aligned}$$

Update:

$$\begin{aligned} \mathbf{y}(t) &= \mathbf{z}(t) - \mathbf{h}(\hat{\mathbf{x}}(t|t-1)) \\ \mathbf{H}(t) &= \left[\frac{\partial \mathbf{h}}{\partial \mathbf{x}}\right]_{\hat{\mathbf{x}}(t|t-1)} \\ \mathbf{S}(t) &= \mathbf{H}(t)\mathbf{P}(t|t-1)\mathbf{H}^T(t) + \mathcal{R}(t) \\ \mathbf{K}(t) &= \mathbf{P}(t|t-1)\mathbf{H}^T(t)\mathbf{S}^{-1}(t) \\ \hat{\mathbf{x}}(t|t) &= \hat{\mathbf{x}}(t|t-1) + \mathbf{K}(t)\mathbf{y}(t) \\ \mathbf{P}(t|t) &= (\mathbf{I} - \mathbf{K}(t)\mathbf{H}(t))\mathbf{P}(t|t-1) \end{aligned}$$

We employ the above filter model by seeding our initial estimate of $\hat{\mathbf{x}}(t|t-1)$ and $\mathbf{P}(t|t-1)$ with random values, then applying the Update equations by monitoring $\mathbf{z}(t)$ to get $\hat{\mathbf{x}}(t|t)$ and $\mathbf{P}(t|t)$, and finally using the Predict values to arrive at the estimated $\hat{\mathbf{x}}(t|t-1)$ and $\mathbf{P}(t|t-1)$. We continue this process iteratively at each 10 second monitoring interval to derive new system state estimates.

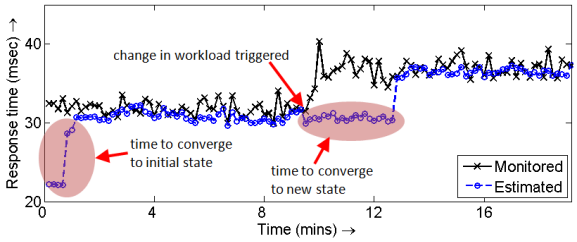


Figure 3: Accuracy and convergence of our Kalman filtering technique when employed in our experiments.

3.3 Performance analysis

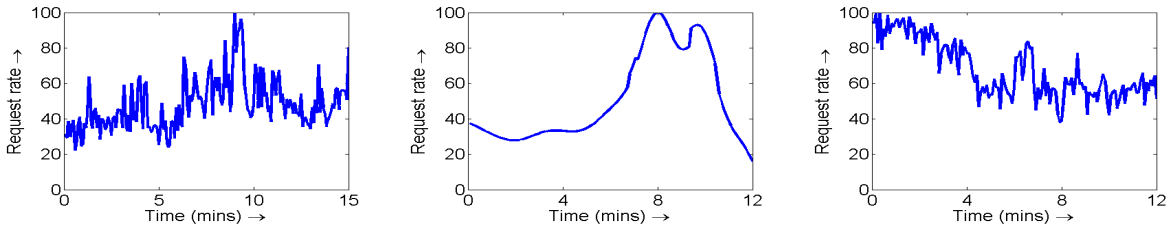
The Kalman filtering technique described above gives us estimates of the unobservable system parameters S_{ij}, u_{0j}, and d_i. We then use these estimates, along with Eqns. (1) and (2), to predict the future values of u_j and R_i. Figure 3 demonstrates our Kalman filtering technique in action. The solid line with crosses shows the monitored values of response time for a specific class of requests in our three-tier application (see Section 2 for details of our application setup). Here, the monitoring interval is 10 seconds. The dashed line with circles shows our estimated values for the predicted response time based on our Kalman filtering technique. It initially takes about a minute for our estimates to converge. After convergence, our estimated values are in very good agreement with the monitored values, thus validating our technique and highlighting its accuracy. Since we leverage the *current* monitored values of $\mathbf{z}$ and λ, our estimated system parameters can adapt to changes in the application. In order to demonstrate this ability, we trigger a change in our workload at about the 10-minute mark (shown in Figure 3) which causes the response time to increase. The change in the workload causes a change in the service time of the requests. Our Kalman filter detects this change based on the monitored values, and quickly adapts (in about 2 minutes) its estimates to converge to the new system state.

3.4 Scaling directives

The estimated values of the system state are used to compute the required scaling actions for DC2. Specifically, given the response time SLA, we use Eqns. (1) and (2) to determine the minimum n_j required to ensure SLA compliance. Note that $\lambda_{ij} = \lambda_i / n_j$ in Eqn. (1). We demonstrate the auto-scaling abilities of the Kalman filtering-based DC2 approach in Section 4.

4 Evaluation

We now evaluate our DC2 scaling policy in various settings using the RUBiS application. We use traces from the WITS traffic archive [29] and the WorldCup98 dataset from the Internet Traffic Archive (ITA) [11] to drive our load generator. The WITS archive contains a large collection of recent internet traces from ISPs and University networks. The WorldCup98 dataset contains



(a) Bursty trace (source:WITS [29]) (b) Rampdown trace (source:ITA [11]) (c) Hill trace (source:WITS [29])

Figure 4: Traces (normalized) used for our experiments.

3 months worth of requests made to the 1998 World Cup Web site. We scaled the traces to fit our deployment. The normalized traces are shown in Figure 4. The workload we use for evaluation is a mix of different RUBiS request classes that together stress the application tier more than the other tiers.

In our experiments, we focus on the response time of browse requests since customers often base their web experience based on how long it takes to browse through online catalogues. We want the response time for the browse requests to be less than 40ms, on average, for *every* 10s monitoring interval. Note that this goal is much more challenging than requiring the response time be less than 40ms over the entire length of the experiment. We set the response time SLA for all other classes to be 100ms. The secondary goal is to minimize the number of application tier VMs employed during the experiment. We consider the following two metrics: **V**, the percentage of time that the response time SLA was violated, and **K**, the average number of application tier VMs used during the experiment. For each experiment, we compare DC2 with the following class of policies:

THRES(x,y) is a family of rule-based provisioning policies that adds one application VM when the average application tier utilization exceeds $y\%$ for successive intervals and removes one application VM when the average utilization falls below $x\%$ for successive intervals. In practice, it suffices to consider two successive intervals for the scaling decisions, just as in the case of DC2.

4.1 Comparison of different policies

Figure 5(a) shows our experimental results for DC2 under the Bursty trace. The figure shows the monitored (black solid line) and estimated (green line with dots) response time under DC2, along with the response time SLA (dashed line). We only show the response time for the browse requests. We see that the monitored response time under DC2 is below the SLA throughout the experiment. The up and down triangles represent the points in time when a scale-up and scale-down action was triggered, respectively. As mentioned in Section 2, a scaling is triggered based on two successive recommendations from the Kalman filter. Observe that the estimated response time is typically in agreement with the monitored response time. This indicates the accuracy of our Kalman filtering technique. However, there is a difference between the estimated and monitored response time for the first few intervals. This is because it takes some time for the Kalman filter to calibrate its model based on the monitored data, as discussed in Section 3.

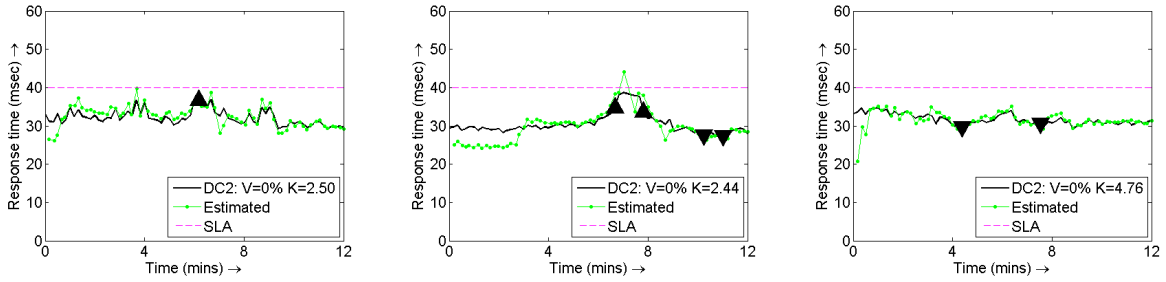
Using the THRES(x,y) policy in practice is tricky since it requires finding the right values for x and y. To find the optimal THRES policy, we start with $x = 20\%$ and $y = 70\%$, and then iterate via trial-and-error till we find the optimal values. Our results indicate that $y = 60\%$ results in the lowest **K** with $\mathbf{V} = 0$. We then experiment with different x values with $y = 60\%$. Based on our results, we conclude that THRES(30,60) is the optimal THRES policy for the Bursty trace.

Table 1 shows the performance of different policies for the Bursty trace. While both DC2 and THRES(30,60) result in zero SLA violations and low resource consumption, THRES requires a lot of experimentation and calibration to achieve the desired performance.

4.2 Comparison under different traces

We now consider the Hill trace and the Rampdown trace. Figures 5(b) and 5(c) show our experimental results for DC2 under these traces. We again see that the (monitored) response time under DC2 is below the SLA throughout the experiment for both traces. It is important to note that we do not change our DC2 algorithm between experiments. DC2 automatically adapts (based on the Kalman filtering technique discussed in Section 3) to the different traces and takes corrective actions to ensure that the SLA is not violated.

Unfortunately, the THRES(30,60) policy is no longer optimal for the Hill or Rampdown traces. For the Hill trace, we find that THRES(30,50) is optimal. This is because the Hill trace exhibits a steep rise in load, requiring more aggressive scaleup. For the Rampdown trace, we find that THRES(40,60) is optimal. This is because the Rampdown traces exhibits a gradually lowering request rate, allowing for more aggressive scaledown. Not using the right THRES policy for each trace can result in expensive SLA violations or increased resource consumption (see Table 1). We thus conclude that *DC2 is more robust to changes in arrival patterns than THRES.*



(a) DC2 under Bursty trace (b) DC2 under Hill trace (c) DC2 under Rampdown trace

Figure 5: Performance of DC2 for all traces.

Trace	Bursty		Hill		Rampdown	
Metric / Policy	V	K	V	K	V	K
THRES(30,60)	**0%**	**2.50**	6.66%	2.56	0%	6.00
THRES(30,50)	0%	2.79	**1.21%**	**2.72**	0%	6.00
THRES(40,60)	2.02%	2.19	15.87%	2.13	**0%**	**4.62**
DC2	**0%**	**2.50**	**0%**	**2.44**	**0%**	**4.76**

Table 1: Comparison of policies for all traces. For each trace, the optimal policies' values are displayed in bold.

5 Related Work

Auto-scaling approaches: Prediction models [6,20,21] use historical data to predict future demand and proactively allocate resources. In most cases, however, some information about the application is required to convert predicted demand into resource requirements. Control-theoretic techniques [6,7,9,13] react to the current system state and adjust the resource allocation accordingly. However, these approaches typically rely on system profiling to convert the system state into scaling actions. Queueing-based models [19,25,26] also require information about the application to make informed scaling decisions. Black-box models do not require information about the application, and instead leverage statistical techniques [16] or machine learning [4] to infer system parameters. DC2 uses a grey-box approach by modeling the system as a queueing network, and then leverages Kalman filtering to infer the parameters of the queueing model. Grey-box approaches typically require less time to converge and infer the system state as opposed to black-box models.

Kalman filtering approaches: An Extended Kalman filter (EKF) based approach was proposed in [31,33] where the system was modeled using a queueing network. The authors in [14] study a three-class, single-tier system and conduct an extensive experimental evaluation of EKF. Recently, [32] applied the EKF approach to track resource usage for a three-class, two-tier system. In this work, we generalize EKF to a three-class, three-tier system, and specifically use EKF for auto-scaling, as opposed to the above works that focus on modeling and offline analysis. The authors in [12] use Kalman filtering for allocating CPU resources to VMs by modeling the application performance as a function of CPU utilization. Our work leverages application-level metrics in addition to resource usage metrics, and employs queueing-theoretic models to capture the interaction between the resources, application load, and performance.

Rule-based approaches: Auto-scaling features are now offered by almost every major CSP including Amazon (AWS) [3], VMware [27], Windows Azure [30], and Google [10]. However, to the best of our knowledge, existing CSP-offered auto-scaling solutions are rule-based and typically require the user to specify the threshold values on the resource usage (e.g., CPU, memory, storage). Further, such rule-based approaches have to be tuned to the specific demand pattern for best results, as demonstrated by the THRES policy in Section 4. By contrast, DC2 does *not* require the user to specify scaling rules.

6 Conclusion

In this paper we present the design and implementation of a new cloud service, Dependable Compute Cloud (DC2), that automatically scales user applications in a cost-effective manner to provide performance guarantees. Since cloud service providers (CSPs) do not have complete control and visibility of a user's cloud deployment, we designed DC2 to be application-agnostic. In particular, unlike most of the existing auto-scaling research, DC2 does *not* require any offline profiling or application benchmarking. Instead, DC2 employs a Kalman filtering technique in combination with a queueing theoretic model to proactively determine the right scaling actions for an application deployed in the cloud. An overarching goal behind the conception of DC2 is to make a case for a CSP-offered auto-scaling service that is superior to existing rule-based offerings. Since the cloud is marketed as a platform designed for all levels of tenants, we believe that users who do not have expert knowledge in performance modeling and system optimization should be able to easily scale their applications. Existing auto-scaling research has ignored this segment of users. We hope that DC2 motivates further research in the area of easy-to-use, application-agnostic auto-scaling.

References

[1] libvirt virtualization API. http://libvirt.org.

[2] RUBiS: Rice University Bidding System. http://rubis.ow2.org.

[3] AMAZON INC. Amazon Auto Scaling. http://aws.amazon.com/autoscaling.

[4] DELIMITROU, C., AND KOZYRAKIS, C. Quasar: Resource-efficient and QoS-aware Cluster Management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (Salt Lake City, UT, USA, 2014), pp. 127–144.

[5] DUBE, P., YU, H., ZHANG, L., AND MOREIRA, J. Performance evaluation of a commercial application, trade, in scale-out environments. In *Proceedings of the 15th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems* (2007), pp. 252–259.

[6] GANDHI, A., CHEN, Y., GMACH, D., ARLITT, M., AND MARWAH, M. Minimizing Data Center SLA Violations and Power Consumption via Hybrid Resource Provisioning. In *Proceedings of the 2011 International Green Computing Conference* (Orlando, FL, USA, 2011), pp. 49–56.

[7] GANDHI, A., HARCHOL-BALTER, M., RAGHUNATHAN, R., AND KOZUCH, M. AutoScale: Dynamic, Robust Capacity Management for Multi-Tier Data Centers. *Transactions on Computer Systems 30* (2012).

[8] GARTNER, INC. Gartner's Advice for CSPs Becoming Cloud Service Providers. https://www.gartner.com/doc/2155315, 2012.

[9] GHANBARI, H., SIMMONS, B., LITOIU, M., BARNA, C., AND ISZLAI, G. Optimal Autoscaling in a IaaS Cloud. In *Proceedings of the 9th International Conference on Autonomic Computing* (San Jose, CA, USA, 2012), pp. 173–178.

[10] GOOGLE CLOUD PLATFORM. Auto Scaling on the Google Cloud Platform. http://cloud.google.com/resources/articles/auto-scaling-on-the-google-cloud-platform.

[11] ITA. The Internet Traffic Archives: WorldCup98. http://ita.ee.lbl.gov/html/contrib/WorldCup.html.

[12] KALYVIANAKI, E., CHARALAMBOUS, T., AND HAND, S. Self-adaptive and Self-configured CPU Resource Provisioning for Virtualized Servers Using Kalman Filters. In *Proceedings of the 6th International Conference on Autonomic Computing* (Barcelona, Spain, 2009), pp. 117–126.

[13] KRIOUKOV, A., MOHAN, P., ALSPAUGH, S., KEYS, L., CULLER, D., AND KATZ, R. NapSAC: Design and implementation of a power-proportional web cluster. In *Proceedings of the 1st ACM SIGCOMM Workshop on Green Networking* (New Delhi, India, 2010), pp. 15–22.

[14] KUMAR, D., TANTAWI, A., AND ZHANG, L. Estimating model parameters of adaptive software systems in real-time. In *Run-time Models for Self-managing Systems and Applications*, D. Ardagna and L. Zhang, Eds., Autonomic Systems. Springer Basel, 2010, pp. 45–71.

[15] LORIDO-BOTRÁN, T., MIGUEL-ALONSO, J., AND LOZANO, J. A. Auto-scaling Techniques for Elastic Applications in Cloud Environments. Tech. Rep. EHU-KAT-IK-09-12, University of the Basque Country, 2012.

[16] NGUYEN, H., SHEN, Z., GU, X., SUBBIAH, S., AND WILKES, J. AGILE: Elastic Distributed Resource Scaling for Infrastructure-as-a-Service. In *Proceedings of the 10th International Conference on Autonomic Computing* (San Jose, CA, USA, 2013), pp. 69–82.

[17] OPENSTACK.ORG. OpenStack Open Source Cloud Computing Software. http://www.openstack.org.

[18] OPSCODE INC. Chef. http://www.opscode.com/chef.

[19] PACIFICI, G., SPREITZER, M., TANTAWI, A., AND YOUSSEF, A. Performance management for cluster-based web services. *Selected Areas in Communications, IEEE Journal on 23*, 12 (2005), 2333–2343.

[20] ROY, N., DUBEY, A., AND GOKHALE, A. Efficient Autoscaling in the Cloud Using Predictive Models for Workload Forecasting. In *IEEE International Conference on Cloud Computing* (2011), pp. 500–507.

[21] SHEN, Z., SUBBIAH, S., GU, X., AND WILKES, J. CloudScale: Elastic resource scaling for multi-tenant cloud systems. In *Proceedings of the 2nd ACM Symposium on Cloud Computing* (Cascais, Portugal, 2011), pp. 1–14.

[22] SIMON, D. *Optimal State Estimation: Kalman, H Infinity, and Nonlinear Approaches*. John Wiley & Sons, 2006.

[23] SOFTLAYER TECHNOLOGIES, INC. http://www.softlayer.com.

[24] SP HOME RUN INC. Cloud Service Provider (CSP) and Inbound Marketing. http://www.sphomerun.com/cloud-service-provider-csp, 2013.

[25] URGAONKAR, B., AND CHANDRA, A. Dynamic Provisioning of Multi-tier Internet Applications. In *Proceedings of the 2nd International Conference on Automatic Computing* (Seattle, WA, USA, 2005), pp. 217–228.

[26] URGAONKAR, B., PACIFICI, G., SHENOY, P., SPREITZER, M., AND TANTAWI, A. An analytical model for multi-tier internet services and its applications. In *Proceedings of the 2005 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (Banff, Alberta, Canada, 2005), pp. 291–302.

[27] VMWARE, INC. VMware vFabric AppInsight. `http://pubs.vmware.com/appinsight-5/index.jsp`.

[28] WALRAND, J. *An Introduction to Queueing Networks*. Prentice Hall, 1988.

[29] WAND NETWORK RESEARCH GROUP. WITS: Waikato Internet Traffic Storage. `http://www.wand.net.nz/wits/index.php`.

[30] WINDOWSAZURE. How to Scale an Application. `http://www.windowsazure.com/en-us/manage/services/cloud-services/how-to-scale-a-cloud-service`.

[31] WOODSIDE, M., ZHENG, T., AND LITOIU, M. Service system resource management based on a tracked layered performance model. In *Proceedings of the 2006 IEEE International Conference on Autonomic Computing* (Washington, DC, USA, 2006), IEEE Computer Society, pp. 175–184.

[32] ZHANG, L., MENG, X., MENG, S., AND TAN, J. K-scope: Online performance tracking for dynamic cloud applications. In *Proceedings of the 10th International Conference on Autonomic Computing* (2013), pp. 29–32.

[33] ZHENG, T., WOODSIDE, M., AND LITOIU, M. Performance model estimation and tracking using optimal filters. *Software Engineering, IEEE Transactions on 34*, 3 (2008), 391–406.

Exploring Graph Analytics for Cloud Troubleshooting

Chengwei Wang, Karsten Schwan, Brian Laub, Mukil Kesavan, Ada Gavrilovska
College of Computing, Georgia Institute of Technology
{*flinter, karsten.schwan, brian.laub, mukilk, ada*}*@cc.gatech.edu*

Abstract

We propose VFocus, a platform which uses streaming graph analytics to narrow down the search space for troubleshooting and management in large scale data centers. This paper describes useful guidance operations which are realized with graph analytics and validated with representative use cases. The first case is based on real data center traces to measure the performance of troubleshooting operations supported by VFocus. In the second use case, the utility of VFocus is demonstrated by detecting data hotspots in a big data stream processing application. Experimental results show that VFocus guidance operations can troubleshoot Virtual Machine (VM) migration failures with accuracy of 83% and with delays of only hundreds of milliseconds when tracking migrations on 256 servers hosing 1024 VMs. Such successes are achieved with negligible runtime overheads and low perturbation for applications, in comparison to brute-force approaches.

1 Introduction

Troubleshooting large scale distributed systems/applications in data centers is important and challenging. There can be millions of entities (e.g., cores) running a large variety of applications across complex software stacks (e.g., hypervisors, guest VMs, middleware). With such complexity, variety, and large numbers, brute-force approaches logging all possible performance-relevant events, at all levels of abstraction, and for all entities, do not scale.

An emerging research area is to build systems that combine online monitoring with online data analytics – Monalytics [12]. As a representative solution, the VScope system developed in our previous work offers useful approaches to capture the 'most relevant' performance data about performance issues observed in large-scale data center applications [17]. Specifically, VScope uses lightweight, continuous, and global monitoring to detect performance anomalies, then 'zooms in' on those anomalies, by dynamically deploying more detailed methods for data capture and online data analysis. Results obtained from representative data center applications demonstrate clearly the advantages of VScope, compared in performance and accuracy to logging approaches capturing all performance-relevant events. Lacking from VScope, however, were the techniques needed to 'guide' the analyses being performed on captured monitoring information. VScope was not able to capture important relationships among the entities being monitored, nor did it provide structured ways to then analyze those relationships. Such a 'guidance' framework for monitoring data analysis is the key contribution of the VFocus system presented in this paper, which offers the following novel functionalities.

1. Interaction snapshot as a general representation for interactions among the software/hardware entities that present in data centers,

2. Streaming graph analytics as the online methods used to evaluate the continually evolving interactions represented in dynamically constructed snapshots.

With VFocus, data center administrators can create diverse interaction tracking methods, and the resulting 'guidance' methods can efficiently troubleshoot performance problems, by 'zooming-in' with both the data collection being performed and the analyses applied to such data. VFocus makes following novel contributions:

1. Graph-based analytics framework: a platform on which interaction graphs are constructed via online monitoring and analyzed using graph analytics, with the choice of graphs and analytics depending on the interactions that users wish to track.

2. Guidance operations using graph analytics: using VFocus guidance operations including *sort*, *group*, and *explore*, users can build various troubleshooting methods to reduce the search space for troubleshooting.

3. Validation with data center traces and use cases:

VFocus and its functionalities are evaluated with use cases to identify VM migration failures and to diagnose 'data hotspots' in the HBase key value store. Experimental results show that VFocus can troubleshoot VM migration failures with an accuracy of 83%, and with low overheads and interference with the applications.

2 VFocus Design and Implementation

2.1 System Overview

The VFocus system realizes the two-phase guidance mechanism illustrated in Figure 1. In the *snapshot construction* phase, VFocus collects metric data from monitored entities and builds interaction snapshots, realtime graphs in which vertices are software/hardware components and edges are interactions among the components. Interaction snapshots are updated continuously, to reflect the latest activities in the system. The second phase is *snapshot analysis*, for which VFocus exposes three primitive operations: *sort, group, and explore*, explained in more detail in Section 2.2. Data center operators use those operations to track and analyze interactions in the system, and to identify the entities to the performance issue being observed.

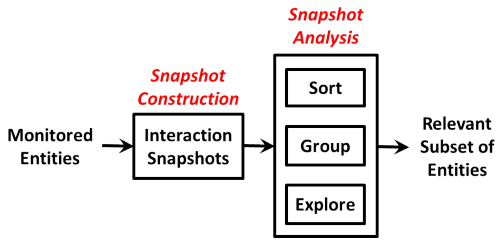


Figure 1: Guidance Framework

VFocus leverages the *DPG (Distributed Processing Graphs)* introduced in [17]. A *DPG* is an overlay network consisting of processing nodes called *VNodes* that each collects and analyzes monitoring data at realtime, in a streaming manner. *DPG* topologies can vary, to match monitoring needs (e.g., scale [18], and they can be deployed at runtime and on-demand, as described in more detail in [17]). On this basis, the system architecture of VFocus depicted in Figure 2 enables users to interact with it via two interfaces, an *analysis console* and an *archive console*, both of which are integrated in the *VMaster*, a controller process for *DPG* manipulations. The *analysis console* takes users' input as operation commands to analyze interaction snapshots and provides users with guidance results. The *archive console* is used to manage snapshots persisted in the VFocus backend store. In some troubleshooting scenarios, e.g., when analyzing snapshots spanning a time duration too long to fit all snapshots in memory, the realtime analysis may query the archiving system for history data.

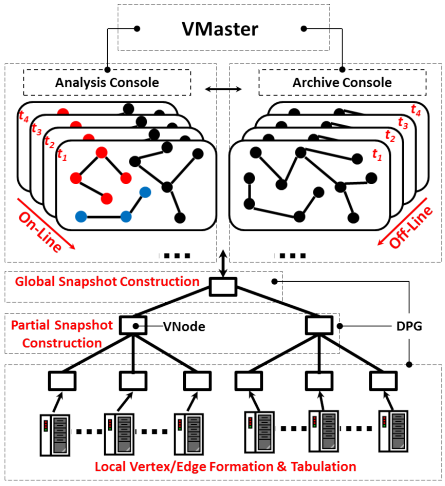


Figure 2: VFocus Architecture

The system operates as follows. The VMaster starts the guidance process by deploying a hierarchical *DPG* on the monitored machines[1]. Raw monitoring data collected periodically by the DPG's leaves are processed to extract vertices and edges for forwarding to parent nodes. Parents aggregate vertices and edges into edge lists that represent partial snapshots of the machines monitored by those parents' leaves. Partial snapshots are passed up the tree to the root for aggregation into a global snapshot.

2.2 Guidance Operations

The primitive graph analysis functions provided by VFocus are listed in Table 1. There are functions to calculate the basic properties of a graph, such as the degree of some specific vertex or the number of vertices and edges in the snapshot. For instance, the degree of a vertex is used in Section 3.1 to find the candidate servers that are most likely to have VM migration failures. There are also functions for tracking the relationships among vertices, e.g., *Search Neighbor* and *Clique Analysis*. Section 3.2 outlines a use case in which neighbor analysis is used to determine the HBase region server with a data hotspot. As stated earlier, all such analysis functions are executed on *VNodes* at runtime, at the same time as monitoring metrics are being collected. This results in repeated incremental snapshots suitable for rapid analysis and subsequent problem troubleshooting. Finally, some analytics are run concurrently and in a distributed manner. For example, an aggregation tree is used to implement the

[1] The *DPG* runtime is installed on those machines.

Table 1: Functions.CC(Connected Components), C(Centralized), H(Hierarchical)

Graph Function	Description	Mode
Search Vertex/Edge	Search vertex/edge	C
Search Neighbor	(In)direct neighbors	C
Count Vertex	Get number of vertices	H
Count Edge	Get number of edges	H
Get Degree	Get degree of a vertex	H
Get Attribute	Get vertex/edge attr.	C
Clustering/Clique/CC	Grouping vertices	C

Table 2: Operations.CC(Connected Components)

Operation	Basic Option	Supporting Fun.
Sort	Vertices by degree	Get Degree
	Edges/Vertices by attr.	Get Vertex/Edge
Explore	(In)direct neighbors	Search Neighbors
	Vertices/Edges by attr.	Search Vertex/Edge
Group	Vertices by attr.	Clustering
	Vertices by connection	Clique/CC

Count Degree function. Others operate in a centralized manner, e.g. the *Search Neighbor* function.

Analysis results are used by guidance operations to narrow down the search space of vertices and edges. Toward this end, VFocus provides three primitive guidance operations listed in Table 2: *sort*, *group*, and *explore*. They are implemented with different snapshot analysis functions listed in Table 1. Data center operators access and execute the operations in analysis console as command lines.

The *Sort* operation, supported by the *Get Degree* and *Get Vertex/Edge* functions, orders vertices or edges by their attributes (or by other aggregated properties like 'degree') and returns the top entities in the sorted list.

The *Explore* operation, supported by the *Search Vertex/Edge* and the *Search Neighbors* functions, can search the neighbors of some specified vertex at different distances (measured by the number of hops) or it can search the vertices/edges with some specified attribute. An example of its use appears in Section 3.2, where the *explore* operation is used to track the data connectivities between the HBase region servers and the HBase clients, in order to troubleshoot the data hotspot issue.

The *Group* operation places closely related vertices into a shared group, returning the group as the entity subsequently manipulated. 'Grouping' style analysis has also been used in previous troubleshooting research [10], e.g., where VMs are clustered by their similarity in terms of their computation behavior. One such similarity is a group of VMs are interacting frequently, a fact that can give rise to optimizations in which the migration of one such VM triggers the migrations of others, to avoid unnecessarily high levels of machine cross-talk in the data center. Finally, the *Group* operation has two options: grouping by attributes using clustering algorithm, or grouping by the connections between vertices, using clique analysis and connected components, etc..

3 Use Cases

3.1 VM Migration Analysis

In virtualized data centers, VM migration moves a VM running on a source host to a different destination host. Performed for consolidation and resource management purposes, migration is a 'heavyweight' management operation both in terms of its effect on the VM itself, the network and machine resources consumed. Migration failures, therefore, have substantial performance implications. A cause for migration failure is the inappropriate choice of destination machines, one reason being an *overloaded host* with insufficient capacity to rapidly complete the migration. Migration methods, therefore, will not select targets with high internal workloads, as indicated by their CPU or memory utilization. A remaining issue, however, is the external workload surrounding target hosts, an example being insufficient current network capacity to move the VM's substantial internal states quickly to the target machine.

VFocus can assist in VM migration management (1) via online tracking of host interactions, scaling to hundreds of server systems, and then (2) by using guidance operations to identify *overloaded hosts* not only based on their internal workloads but also based on their interactions with other hosts. We demonstrate the utility of VFocus for identifying overloaded hosts by 'replaying' a VM management operation trace recorded from a virtualized data center. The trace is collected from 256 servers in the Techway virtualized data center at Georgia Tech, a 'green computing' facility run jointly by the CERCS and CEETHERM research centers. The servers host a total of 1024 VMs running enterprise workloads, including **(1) Nutch:** a data analytics benchmark, **(2) Voldemort:** a key-value store [16] with *YCSB* [9] as load generator, **(3) Cloudstone:** a cloud web-serving benchmark, **(4) Linpack:** a high performance computing workload. The VM migration traces are collected from October 05, 2012 8:23:15 AM to October 12, 2012 12:04:20 PM. These 172 hours of data document 31790 successful and 2845 failed migrations.

VFocus is used as follows. A centralized *DPG* with one master *VNode* and 8 slave *VNodes* is driven by attaching to each slave log record generators. Each generator runs 1/8th of the log describing successful migrations, injecting log records into the slave *VNodes*. These leaf VNodes, then, periodically process log records to gener-

ate local snapshots and perform local snapshot analysis, the results of which are then sent to the master *VNode* to create global snapshots for guidance operations. We use an adjustable sliding window storing 1000 migrations in these experimental evaluations, in order to make statistically sound predictions, and slide the window after every new migration. In each snapshot, the vertices are the hosts, and the edges are migrations from source to destination host. There are approximately 256 vertices and 1000 edges in every snapshot, and there can be multiple edges between two vertices due to multiple migrations between two hosts in the same time period.

The troubleshooting process has following steps. In the first step, we use *group* operation to divide the global interaction snapshot, which consist of partial snapshots, into sub-graphs in each of which there is at least one path between two vertices. The hypothesis behind using this operation is that the load of a host does not come from the other host which it does not interact with.

When there is a new migration observed on a host, VFocus first checks which group this host belongs to, and then uses the *sort* operation on degrees of all the vertices (hosts) in that group and ranks the hosts in a descending order. The reason behind this operation is that the top hosts on the list are having/or had most migrations, and hence are more likely to have resource scarcity issue when new migration requests take place. In the trace, any two migrations happen at different time, therefore there is only one group chosen when the *sort* operation is executed. The *sort* operation will provide the top n hosts as candidates which may potentially have VM migration failures in the future. n is tunable and in our experiment, we find that n=45 gives us a good performance.

After replaying the trace in VFocus, it produces a series of candidate lists. To validate the guidance methodology, we match the list to the respective error log record at each failure point. If the actual host with migration failure is in the list, then we consider VFocus as a 'hit' as the predicted list contains the actual failure node, otherwise we consider it as a 'miss'. In this paper we only study the failures due to overloaded hosts which are indicated as 'operation timed out' in the failure log.

Table 3 shows the performance of VFocus guidance. We can see in the table that, the overloaded host errors have a significant percentage of all the errors (over 50%). As a guidance approach, VFocus reduces the search space from 256 servers to 45 servers, with an overall hit rate of over 83%. We further study the 'position' of a hit which is essentially the actual number of candidates needed to be checked before reaching the real failure node. The distribution of hit position is shown in Figure 3. About 70% of hits happen within 30 hosts and nearly 40% is within 15 hosts, which means for the majority of time, size of search space can be further reduced from 45 to 30.

Table 3: VFocus Guidance Accuracy

# Overload Errors	# Overload Found	Hit Rate
1441 (51% of all errors)	1195	83%

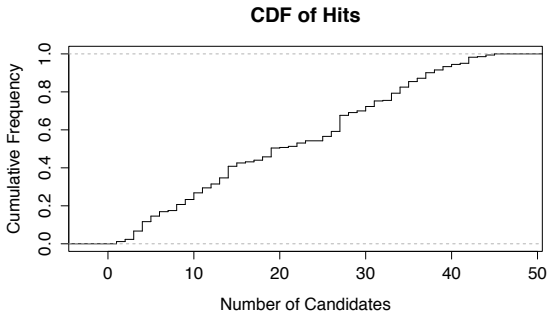


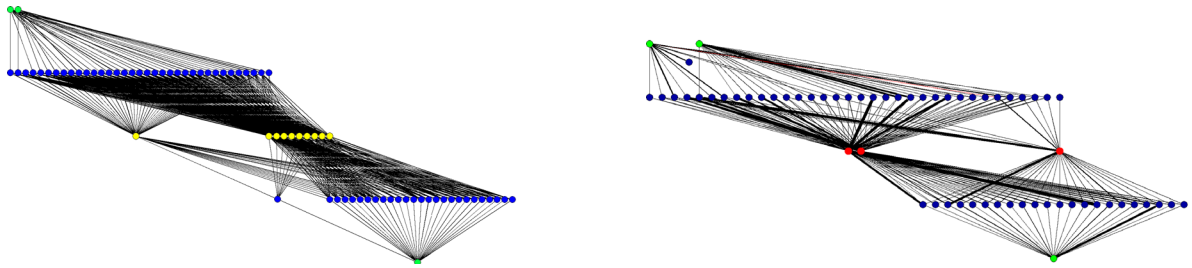
Figure 3: Distribution of Hit Positions

3.2 On-line Data Hotspot Analysis

HBase, a distributed key-value store system, is widely used as an infrastructure to support big data applications. HBase consists of multiple region servers which serve the read/write requests (mostly through memory cache) concurrently to reduce the latency and increase the throughput. One of the annoying performance issues in this platform is that some region servers in HBase become data hotspots which receive significantly more read/write requests than other region servers. As a majority portion of requests go to the hotspot region servers, their cache in main memory is filled up quickly. Those region servers then become the bottleneck and bring down the overall read/write performance of the HBase. One of the common reasons for the hotspot issue is the imbalanced accessing pattern on the row key space. Which region server each read/write request goes to is decided by the row key value in the request, and in HBase the total row key space is evenly divided, as equal regions managed by region servers. Therefore, if the row keys in the requests, either from one application or from multiple applications, concentrate on some of the key regions, the according region servers become hotspots.

VFocus addresses this challenge in two aspects. First the interaction snapshots can track the runtime communications among HBase region servers online. Secondly, the hotspots can then be easily spotted by using guidance operations.

We validate VFocus' effectiveness in our virtualized data center supported by OpenStack. We started over 100 VMs on 30 physical servers running Xen hypervisor. We



(a) Normal Interactions

(b) Interactions with Hotspots

Figure 4: Interaction Snapshots in Hotspot Analysis

deployed two applications (1) GTStream [2], a big data streaming processing platform using FlumeNG [1], (2) traditional read/write clients using YCSB [9], a widely used cloud benchmark. The two applications share the same HBase infrastructure though they access different tables in the HBase. In GTStream application, each Flume Agent writes at a rate of 100 requests per second to HBase accessing a table named 'test'. We use a mix of 50/50 reads and writes workload in YCSB clients and each client send 100 requests per second to HBase accessing a table named 'usertable'. Both tables' key regions are divided evenly across all the region servers. We deploy VFocus on all the VMs and create a hierarchical *DPG* with 10 parent *VNodes* on 10 dedicated VMs. We use *Libpcap* [3] to sniff the network traffic on every VM.

We showcase VFocus' guidance functionalities in two scenarios. The first scenario represents a normal workload of HBase where there is no hotspot. In this scenario, both the YCSB clients and Flume agents assign row key value in a balanced way. For every request, YCSB creates a key value as a combination of a string and a hash value randomly selected from the key value space. In Flume, a key value is assigned as a combination of an arbitrary web url which the flume agent is processing and a timestamp. Figure 4(a) illustrates one of the snapshots constructed by VFocus online. Yellow dots are all the region servers. Green dots represent zookeeper servers. Blue dots represent HBase clients including Flume agent and YCSB clients. The edges among vertices are the network communications and the thickness of an edge indicates the traffic intensity between the two VMs. The snapshot tracks the network interactions among zookeepers, HBase clients (YCSB clients and Flume agents), and region servers. By using *explore* operation on vertices representing region servers to find its neighbors, we can easily find that all the region servers are being accessed by all the YCSB clients and Flume agents with approximately equal traffic between any two edges.

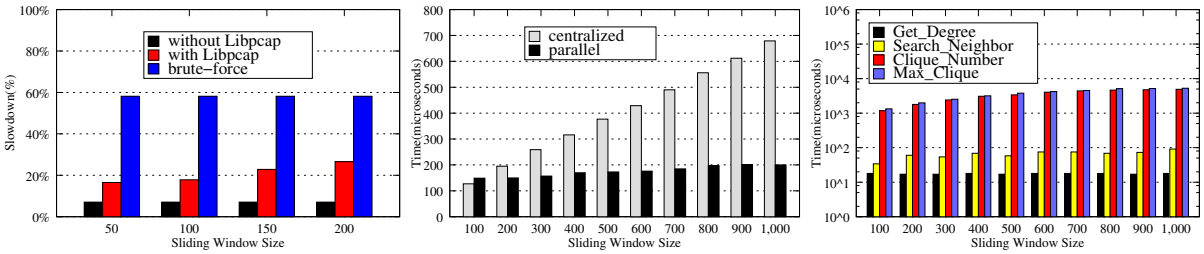
In the second scenario, both the YCSB clients and Flume agents assign row key value as a combination of a fixed string and current timestamp, which monotonically increases instead of randomly distributed in the key space. Therefore, the key values will condense in a limited range and the requests from both YCSB and Flume will go to a subset of region servers, making them hotspots. Figure 4(b) illustrates one of the snapshots constructed in this hotspot scenario, where red dots are hotspot region servers. Besides tracking the interactions among VMs according to their roles in the applications, running *explore* operation on all the region servers figures that only three region servers are communicating with HBase clients, indicating the three hotspots.

4 Performance Evaluation

4.1 VFocus Overhead

To test the overhead of VFocus on a VM, we run the network interaction tracking used in Section 3.2 on VFocus and change the durations of sniffing. We measure the VM's CPU and Memory utilization increase as the overheads.The overhead for VFocus is within 2% in CPU utilization while the memory consumption is no more than 0.2%. The CPU utilizations increase slightly as the duration increases as the network sniffing consumes more CPU cycles as it continuously runs for a longer time.

The interferences of VFocus and brute-force approach to the application are shown in Figure 5(a). It shows that as the monitoring duration increases, VFocus' interference to the application increases, while the interferences are within 30% when the duration increases from 50 seconds to 200 seconds. By contrast, the always-on brute-force approach has a considerably higher interference at around 58%. To further study the breakdown of the overheads, we turned off the *Libpcap-based* data collection function on all the Flume agents, and instead stored network connection metrics, which are pre-recorded in file using Libpcap, in memory. VFocus reads the memory and processes data in the same way as we did when using Libpcap. The black bar in Figure 5(a) indicates that the VFocus itself does not play the major role in the total overheads because it only incurs 7.06% slowdown,



(a) Interference to Application (b) Graph Construction Time (c) Graph Analysis Performance

Figure 5: VFocus Performance Results

compared to over 20% slowdown when *Libpcap-based* monitoring is enabled.

4.2 Snapshot Construction Performance

We test the snapshot construction performance by generating VM migration snapshots in VM migration use case with different sliding window sizes, and compare two construction strategies, centralized and parallel. In the centralized strategy, we send all the migration data to a central node and generate a global snapshot; while in the parallel strategy, we create a snapshot in a distributed way by which 8 *VNodes* create partial snapshots which are aggregated at the root *VNode*. Each snapshot has up to 256 vertices and 1000 edges. As shown in Figure 5(b), parallel construction outperforms centralized strategy and as the size of sliding window increases the construction time increases as well. However the maximum construction time is within 700 milliseconds, and the parallel construction strategy increases much slower than the centralized strategy does. Validated by real data center monitoring traces, the results show than the snapshot construction in VFocus is fast, hence VFocus is capable of tracking online, runtime interactions responsively. Parallel construction strategy used by VFocus is more scalable and faster than the centralized strategy.

4.3 Graph Analysis Performance

We evaluate the VFocus' performance on snapshot analysis by measuring the computation time of analysis listed in Table 1. Clique analysis has two functions (1) *Clique_Number* function that counts the number of cliques in the snapshot and (2) *Max_Clique* function that yields the largest clique in the snapshot. The monitoring data is the trace used in Section 3.1. We measure computation time at different sliding window sizes.

As shown in Figure 5(c), the computation time is within 10000 microseconds when the sliding window size changes from 100 to 1000, which means for VM migration use case, the graph analysis can be conducted in a timely manner. As the sliding window enlarges, the time just increases slightly because all the computation are processed in memory, which make it suitable for real-time graph analysis. Among different analysis functions, the two clique analyses have longer computation time because they have higher computation complexities than degree analysis and neighbor analysis.

5 Related Work

VFocus is similar to previous research on *dependency inference*.[8, 5, 4] use network traffic and signal processing methods to infer dependencies. [7] leverages application-level knowledge along with network traffic information, to infer dependencies. The fundamental difference between VFocus and dependency inference is that the latter does not provide a general framework for graph abstraction and analysis on dependencies. In *request path diagnosis* research, [13, 6] highlight performance differences between application activities by comparing their request execution paths. VFocus is different because it focuses on creating continuous snapshots of the interactions on-line while request path diagnosis are off-line approaches. VScope [17] is a flexible middleware which can dynamically deploy monitoring and analysis functions on any monitored entities at any time. VFocus leverages VScope's flexible architecture but it is a graph analysis system aiming to track and analyze the interactions among entities in data centers on-line at real-time. VScope has built-in guidance mechanism using ad-hoc approaches, which lacks extensibility and generality while VFocus provides a guidance framework and primitive guidance operations by which different troubleshooting approaches can be implemented.

References

[1] Apache flume. `http://flume.apache.org/`.

[2] Gtstream. `https://github.com/chengweiwang/GTStream`.

[3] Libpcap. `http://www.tcpdump.org/`.

[4] S. Agarwala, F. Alegre, K. Schwan, and J. Mehalingham. E2EProf: Automated End-to-End Performance Management for Enterprise Systems. In *IEEE Conference on Dependable Systems and Networks (DSN)*, 2007.

[5] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed system of black boxes. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[6] M. Attariyan, M. Chow, and J. Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[7] P. Bahl, R. Chandra, A. G. Greenberg, S. Kandula, D. A. Maltz, and M. Zhang. Towards Highly Reliable Enterprise Network Services via Inference of Multi-level Dependencies. In *ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, 2007.

[8] X. Chen, M. Zhang, Z. M. Mao, and P. Bahl. Automating Network Application Dependency Discovery: Experiences, Limitations, and New Solutions. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[9] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, 2010.

[10] L. Hu, K. Schwan, A. Gulati, J. Zhang, and C. Wang. Net-Cohort: Detecting and Managing VM Ensembles in Virtualized Data Centers. In *ACM International Conference on Automatic Computing (ICAC)*, 2012.

[11] M. Kesavan, A. Gavrilovska, and K. Schwan. Elastic Resource Allocation in Datacenters: Gremlins in the Management Plane. In *VMware Technical Journal*, 2012.

[12] M. Kutare, G. Eisenhauer, C. Wang, K. Schwan, V. Talwar, and M. Wolf. Monalytics: Online Monitoring and Analytics for Managing Large Scale Data Centers. In *ACM International Conference on Automatic Computing (ICAC)*, 2010.

[13] R. R. Sambasivan, A. X. Zheng, M. De Rosa, E. Krevat, S. Whitman, M. Stroucken, W. Wang, L. Xu, and G. R. Ganger. Diagnosing Performance Changes by Comparing Request Flows. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[14] B. C. Tak, C. Tang, C. Zhang, S. Govindan, B. Urgaonkar, and R. N. Chang. vpath: precise discovery of request processing paths from black-box observations of thread and network activities. In *USENIX Annual Technical Conference (ATC)*, 2009.

[15] R. Van Renesse, K. P. Birman, and W. Vogels. Astrolabe: A Robust and Scalable Technology for Distributed System Monitoring, Management, and Data Mining. *ACM Transactions on Computer Systems*, 2003.

[16] Voldemort. Project voldemort - a distributed database. http://project-voldemort.com/.

[17] C. Wang, I. A. Rayan, G. Eisenhauer, K. Schwan, V. Talwar, M. Wolf, and C. Huneycutt. VScope: Middleware for Troubleshooting Time-Sensitive Data Center Applications. In *ACM/IFIP/USENIX International Conference on Middleware (Middleware)*, 2012.

[18] C. Wang, K. Schwan, V. Talwar, G. Eisenhauer, L. Hu, and M. Wolf. A Flexible Architecture Integrating Monitoring and Analytics for Managing Large-Scale Data Centers. In *ACM International Conference on Automatic Computing (ICAC)*, 2011.

[19] P. Yalagandula and M. Dahlin. A Scalable Distributed Information Management System. In *ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, 2004.

[20] Z. Zhang, J. Zhan, Y. Li, L. Wang, D. Meng, and B. Sang. Precise Request Tracing and Performance Debugging for Multi-Tier Services of Black Boxes. In *IEEE Conference on Dependable Systems and Networks (DSN)*, 2009.

Inferring Origin Flow Patterns in Wi-Fi with Deep Learning

Youngjune L. Gwon
Harvard University

H. T. Kung
Harvard University

Abstract

We present a novel application of deep learning in networking. The envisioned system can learn the original flow characteristics such as a burst size and inter-burst gaps conceived at the source from packet sampling done at a receiver Wi-Fi node. This problem is challenging because CSMA introduces complex, irregular alterations to the origin pattern of the flow in the presence of competing flows. Our approach is semi-supervised learning. We first work through multiple layers of feature extraction and subsampling from unlabeled flow measurements. We use a feature extractor based on sparse coding and dictionary learning, and our subsampler performs overlapping max pooling. Given the layers of learned feature mapping, we train SVM classifiers with deep feature representation resulted at the top layer. The proposed scheme has been evaluated empirically in a custom wireless simulator and OPNET. The results are promising that we achieve superior classification performance over ARMAX, Naïve Bayes classifiers, and Gaussian mixture models optimized by the EM algorithm.

1 Introduction

Machine learning plays an increasingly important role in the complex, data-intensive tasks required by today's sensing and computing systems. A flow is a sequence of data packets sharing the same context (*e.g.*, TCP connection, media stream) sent from a source to its destination. Accurate knowledge about the flow characteristics such as a burst size (in number of packets or bytes) and inter-burst gap, as were originated at the source, can be used beneficially to manage scarce networking resources. One motivating example would be software-defined networking (SDN) [3], which can leverage detailed flow knowledge to program routers and access points (APs) for scheduling a congested data traffic or mitigating wireless interferences more intelligently.

This paper describes our first work in developing inference schemes to learn the original properties of a flow from packet sampling at a receiver that is not necessarily the destination of the flow. We focus on the case where the source and the receiver are Wi-Fi nodes, and there are other Wi-Fi nodes that transmit their own flows. In particular, the receiver for our case is a network node such as a Wi-Fi AP that forwards or broadcasts packets, being a spot of aggregating different flows. The key challenge is how to unravel the work of CSMA that introduces a complicated mixture of competing flows. We believe that the approach of this paper can be extended for various other wireless and wired networks.

1.1 The Problem

Figure 1 explains our origin flow inference problem. Wi-Fi node A is the source of flow $\mathbf{f}_\mathrm{A}$ transmitted to Node B, a Wi-Fi AP. We denote $\mathbf{x}_{\mathrm{A}|\mathrm{B}}$ a sample of $\mathbf{f}_\mathrm{A}$ measured by B. We use vector notation $\mathbf{f}$ to represent an origin flow pattern over time, and $\mathbf{x}$ its measurement. (Section 2 will explain how we describe patterns of a flow in detail; for now, consider $\mathbf{f}$ and $\mathbf{x}$ finite sequences of numbers.) Notice that there are other Wi-Fi nodes in the channel, namely nodes C, D, E, F, and G, that transmit own flows, creating contentions.

Distributed Coordination Function (DCF) provides the fundamental mechanism to access wireless media for the IEEE 802.11 Wi-Fi [1]. DCF employs Carrier Sense Multiple Access (CSMA) with a random backoff drawn from an exponentially growing window. Mixed with other transmissions, the sample $\mathbf{x}_{\mathrm{A}|\mathrm{B}}$ could hardly preserve the original patterns in $\mathbf{f}_\mathrm{A}$. For example, the received packet burst lengths and gaps between bursts can be altered significantly. The exactness of such alteration is difficult to estimate, but there are both linear (*e.g.*, geometric increase of burst lengths) and nonlinear (*e.g.*, packet loss, retransmission, timeout) distortions. Among all possible causes, the main culprit should be CSMA.

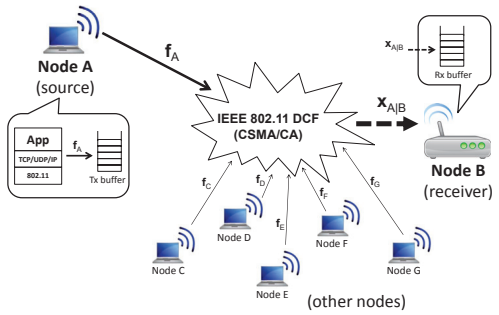


Figure 1: Flow inference problem illustrated

We aim to solve the following.

1. Classify received frame/packet pattern sampled at a receiver Wi-Fi node to the origin flow pattern;
2. Infer the original properties of a flow such as burst sizes and inter-burst gaps originated at the source.

We clarify several points. First, the *origin flow* pattern conveys context of application-level data. This is depicted under node A in Figure 1. $\mathbf{f}_A$ instantiates a pattern of the data formation mostly passed from the application layer (and the lesser from Transport/IP/MAC) to the transmit unit. Thus, our inference problem can lead to important understanding of application context mining. Secondly, our primary work domain is the MAC layer. We deduce observable patterns of a conventional TCP/UDP/IP flow from measuring directly the 802.11 MAC frames over time. Lastly, a sampled flow pattern at best is the origin flow pattern shifted (delayed) in time. For our inference to be effective, it is crucial to learn invariance such as some preserved original burst lengths that can spread widely over time.

1.2 Motivating Applications

Traffic monitoring capabilities are crucial for network management and security. Wireless bandwidth is among the most precious networking resources. Accurate origin flow inference can help derive efficient scheduling for wireless channels. The inferred information can also be used to classify legitimate traffic from malicious attacks.

Programming network nodes. Software-defined networking (SDN) is an emerging paradigm to build highly dynamic networks. Inferred origin traffic information can help program SDN nodes. For example, we can improve transmit-receive scheduling and avoid interferences.

Resource provisioning. The state-of-the-art networks can provision almost all networking resources elastically. The origin traffic inference will reveal the original properties of a flow that resource provisions such as communication bandwidth, flow cache, and compute cycles should strive to satisfy.

Queue management. A network node (*e.g.*, router, AP, switch) can leverage the source sending rates of large flows to manage its receive buffers and scheduling mechanisms dynamically. With knowledge on origin characteristics of a flow, networks can improve overall fairness.

1.3 Our Contributions

The main contribution of our work is to demonstrate the effectiveness of learning algorithms applied to an important networking problem mostly studied under parametric, model-based frameworks. Our approach is semi-supervised learning. We set up and train deep, unsupervised feature learning that constitutes multiple layers of sparse coding and pooling units. Given the learned feature mapping, we train classifiers in supervised learning. We have identified the key attributes for successful learning approaches to enhance our baseline such as forcing incoherency for sparse coding dictionary to extract more discriminative features, dense arrangement of sparse coding units, and max pooling on overlapping intervals. We have also explored and experimented with other learning methods from classical autoregressive time-series prediction and Naïve Bayes classifiers to the EM-optimized Gaussian mixtures. Our evaluation empirically confirms superior performance of the proposed learning methods in recovering the original properties of a flow.

1.4 Related Work

There is considerable work in model-based estimation for origin flow properties. Basu and Mukherjee [5] discuss numerous time-series models for Internet data traffic, including the autoregressive moving average process helpful for some of our formulation in Sections 2 and 3. Claffy *et al.* [9] present one of the earliest work to infer the original packet size distribution of a flow from packet sampling at routers. Duffield *et al.* [12] analyze methods to infer the original frequencies of flow lengths from sparse packet sampling.

The way sparse representations are used in computer vision and pattern recognition has inspired our method. Wright *et al.* [27] have developed a face recognition system that performs classification with sparse representation of features, which is based on a similar idea as ours. The idea of pooling sparse representations of features provides an important primitive to construct higher-level features as studied by Raina *et al.* [22], although pooling techniques date back to Riesenhuber and Poggio [24]. Coates and Ng [10] propose to pool over multiple features for deep learning.

Heisele, Ho, and Poggio [14] explain useful techniques of applying SVM for multi-class classification,

which is inherent in our origin flow pattern inference problem. There are a number of existing techniques to learn incoherent dictionary atoms. Ramirez *et al.* [23], Zhang & Li [28], and Lin *et al.* [18] have proposed similar ideas that force orthonormal dictionary columns as we maximize incoherency among dictionary columns in §5.1. Our idea of accompanying sparsity relaxation (§5.2) for sparse coding with forced incoherent dictionary atoms is new.

1.5 Outline

Section 2 explains the time-series representation and processing of a flow. In Section 3, we explore supervised learning methods for the origin flow inference. Section 4 describes our baseline semi-supervised learning method. We propose several enhancements to the baseline method in Section 5 and evaluate the learning methods with a custom simulator and OPNET in Section 6. The paper concludes in Section 7.

2 Time-series Representation of Flow

The runs-and-gaps model [16] gives a concise way to describe a flow. In Figure 2, characteristic patterns of an example flow are captured by packet *runs* and *gaps* measurable over time. As indicated earlier, we perform our flow measurements directly at the MAC layer rather than at the transport or IP layers by sampling and processing Wi-Fi frames.

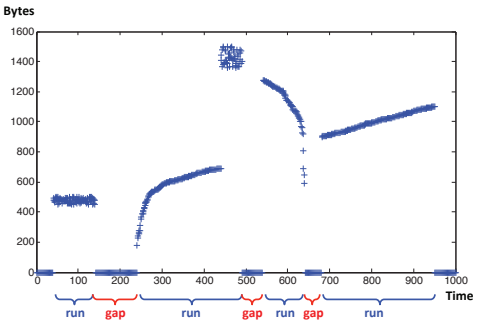


Figure 2: Runs-and-gaps model

Let $\mathbf{w} = [w_1\ w_2\ \ldots\ w_t\ \ldots\ w_N]$ be a vector containing the number of packets in a flow measured over N time intervals. Here, an important parameter is the *unit interval* T_s or *sampling period* during which each element w_t is sampled and recorded. The total measurement time or observation window is $N \times T_s$. Alternatively, we have vector $\mathbf{x} = [x_1\ x_2\ \ldots\ x_t\ \ldots\ x_N]$ for $\mathbf{w}$ where x_t is the corresponding byte count of the total payload at time t. Hence, a zero in $\mathbf{x}$ (or $\mathbf{w}$) indicates a gap. If $w_7 = 3$ and $x_7 = 1{,}492$, we have 3 packets for the flow at $t = 7$, and the sum of the payloads from the 3 packets is 1,492 bytes. We will call either $\mathbf{w}$ or $\mathbf{x}$ a measurable input vector for inference, which contains *extractable* features. While both $\mathbf{w}$ and $\mathbf{x}$ carry unique information, we mainly work with $\mathbf{x}$ throughout this paper.

We designate an origin flow (pattern) with another vector $\mathbf{f}$. Just like $\mathbf{x}$, $\mathbf{f}$ is a sequence of byte counts uniformly sampled, but the difference is that $\mathbf{f}$ reflects the initial pattern (or signature) originated at its source. Note $\mathbf{f} \in \mathbb{R}^M$ whereas $\mathbf{x} \in \mathbb{R}^N$, and M and N are not necessarily equal. We use notation $\mathbf{x}_{i,k}$ to refer kth measurement on flow i since there can be many measurements on $\mathbf{f}_i$. We also use $\mathbf{f}_{i,k}$ to designate the kth instance of origin flow i because there could be many origin patterns, or the pattern can be a stochastic process and changes dynamically over time. In summary, $\mathbf{f}$, $\mathbf{w}$, and $\mathbf{x}$ are all finite time-series representations of a flow.

Consider sampling and processing of three example flows in Fig. 3 at a receiver. The receive buffer first timestamps each arriving data frame and marks with flow ID. At $t = 1$, the received frame for flow 1 contains 2 packets whose payload sizes are 50 and 50 bytes, denoted in $(2, 50/50\text{B})$. At $t = 6$, flow 3 has two received frames. The first frame contains 2 packets with sizes 100 and 400 bytes whereas the second frame contains only one packet with 1,000 bytes. The example results in the following:

1. $\mathbf{w}_1 = [2\ 1\ 2\ 0\ 1\ 2]$, $\mathbf{x}_1 = [100\ 80\ 110\ 0\ 80\ 100]$
2. $\mathbf{w}_2 = [1\ 0\ 1\ 0\ 1\ 0]$, $\mathbf{x}_2 = [600\ 0\ 600\ 0\ 600\ 0]$
3. $\mathbf{w}_3 = [4\ 0\ 0\ 0\ 0\ 3]$, $\mathbf{x}_3 = [1500\ 0\ 0\ 0\ 0\ 1500]$

With $T_s = 10$ msec, each time series take 60 msec to measure. Flow 1 has 133.3 packets/sec, Flow 2 with 50 packets/sec, and Flow 3 with 116.7 packets/sec. In bit rates, they are 62.7, 240, and 400 kbps, respectively.

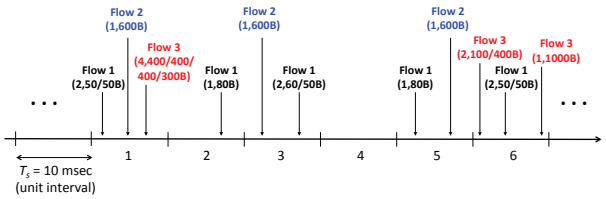


Figure 3: Time-series processing example

3 Origin Flow Inference with Supervised Feature Learning

The core of an inference system comprises a feature extractor (FE) and a classifier (CL) that need to be trained. Figure 4 describes the supervised learning framework. Supervised learning requires a labeled training dataset that consists of training examples $\{\mathbf{x}_1, \ldots, \mathbf{x}_T\}$ with corresponding desired output values (*i.e.*, labels) $\{l_1, \ldots, l_T\}$. There are two mappings, FE : $\mathbf{x} \rightarrow \mathbf{y}$ that

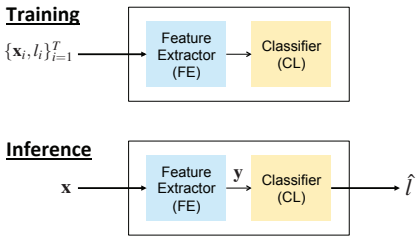


Figure 4: Supervised learning framework

maps an input $\mathbf{x}$ to its feature $\mathbf{y}$ and CL : $\mathbf{y} \rightarrow \hat{l}$ that performs classification on extracted features of the input. The inference system learns the mappings FE and CL from training examples and their labels. Once trained, when an unknown data $\mathbf{x}$ comes in, the system makes an inference by classifying it to a class $\hat{l}$.

Supervised learning for the origin flow inference problem considers a training dataset $\{\mathbf{x}_i, \langle \mathbf{f}_{l_i}, l_i \rangle\}_{i=1}^T$ collected at the Wi-Fi receiver of interest. $\mathbf{x}_i$ is a time-series representation of flow l_i sampled at the receiver. We also make $\mathbf{f}_{l_i}$ available, the corresponding origin flow time-series representation. So, when a measured sample $\mathbf{x}$ is classified as l_j, we can infer the original flow properties from looking up $\mathbf{f}_{l_j}$'s.

We now explore supervised learning methods. We note that most of these methods naturally lead to binary classifiers. Our origin flow inference problem, however, is a multi-class classification. We will revisit this issue in §4.3. For clarity of explanation, this section accompanies binary classification.

Autoregressive moving average with exogenous inputs (ARMAX) [19] is a widely-studied model for linear system identification. ARMAX models the current output of a system with the previous (delayed) output and input values. With ARMAX, we can directly estimate the origin flow time-series $\mathbf{f} = [f_1\ f_2\ \ldots\ f_{t-1}\ f_t]$ from the measurement $\mathbf{x} = [x_0\ x_1\ \ldots\ x_{t-2}\ x_{t-1}]$ in a linear difference equation: $f_t + a_1 f_{t-1} + \ldots + a_n f_{t-n} = b_1 x_{t-1} + \ldots + b_m x_{t-m} + \varepsilon$. Note that ε gives the model error, which itself can be written elaborately over time, *i.e.*, $c_1\varepsilon_{t-1} + \ldots + c_m\varepsilon_{t-m}$. The ARMAX matrix form is

$$\underbrace{\begin{bmatrix} f_t \\ f_{t-1} \\ \vdots \\ f_1 \end{bmatrix}}_{\mathbf{f}} = \underbrace{\begin{bmatrix} f_{t-1} & \cdots & f_{t-n} & x_{t-1} & \cdots & x_{t-m} \\ \vdots & & \vdots & \vdots & & \vdots \\ f_0 & \cdots & f_{1-n} & x_0 & \cdots & x_{1-m} \end{bmatrix}}_{\boldsymbol{\Phi}} \underbrace{\begin{bmatrix} -a_1 \\ \vdots \\ -a_n \\ b_1 \\ \vdots \\ b_m \end{bmatrix}}_{\boldsymbol{\theta}}$$

Under supervised learning, if we have many training examples $\{\mathbf{x}_i, \langle \mathbf{f}_{l_i}, l_i \rangle\}_{i=1}^T$, we will have a massively *over-constrained* system for our inference problem. Least squares can train $\boldsymbol{\theta}$. However, when the normal equation $\hat{\boldsymbol{\theta}} = (\boldsymbol{\Phi}^\mathsf{T}\boldsymbol{\Phi})^{-1}\boldsymbol{\Phi}^\mathsf{T}\mathbf{y}$ becomes unstable, the recursive least squares via Kalman filtering [8] can be used instead.

Naïve Bayes Classifier. The key for Naïve Bayes is to define a good feature extractor for a flow measurement $\mathbf{x}$. We use a feature $\mathbf{y} = \left[\hat{\mu}_{\text{run\_size}}\ \hat{\mu}_{\text{gap\_length}}\right]$ by computing the sample mean values of run size (bytes) and gap length (unit intervals) from $\mathbf{x}$. The classifier is constructed from computing empirical conditional distribution $p(\mathbf{y}|l_i)$ of the train dataset. In runtime, the trained classifiers infer the origin flow: l_i if $p(l_i|\mathbf{y}) \geq p(l_j|\mathbf{y}) \Leftrightarrow p(\mathbf{y}|l_i)p(l_i)/p(\mathbf{y}|l_j)p(l_j) \geq 1$. Here, we use a simple decision rule that compares the learned likelihood ratios $p(\mathbf{y}|l_i)$ and $p(\mathbf{y}|l_j)$ for binary classification.

Support vector machine (SVM). Boser, Guyon & Vapnik [6] first proposed SVM. SVM is a binary classification model searching for a hyperplane that maximizes separation between two classes. The hyperplane is orthogonal to the shortest line connecting the convex hulls of the classes. Support vectors are the data points along the shortest line. The hyperplane has the form $h(\mathbf{x}) = \sum_{i=1}^T \alpha_i l_i \mathbf{x}_i \cdot \mathbf{x} + b$, where $\mathbf{x}_i$ is a training example for flow i, and α_i, b the solution of a quadratic programming (QP) problem. Class label $l_i \in \{-1, 1\}$ for each $\mathbf{x}_i$ must be provided during the training. Classification of a runtime input $\mathbf{x}$ computes $\text{sign}(h(\mathbf{x}))$.

The SVM kernel trick can work with nonlinearity of data. A kernel function $K(.)$ (*e.g.*, radial basis, sigmoid) maps the input $\mathbf{x}$ to a higher dimensional feature space where a better margin is possible. The hyperplane with the kernel trick becomes $h_{\text{kern}}(\mathbf{x}) = \sum_{i=1}^T \alpha_i l_i K(\mathbf{x}_i, \mathbf{x}) + b$.

Gaussian mixture model (GMM). Strictly speaking, GMM is an unsupervised feature learning method that is later paired with supervised classifier training. GMM assumes the probability density of input data as a weighted sum of K Gaussian distributions. GMM can be thought as a model-based version for K-means clustering. Parameterized by $\{w_j, \mu_j, \Sigma_j\}_{j=1}^K$, the feature for an input $\mathbf{x}$ is a combination of posterior membership probabilities evaluated by the Gaussians. For jth Gaussian, we have

$$p_j(\mathbf{x}) = \frac{1}{(2\pi)^{N/2}|\Sigma_j|^{1/2}} \cdot \exp\left[-\frac{1}{2}(\mathbf{x}-\mu_j)^\top \Sigma_j^{-1}(\mathbf{x}-\mu_j)\right]$$

Expectation maximization (EM) [11] trains GMMs iteratively. In E-step, EM creates a function evaluating the expected log-likelihood with the current estimate of the parameters. M-step computes new parameter values that maximizes the expected log-likelihood of the E-step.

4 Origin Flow Inference with Deep Learning

Deep learning refers to multiple layers of extracting features and nonlinear aggregation of the extracted features.

This section presents our deep learning approach for the origin flow inference problem.

4.1 Overview

We propose an inference system based on semi-supervised learning. At the first stage, the system performs *unsupervised* feature learning over multiple layers.

1. Do sparse coding and dictionary learning with unlabeled training dataset
2. Pool sparse representations of the training dataset to reduce the number of features
3. Pass the resulting features (*i.e.*, pooled sparse representations) to next layer and repeat by treating current layer's features as input data for next layer

Given multiple layers of the learned dictionaries and features, the system next performs *supervised* learning.

1. Do multi-layer sparse coding and pooling with labeled training dataset
2. Train (linear) SVM classifiers with the final form of feature vector resulted at the top layer

At runtime, the system takes a sample measurement of a flow, performs the multi-layer inference (*i.e.*, sparse coding and pooling), and predicts the origin flow pattern and properties.

4.2 Unsupervised Feature Learning

4.2.1 Sparse coding

We use sparse coding [20] as the primary means to extract features from the sampled time-series data. Consider unlabeled dataset $\{\mathbf{x}_k\}_{k=1}^T$ with each $\mathbf{x}_k \in \mathbb{R}^N$. We pack $\{\mathbf{x}_k\}_{k=1}^T$ to the columns of $\mathbf{X} = [\mathbf{x}_1^\top \ \mathbf{x}_2^\top \ \dots \ \mathbf{x}_T^\top]$. Note $\mathbf{X} \in \mathbb{R}^{N \times T}$. Sparse coding requires a dictionary $D \in \mathbb{R}^{N \times P}$ learned from $\mathbf{X}$. We adopt K-SVD [4] that learns D in the following optimization

$$\min_{D,\mathbf{Y}} \|\mathbf{X} - D\mathbf{Y}\|_F^2 \quad \text{s.t. } \|\mathbf{y}_k\|_0 \le K \ \ \forall k \tag{1}$$

Here, the columns of $\mathbf{Y} \in \mathbb{R}^{P \times T}$ or $\{\mathbf{y}_k\}_{k=1}^T$, are the sparse representations of $\{\mathbf{x}_k\}_{k=1}^T$. (Note $\mathbf{y}_k \in \mathbb{R}^P$.)

K-SVD is a fast iterative algorithm and requires to compute sparse code $\mathbf{y}_k$ for each $\mathbf{x}_k$ with current D. We use orthogonal matching pursuit (OMP) [21] for computing sparse codes. Our choice of OMP is merely based on its computational efficiency, and there are other algorithms such as LASSO [26] and LARS [13] that also work well.

The columns of D, $\{\mathbf{d}_j\}_{j=1}^P$, are dictionary atoms. Hence, each element in vector $\mathbf{y}$ reflects a degree of membership to the corresponding dictionary atom. To represent unbiased membership, dictionary atoms are normalized such that $\|\mathbf{d}_j\|_{\ell_2}^2 = 1$. To make every $\mathbf{d}_j$ meaningful, we need more training samples than dictionary atoms, so $T \ge P$. For discriminative convenience, D is overcomplete—*i.e.*, $P > N$. This means that sparse code $\mathbf{y}$ has a higher dimensionality than $\mathbf{x}$, but $\mathbf{y}$ is K-sparse, that is, only $K \ll N$ entries of $\mathbf{y}$ are nonzero.

4.2.2 Max pooling

Every input vector $\mathbf{x}_k$ is mapped to the corresponding feature vector $\mathbf{y}_k$ via sparse coding. If all feature vectors were used straightforwardly, we could overwhelm the unsupervised feature learning process. It is customary to reduce the number of extracted features by subsampling (or summarizing over) feature vectors—note, this is by no means to discard any useful information.

Pooling, popular in convolutional neural networks [17], operates over multiple (sparse) feature representations and aggregates to a higher level of features in reduced dimension. An important property of pooled feature representation is translation invariance [24]. We use *max* pooling [7] that takes the maximum value for the elements in the same position over a group of feature vectors. For example, consider max pooling of L sparse codes $\{\mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_L\}$ that yields $\mathbf{z} = \text{max\_pool}(\mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_L)$ in Figure 5. Noting $\mathbf{y}_k = [y_{k,1} \ \dots \ y_{k,P}]$ and $\mathbf{z} = [z_1 \ \dots \ z_P]$, max pooling results in $z_j = \max(y_{1,j}, y_{2,j}, \dots, y_{L,j})$.

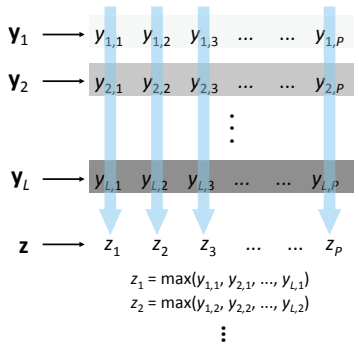


Figure 5: Max pooling of L sparse codes

4.2.3 Multi-layer deep learning

Figure 6 presents our multi-layer deep learning. We use 2 layers. Each layer trains own dictionary and has separate sparse coding and pooling units. Assuming input vector $\mathbf{x}_k$ has a moderate size N, we perform batch processing of multiple $\mathbf{x}_k$'s concatenated in series. Figure 6 showcases 4 input vectors with pooling unit configured at $L = 2$. If $\mathbf{x}_k$ has a very large N on the other hand, $\mathbf{x}_k$ can be divided

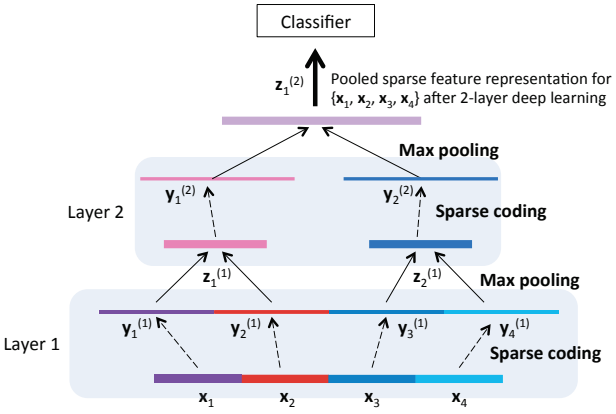


Figure 6: Baseline 2-layer deep learning with sparse coding and max pooling

into subpatches, and we perform sparse coding on each subpatch. The input vector length for sparse coding is an important system parameter for deep feature learning.

Max pooling is performed over sets of two consecutive sparse representations $\{\mathbf{y}_1^{(1)},\mathbf{y}_2^{(1)}\}$ and $\{\mathbf{y}_3^{(1)},\mathbf{y}_4^{(1)}\}$. We use notation $\mathbf{y}_k^{(1)}$ for kth sparse code at layer 1. The intermediate pooled features, $\mathbf{z}_1^{(1)}$ and $\mathbf{z}_2^{(1)}$, are sent to layer 2 for another round of sparse coding and max pooling. At the top, $\mathbf{z}_1^{(2)}$—obtained by pooling the layer 2 sparse codes $\mathbf{y}_1^{(2)}$ and $\mathbf{y}_2^{(2)}$—gives the final feature representation for $\{\mathbf{x}_1,\mathbf{x}_2,\mathbf{x}_3,\mathbf{x}_4\}$ in this 2-layer deep learning.

In general, a depth or the number of layers reflects the coverage of deep learning. Layer 1 extracts small, local features over multiple intervals spanned by consecutive input vectors. The resulting feature representations are subsampled with max pooling before passed to layer 2. Layer 2 builds larger features using its own dictionary. Because the feature coverage by layer 1 coding and pooling is over a subregion for the layer 2 coverage, the features aggregated at layer 2 are novel and could not be seen at layer 1.

4.3 Supervised Learning

We embrace the supervised learning that largely consists of training SVM classifiers. The SVM classification framework is generic and can directly work on $\mathbf{x}$ without any feature extraction or pooling. SVMs could be trained with a single-layer sparse representation $\mathbf{y}^{(1)}$ subject to $\mathbf{x} = D^{(1)}\mathbf{y}^{(1)}$. Under our 2-layer deep learning setup, we train linear SVM classifiers using the final pooled feature vectors $\mathbf{z}^{(2)}$.

Considering there are many data patterns generated by applications, the origin flow inference is a multi-class classification problem. There are two approaches for SVM, which is inherently a binary classifier, to classify q origin flow patterns. The first approach is to train all $\frac{q(q-1)}{2}$ 1-vs-1 SVMs exhaustively. Each SVM is dedicated to distinguish between any pair out of q origin flows and infers the original flow properties mapped by the classification result.

The second approach is to train q 1-vs-all SVMs. For flow i, training 1-vs-all SVM will require two datasets $\mathbf{X}_i$ with label $l_i = 1$ consisting of measured patterns of flow i only and $\mathbf{X}_{\setminus i}$ with label $l_j = -1$ containing measurements for all other flows $j\ \forall j \neq i$. Ideally, $\mathbf{X}_{\setminus i}$ should contain unbiased mix of the other flows. Our empirical evaluation in Section 6 considers the 1-vs-all approach.

5 Enhancements

5.1 Incoherent Dictionary Learning

Dictionary learning algorithms address the performance of sparse coding in two aspects: 1) reconstructive accuracy and 2) discriminative ability of the learned dictionary atoms (*i.e.*, the column vectors of D). We emphasize the latter aspect because our primary objective is classification rather than compressing data. Discriminative ability of a dictionary is related to making its atoms as *incoherent* as possible. Sparse coding with a dictionary consisting of more incoherent column vectors should improve the margin of an SVM, which results in a better classification performance.

The maximally incoherent D is constrained such that $D^{\mathsf{T}}D = I$. In other words, an incoherent dictionary matrix has orthonormal columns. This is equivalent to minimizing $\left\|D^{\mathsf{T}}D - I\right\|_F^2$. We can also think of having the two conditions $\mathbf{d}_k^{\mathsf{T}}\mathbf{d}_j = 0\ \forall k \neq j$ (orthogonal columns) and $\left\|\mathbf{d}_k^{\mathsf{T}}\mathbf{d}_k\right\|_2 = 1$ (normalized).

Since we use K-SVD, we add the incoherence optimization term to Equation (1)

$$\min_{D,\mathbf{Y}} \|\mathbf{X} - D\mathbf{Y}\|_F^2 + \gamma\left\|D^{\mathsf{T}}D - I\right\|_F^2 \text{ s.t. } \|\mathbf{y}_k\|_0 \leq K\ \ \forall k \tag{2}$$

The new optimization here, however, is not a trivial task. For the time being, we propose a two-stage algorithm presented below instead.

In the outer **for** loop, we run K-SVD unmodified. The resulting D then enters the inner **while** loop that implements the gradient descent algorithm [25] to *regularize* the incoherence term in Eq. (2). γ is the step size for gradient search and decayed by $0 < \delta < 1$ within the inner loop until initialized back to the default value γ_0 in the outer loop after running K-SVD with the next training vector.

Algorithm 1 Two-stage incoherent dictionary learning

Require: training dataset $\mathbf{X} = \left[\mathbf{x}_1^\top \ \ldots \ \mathbf{x}_T^\top\right]$

initialize $D := I$
for $i = 1$ to T
  $D := \mathsf{ksvd}(D, \mathbf{x}_k, K)$
  initialize $\gamma := \gamma_0$
  while $\left\|D^\mathsf{T} D - I\right\|_F^2 > \varepsilon$
    $D := D - \gamma D(D^\mathsf{T} D - I)$
    $D := \mathsf{normalize\_columns}(D)$
    $\gamma := \gamma \cdot \delta$
  end
end

5.2 Sparsity Relaxation

Strictly speaking, the way we force the dictionary incoherence is flawed. The gradient descent takes over after K-SVD, but K-SVD and the gradient descent regularization have to take place jointly as Equation (2) suggests. For this reason, the resulting effect of the outer loop computation perturbs the K-SVD optimization. In other words, improving the dictionary incoherence comes with the cost of reconstructive accuracy.

As a result, using the same value of K for sparse coding may be too tight to meet the minimal error criterion. Therefore, we must relax the original sparsity K for sparse coding to K' such that $K' > K$. We do sparsity relaxation as follows. Let D' represent the incoherent dictionary resulted from Algorithm 1. We repeat sparse coding with D' to find $\mathbf{Y}'$

$$\min_{\mathbf{Y}'} \left\|\mathbf{X} - D'\mathbf{Y}'\right\|_F^2 \quad \text{s.t.} \ \left\|\mathbf{y'}_k\right\|_0 \le K' \ \ \forall k$$

5.3 Dense Sparse Coding and Overlapping Pooling

The baseline deep learning scheme in Figure 6 does batch sparse coding of $\{\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \mathbf{x}_4\}$ and max pooling of $\{\mathbf{y}_1, \mathbf{y}_2\}$ and $\{\mathbf{y}_3, \mathbf{y}_4\}$. We can enhance this baseline by performing sparse coding on shifted $\mathbf{x}_k$'s and max pooling over the resulting, overlapping sparse codes. This is illustrated in Figure 7. The overlapping intervals are formed by shifting (*i.e.*, delaying) the elements in $\mathbf{x}_k$'s altogether by τ. Note that $\tau = 1$ gives the fully overlapped intervals while there is no overlapping for $\tau = 4 \cdot N$, which equals the baseline. (In our evaluation, we use a 95% overlap between consecutive $\mathbf{x}$'s.) Dense sparse coding can substantially reduce chances to miss a feature with increased cost of computing. Overlapping pooling further improves on the translational invariance of a feature. Also, according to Krizhevsky *et al.* [15], overlapping pooling reduces overfit in classifiers.

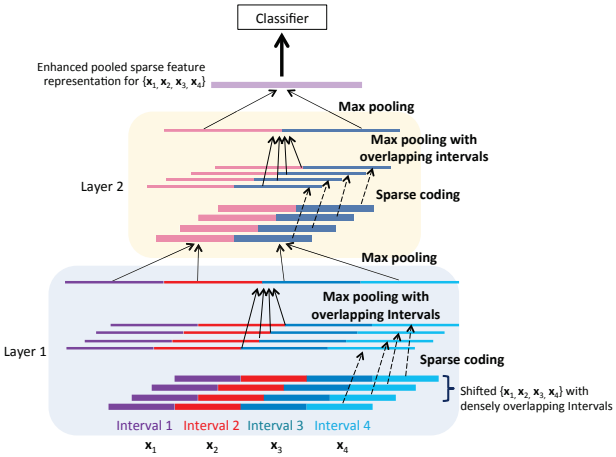


Figure 7: Enhanced 2-layer deep learning with dense sparse coding and overlapping max pooling

6 Evaluation

We evaluate the proposed baseline and enhanced inference schemes in comparison to classical machine learning approaches described in Section 3. We have implemented a simple setup featuring three Wi-Fi nodes in a custom MATLAB simulator and used OPNET Modeler to test more elaborated, seven Wi-Fi node scenario.

6.1 Methodology

6.1.1 Flow generation

We generate flows based on the runs-and-gaps model explained in Section 2. The triplet $\langle t_r, s_r, t_g \rangle$ describes the generative pattern of a flow, where t_r and t_g are the run and gap lengths in number of unit intervals (T_s), and s_r denotes the size of payload in bytes generated per each unit interval of a run. A flow type can be *constant*, *stochastic*, or *mixed*. A constant flow has deterministic t_r, s_r, and t_g values. For example, flow 1 with $\langle 2, 100, 4 \rangle$ creates the origin flow pattern (time series) $\mathbf{f}_1 = [100\ 100\ 0\ 0\ 0\ 0\ 100\ 100\ 0\ 0\ \ldots\ ]$. For a stochastic flow, t_r, s_r, and t_g are random variables. For example, flow 6 with $\langle \mathrm{Exp}(0.5), \mathrm{Pareto}(40, 1), \mathrm{Exp}(0.25) \rangle$ has exponentially distributed run and gap lengths (with mean of 1/0.5 and 1/0.25 unit intervals, respectively), and Pareto-distributed payload sizes. An instance of flow 6 could be $\mathbf{f}_6 = [518\ 97\ 0\ 0\ 0\ 0\ 0\ 0\ 32\ 0\ \ldots\ ]$.

We consider 10 origin flows summarized in Table 1. Notice we also use the normal (N) and uniform (U) distributions. We round fractions, discard negative numbers drawn from a normal distribution and regenerate. Using $T_s = 10$ msec, we generate 2,000 instances of time series for each flow. We use the first 1,000 instances for training and the other 1,000 for testing. Each instance is a vector

Table 1: Origin flows used for evaluation

Flow	Type	Generative triplet $\langle t_r, s_r, t_g \rangle$
Flow 1	Constant	$\langle 2, 100, 4 \rangle$
Flow 2	Constant	$\langle 2, 500, 2 \rangle$
Flow 3	Constant	$\langle 5, 200, 5 \rangle$
Flow 4	Constant	$\langle 10, 200, 10 \rangle$
Flow 5	Stochastic	$\langle$Exp(1), Pareto(100, 2), Exp(0.1)$\rangle$
Flow 6	Stochastic	$\langle$Exp(0.5), Pareto(40, 1), Exp(0.25)$\rangle$
Flow 7	Stochastic	$\langle$U(4, 10), Pareto(100, 2), Exp(0.5)$\rangle$
Flow 8	Stochastic	$\langle$N(10, 5), Pareto(40, 1), N(10, 5)$\rangle$
Flow 9	Mixed	$\langle 1$, Pareto(100, 2), $1 \rangle$
Flow 10	Mixed	$\langle 1$, Pareto(100, 2), Exp(0.25)$\rangle$

of 500 elements.

6.1.2 Preprocessing generated origin flow patterns

We precompute the mean run and gap lengths from the generated origin flow patterns in the training dataset. This is convenient because we enable simple lookup (of the precomputed values) based on the classification result of a measured flow in order to estimate the origin run and gap properties. In Figure 8, we have $\left[s_1^1\ s_2^1\ 0\ 0\ 0\ s_1^2\ 0\ 0\ 0\ 0\ 0\ s_1^3\ s_2^3\ s_3^3\ 0\ 0\ \ldots\right]$, where $s^1 = \sum_{k=1}^{2} s_k^1$, $s^2 = \sum_{k=1}^{1} s_k^2$, $s^3 = \sum_{k=1}^{3} s_k^3$ give total bytes of the three bursts. We can then compute the mean burst size for this pattern. We also compute $\{t_r^1, t_r^2, t_r^3, \ldots\}$, $\{t_g^1, t_g^2, t_g^3, \ldots\}$, and their mean values.

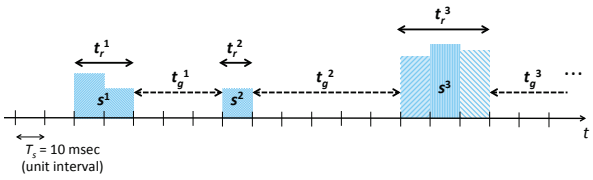


Figure 8: Computing generated flow statistics

6.1.3 Evaluation metrics

We are foremost interested in the accuracy of classifying a measured pattern $\mathbf{x}$ to its ground-truth origin flow pattern $\mathbf{f}$. We compute two metrics, *recall* (true positive rate) and *false alarm* (false positive rate), to evaluate classification performance:

$$Recall = \frac{\sum \text{True positives}}{\sum \text{True positives} + \sum \text{False negatives}}$$

$$False\ alarm = \frac{\sum \text{False positives}}{\sum \text{False positives} + \sum \text{True negatives}}$$

Without false alarm rate, we cannot truly assess the probability of detection for a classifier using a computed recall value because the classifier can be configured to declare positive only, automatically achieving to guess all positives correctly. Classification leads to inferring other important properties of a flow from its training dataset records. As our secondary evaluation metrics, we calculate errors in estimating the original mean burst size and mean gap length of the flow.

Table 2: Wi-Fi parameter configuration for Scenario 1

Parameter	Description	Value
aSlotTime	Slot time	20 μsec
aSIFSTime	Short interframe space (SIFS)	10 μsec
aDIFSTime	DCF interframe space (DIFS)	50 μsec
aCWmin	Min contention window size	15 slots
aCWmax	Max contention window size	1023 slots
tPLCPPreamble	PLCP preamble duration	16 μsec
tPLCP_SIG	PLCP SIGNAL field duration	4 μsec
tSymbol	OFDM symbol duration	4 μsec

6.2 Scenario 1: Three Wi-Fi Nodes

Figure 9 depicts Scenario 1. In this simple scenario, we infer the origin time series $\mathbf{f}_A$ sent by source node A, using $\mathbf{x}_{A|B}$ measured at receiver node B. Node C, another source, contends with node A by transmitting its own flow $\mathbf{f}_C$. We carry out cross-validation with all 10 flow datasets by setting $\mathbf{f}_A = \mathbf{f}_i\ \forall i \in \{1, \ldots, 10\}$, flow by flow at once. When $\mathbf{f}_A = \mathbf{f}_i$, we randomly set $\mathbf{f}_C = \mathbf{f}_j\ \forall j \neq i$. Node C can change its flow pattern from $\mathbf{f}_j$ to $\mathbf{f}_k$, while node A still running $\mathbf{f}_i$, but $\mathbf{f}_k$ is chosen such that $k \neq i$.

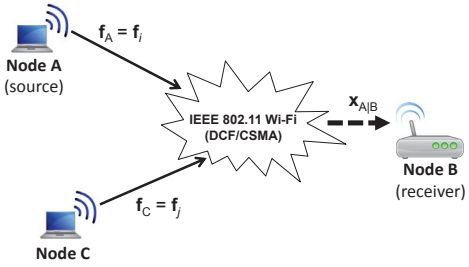


Figure 9: Scenario 1

Wi-Fi setup. We have implemented a custom discrete-event simulator in MATLAB, assuming the IEEE 802.11g our baseline Wi-Fi system. At its core, our CSMA implementation is based on an open-source wireless simulator [2]. The backoff mechanism works as follows. The contention window `CW` is initialized to *aCWmin*. In case of timeout, CSMA doubles `CW`, otherwise waits until the channel becomes idle with an additional DCF interframe space (DIFS) duration. CSMA chooses a uniformly random wait time between [1, `CW`]. `CW` can grow up to *aCWmax* of 1,023 slots. `CW` is decremented only when the media is sensed idle. RTS and CTS are disabled. The Wi-Fi configuration is summarized in Table 2.

Inference schemes. We have implemented all of the inference schemes in MATLAB. We consider ARMAX-

Table 3: Errors to estimate original mean burst size and mean gap length with 5-sec observation window (error percentages in parenthesis are for Scenario 2)

Scheme	Origin burst size estimation error	Origin gap length estimation error
ARMAX	39.3% (45.9%)	28.1% (36.7%)
Naïve Bayes	31.4% (37.5%)	15.8% (24.6%)
GMMs	23.2% (31.3%)	11.7% (18.1%)
DL (baseline)	18.7% (28.3%)	9.3% (16.2%)
DL (enhanced)	12.2% (22.8%)	6.8% (11.4%)

least squares, Naïve Bayes classifiers, Gaussian mixture models (GMM), and the two deep learning methods we proposed. The baseline deep learning method has 2 layers implemented as described in Section 4. The enhanced deep learning method also has 2 layers, but we additionally have implemented incoherent dictionary training, dense sparse coding, and overlapping max pooling as described in Section 5. We call the size of vector $\mathbf{x}$ *observed length* or *observation window size* and have varied 100, 200, 300, and 500.

Training. As mentioned in §6.1.1, the training dataset for each flow has 1,000 instances. For flow i, we transmit training examples $\{\mathbf{f}_{i,k}\}_{k=1}^{1000}$ serially. (Note the notation $\mathbf{f}_{i,k}$ for kth instance of flow i.) We consider 1-vs-all linear SVM classifiers for all inference schemes.

We train ARMAX with n previously *transmitted* origin flow patterns and m previous inputs—*i.e.*, for flow i, we feed $\{\mathbf{f}_{i,k-1},\ldots,\mathbf{f}_{i,k-n}\}$ and $\{\mathbf{x}_{i,k-1},\ldots,\mathbf{x}_{i,k-m}\}$—at time k. After trying out various configurations, we choose $n=2$ and $m=3$. The least squares directly estimate $\hat{\mathbf{f}}$ given $\mathbf{x}$. SVMs for ARMAX are trained with $\hat{\mathbf{f}}$s.

For Naïve Bayes, we extract the statistic $\mathbf{y} = \left[\hat{\mu}_{\text{run\_size}}\ \hat{\mu}_{\text{gap\_length}}\right]$ from $\mathbf{x}$ by averaging run burst sizes and gap lengths and use $\mathbf{y}$ as a feature with label l_i for flow i to build empirical distributions $p(\mathbf{y}|\mathbf{x},l_i)$. Note training Naïve Bayes yields classifiers, so we do not need to train any SVMs for Naïve Bayes.

We use K = 10 GMMs. Like Bayes, we use the same static $\mathbf{y}$ from $\mathbf{x}$ as a feature to train GMMs via the EM algorithm. However, unlike Bayes, GMMs do not yield classifiers. We train SVMs with the membership probabilities from the trained K Gaussians evaluated on $\mathbf{y}$ given l_i.

The proposed deep learning methods produce a max-pooled sparse representation $\mathbf{z}^{(2)}$ at the top of layer 2 for given $\mathbf{x}$. We use labeled $\mathbf{z}^{(2)}$'s to train SVMs.

Results. Figure 10 presents classification recall and false alarm rate of each inference scheme for Scenario 1. Overall, deep learning (DL) yields consistently higher recall at lower false alarm. When using a small observation window length to sample $\mathbf{x}$, the recall gap between the enhanced and baseline deep learning methods is noticeably larger than the case of using a large observation window. This is because dense sparse coding and overlapping max pooling in the enhanced deep learning scheme substantially reduce the probability of missing a feature. A possible explanation for poor ARMAX performance could be that CSMA introduces significant nonlinear distortions. GMMs are on par with our baseline deep learning. If optimized to their limits, GMMs seem to be a reasonable alternative to deep learning for classification. This is not surprising since GMMs are really a form of K-SVD. (Note also that our effort to fine-tune GMMs was only fair as our focus in this paper was on deep learning.)

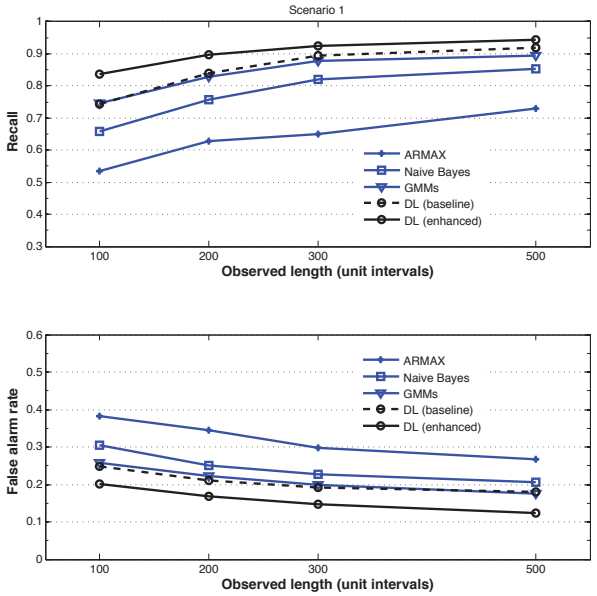


Figure 10: Classification recall and false alarm rate for Scenario 1

After classification, we predict the original mean burst size and gap length of a flow based on lookup of the precomputed values from the training dataset (see explanation in §6.1.2). The estimation errors in Table 3 are the best case, obtained with the maximum observation window size of 5 seconds that we have tried.

6.3 Scenario 2: Wi-Fi Nodes in OPNET

Figure 11 illustrates Scenario 2 featuring seven Wi-Fi nodes simulated in OPNET. This scenario is important for several reasons. First, we can configure more realistic application profiles for simulated nodes. We can also scale the simulation. Lastly, we can validate our schemes with OPNET's built-in Wi-Fi protocols, particularly the IEEE 802.11g, which should be more complete than our MATLAB simulator. Scenario 2 preserves nodes A, B, C and their activities the same as Scenario 1. There are five additional Wi-Fi nodes that communicate in typical Internet styles as summarized in Table 4.

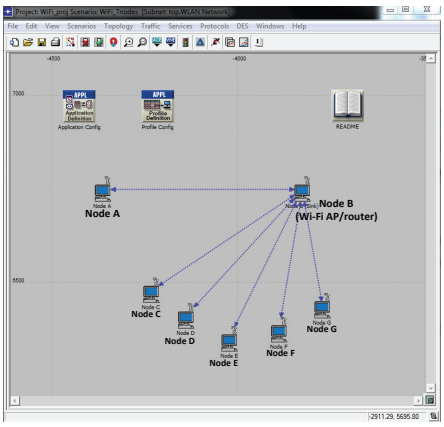


Figure 11: Scenario 2

Table 4: Configuration summary of Scenario 2

Node	Role	Main networking activity
A	Flow source	Transmits $\mathbf{f}_i$
B	Receiver	Intercepts flows as Wi-Fi router/AP
C	Flow source	Transmits $\mathbf{f}_j \; \forall \; j \neq i$
D	Flow source	Multimedia streaming over RTP/UDP/IP
E	Flow dest.	HTTP with page size $\sim$ U[10,400] B
F	Flow dest.	ftp file transfer with size 50000 B
G	Flow dest.	DB access with inter-arrival $\sim$ Exp(3) sec

We test the same inference schemes and keep the training methodology of Scenario 1. In Figure 12, we plot classification recall and false alarm rate of each scheme evaluated under Scenario 2. With more nodes and increased traffic, the overall classification performance is worse. Again, we can clearly see the benefits of dense sparse coding and overlapping pooling for enhanced deep learning that consistently outperforms the other schemes over various observation window sizes. For a small observation window in particular, recall for enhanced deep learning is substantially higher while achieving the lowest false alarm rate. Table 3 shows the estimation errors (in parenthesis) to predict original properties of the flows, the mean burst size and mean gap length, after classification.

7 Conclusion

We have addressed the problem of inferring the original properties of a flow sampled from a received Wi-Fi traffic mix. This inverse problem is challenging because CSMA significantly changes the origin pattern of the flow while scheduling with other flows in competition. Machine learning can provide tools to harness complexity and nonlinearity, but it requires to apply adept domain knowledge or application-specific insights to configure the overall learning pipeline and fine-tune all model parameters to their meticulous detail.

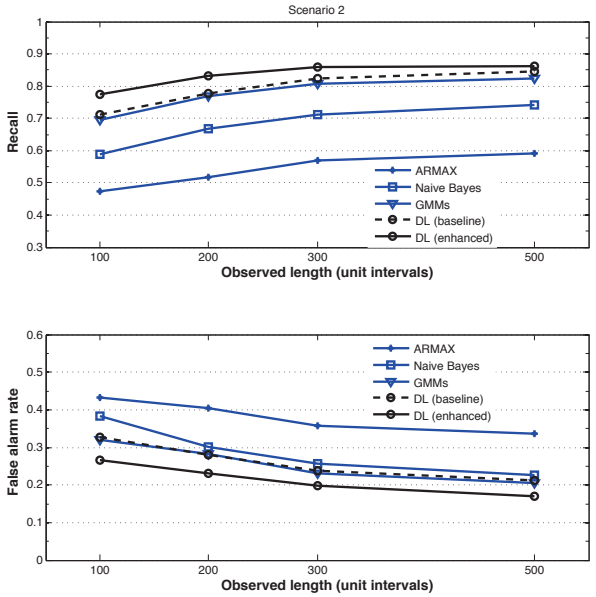


Figure 12: Classification recall and false alarm rate for Scenario 2

Learned from our initial, unsuccessful attempt to straightforwardly integrate sparse coding and dictionary learning into SVM classification, we have set up multiple layers of sparse feature extraction and max pooling that enable deep learning from received flow patterns. Multi-layer sparse coding allows us to learn local, atomic features such as run and gap sizes of a flow accurately at the lowest layer and global features such as periodicity in traffic patterns at higher layers. Max pooling summarizes often too many extracted features while providing translation invariance. We have explained how the proposed approaches incorporate these ideas and validated their superior performance.

In summary, contributions of this paper include a novel formulation of an inverse problem to recover origin flow patterns in Wi-Fi, identification of the key attributes for successful machine learning approaches to solve such inverse problems (*i.e.*, the use of sparse representation of features, multi-layer inference, and pooling), and demonstration of the sound working of these methods in recovering original flow properties. We have chosen not to discuss possible applications of this paper in security and network anomaly detection. We plan to address such applications in our future work.

Acknowledgment

We thank anonymous reviewers of ICAC for their valuable comments in revising the final version of this paper. This material is based on research sponsored in part by Intel Corporation.

References

[1] IEEE 802.11™ Wireless Local Area Networks. `http://www.ieee802.org/11/`.

[2] MATLAB Wireless Network Simulator. `http://wireless-matlab.sourceforge.net`.

[3] Software-defined Networking: The New Norm for Networks. Open Networking Foundation (White Paper), 2012.

[4] AHARON, M., ELAD, M., AND BRUCKSTEIN, A. K-SVD: An Algorithm for Designing Overcomplete Dictionaries for Sparse Representation. *IEEE Trans. on Sig. Proc. 54*, 11 (2006).

[5] BASU, S., MUKHERJEE, A., AND KLIVANSKY, S. Time Series Models for Internet Traffic. In *INFOCOM* (1996).

[6] BOSER, B. E., GUYON, I. M., AND VAPNIK, V. N. A Training Algorithm for Optimal Margin Classifiers. In *COLT* (1992).

[7] BOUREAU, Y.-L., PONCE, J., AND LECUN, Y. A Theoretical Analysis of Feature Pooling in Visual Recognition. In *ICML* (2010).

[8] CIOFFI, J., AND KAILATH, T. Fast, Recursive-least-squares Transversal Filters for Adaptive Filtering. *IEEE Trans. on Acoustics, Speech and Signal Processing 32*, 2 (1984), 304–337.

[9] CLAFFY, K. C., POLYZOS, G. C., AND BRAUN, H.-W. Application of Sampling Methodologies to Network Traffic Characterization. In *ACM SIGCOMM* (1993).

[10] COATES, A., AND NG, A. Selecting Receptive Fields in Deep Networks. In *NIPS*. 2011.

[11] DEMPSTER, A. P., LAIRD, N. M., AND RUBIN, D. B. Maximum Likelihood from Incomplete Data via the EM Algorithm. *Journal of Royal Statistical Society, Series B 39*, 1 (1977).

[12] DUFFIELD, N., LUND, C., AND THORUP, M. Estimating Flow Distributions from Sampled Flow Statistics. In *ACM SIGCOMM* (2003).

[13] EFRON, B., HASTIE, T., JOHNSTONE, I., AND TIBSHIRANI, R. Least Angle Regression. *Annals of Statistics 32* (2004), 407–499.

[14] HEISELE, B., HO, P., AND POGGIO, T. Face Recognition with Support Vector Machines: Global versus Component-based Approach. In *ICCV* (2001).

[15] KRIZHEVSKY, A., SUTSKEVER, I., AND HINTON, G. E. ImageNet Classification with Deep Convolutional Neural Networks. In *NIPS* (2012), vol. 1, p. 4.

[16] KUNG, H. T., LIN, C.-K., LIN, T.-H., TARSA, S. J., VLAH, D., HAGUE, D., MUCCIO, M., POLAND, B., AND SUTER, B. Location-dependent Runs-and-gaps Model for Predicting TCP Performance over UAV Wireless Channel. In *MILCOM* (2010).

[17] LAWRENCE, S., GILES, C. L., TSOI, A. C., AND BACK, A. D. Face Recognition: A Convolutional Neural Network Approach. *IEEE Trans. on Neural Networks 8*, 1 (1997), 98–113.

[18] LIN, T., LIU, S., AND ZHA, H. Incoherent Dictionary Learning for Sparse Representation. In *ICPR* (2012).

[19] LJUNG, L., Ed. *System Identification (2nd ed.): Theory for the User*. Prentice Hall, 1999.

[20] OLSHAUSEN, B., AND FIELD, D. Emergence of Simple-cell Receptive Field Properties by Learning a Sparse Code for Natural Images. *Nature 381*, 6583 (1996), 607–609.

[21] PATI, Y. C. *et al.* Orthogonal Matching Pursuit: Recursive Function Approximation with Applications to Wavelet Decomposition. In *Asilomar Conference on Signals, Systems and Computers* (1993).

[22] RAINA, R., BATTLE, A., LEE, H., PACKER, B., AND NG, A. Y. Self-taught Learning: Transfer Learning from Unlabeled Data. In *ICML* (2007).

[23] RAMIREZ, I., SPRECHMANN, P., AND SAPIRO, G. Classification and Clustering via Dictionary Learning with Structured Incoherence and Shared Features. In *IEEE CVPR* (2010).

[24] RIESENHUBER, M., AND POGGIO, T. Hierarchical Models of Object Recognition in Cortex. *Nature 2*, 11 (1999), 1019–1025.

[25] SNYMAN, J. A. *An Introduction to Basic Optimization Theory: Classical and New Gradient-based Algorithms*. Springer, 2005.

[26] TIBSHIRANI, R. Regression Shrinkage and Selection via the Lasso. *Journal of Royal Statistical Society, Series B 58* (1994).

[27] WRIGHT, J., YANG, A., GANESH, A., SASTRY, S., AND M., Y. Robust Face Recognition via Sparse Representation. *IEEE Trans. on Pattern Analysis and Machine Intelligence 31*, 2 (2009), 210–227.

[28] ZHANG, Q., AND LI, B. Discriminative K-SVD for Dictionary Learning in Face Recognition. In *IEEE CVPR* (2010).

Guarded Modules: Adaptively Extending the VMM's Privileges Into the Guest

Kyle C. Hale and Peter A. Dinda
{k-hale, pdinda}@northwestern.edu
Department of Electrical Engineering and Computer Science
Northwestern University

Abstract

When a virtual machine monitor (VMM) provides code that executes in the context of a guest operating system, allowing that code to have privileged access to specific hardware and VMM resources can enable new mechanisms to enhance functionality, performance, and adaptability. We present a software technique, guarded execution of privileged code in the guest, that allows the VMM to provide this capability, as well as an implementation for Linux guests in the Palacios VMM. Our system, which combines compile-time, link-time, and runtime techniques, provides the module developer with the following guarantees: (1) A kernel module will remain unmodified and it will acquire privilege only when untrusted code invokes it through developer-chosen, valid entry points with a valid stack. (2) Any execution path leaving the module will trigger a revocation of privilege. (3) The module has access to private memory. The system also provides the administrator with a secure method to bind a specific module with particular privileges implemented by the VMM. This lays the basis for guaranteeing that *only* trusted code in the guest can utilize special privileges. We give two examples of guarded Linux kernel modules: a network interface driver with direct access to the physical NIC and an idle loop that uses instructions not usually permitted in a guest, but which can be adaptively selected when no other virtual core shares the physical core. In both cases *only* the guarded module has these privileges.

This project is made possible by support from the United States National Science Foundation (NSF) via grant CNS-0709168 and the Department of Energy (DOE) via grant DE-SC0005343.

1 Introduction

By design, a virtual machine monitor (VMM) does not trust the guest operating system and thus does not allow it access to privileged hardware or VMM state. However, such access can allow new or better services for the guest, such as the following examples.

- Direct guest access to I/O devices would allow existing guest drivers to be used, avoid the need for virtual devices, and accelerate access when the device could be dedicated to the guest. In existing systems, the VMM limits the damage that a rogue guest could inflict by only using self-virtualizing devices [14, 19] or by operating in contexts such as HPC environments, where the guest is trusted and often runs alone [10].
- Direct guest access to the Model-Specific Registers (MSRs) that control dynamic voltage and frequency scaling (DVFS) would allow the guest's adaptive control of these features to be used instead of the VMM's whenever possible. Because applications running on the guest enjoy access to more rich information than the VMM does, there is reason to believe that guest-based control would perform better.
- Direct guest access to instructions that can halt the processor, such as `monitor` and `mwait`, would allow more efficient idle loops and spinlocks when the VMM determines that such halts can be permitted given the current configuration.

Since we cannot trust the guest operating system, to create such services we must be able to place a component *into* the guest operating system that is both tightly coupled with the guest and yet protected from it. In prior work [5], we presented GEARS, a framework for allowing the implementation of a service to span the guest and the VMM, even without guest cooperation. GEARS provides the ability to inject modules into the guest, but the injected code runs with the same privilege and the same hardware access as other, untrusted guest code. In this

paper, we extend this functionality to allow for the injected code to be endowed with privileged access to hardware and the VMM that the VMM selects, but only under specific conditions that preclude the rest of the guest from taking advantage of the privilege. We refer to this privileged injected code as a *guarded module*, and it is effectively a piece of the VMM running in the guest context.

Our technique leverages compile-time and link-time processing which identifies valid entry and exit points in the module code, including function pointers. These points are in turn "wrapped" with automatically generated stub functions that communicate with the VMM. Our current implementation of this technique applies to Linux kernel modules. The unmodified source code of the module is the input to the implementation, while the output is a kernel object file that includes the original functionality of the module and the wrappers. Conceptually, a guarded module has a *border*, and the wrapper stubs (and their locations) identify the valid *border crossings* between the guarded module, which is trusted, and the rest of the kernel, which is not.

A wrapped module can then be injected into the guest using the existing GEARS framework, or added to the guest voluntarily. The wrapper stubs and other events detected by the VMM drive the second component of our technique, a state machine that executes in the VMM. An initialization phase determines whether the wrapped module has been corrupted and where it has been loaded, and then protects it from further change. Attempted border crossings, either via the wrapper functions or due to interrupt/exception injection, are caught by the VMM and validated. Privilege is granted or revoked on a per-virtual core basis. Components of the VMM that implement privilege changes are called back through a standard interface, allowing the mechanism for privilege granting/revoking to be decoupled from the mechanism for determining when privilege should change. The privilege policy is under the ultimate control of the administrator, who can determine the binding of specific guarded modules with specific privilege mechanisms.

Our contributions are as follows:

- We describe the design of the joint compile-time and run-time guarded module mechanism.
- We describe the implementation of the design for supporting guarded Linux modules in the context of the Palacios VMM [11, 9]. Our implementation is publicly available within the Palacios codebase.
- We evaluate the performance of our implementation, independent of the service and the privilege.
- We extend Palacios with a privilege mechanism, a PCI device passthrough capability that can dynamically acquire and release privilege, and then demonstrate passthrough NIC access using a guarded module that drives this mechanism. Only the module has access to the NIC.
- We extend Palacios with a second privilege mechanism, selectively-enabled access to the `monitor` and `mwait` instructions, and then demonstrate adaptive use of these instructions in a guarded module. Only the module has access to the instructions and can halt the physical core using them.

2 Related work

Process Isolation Protecting trusted applications from an untrusted OS has recently become an active area of research. Overshadow [3] first showed that hardware virtualization techniques can be used to ensure control-flow, data, and address space integrity for a process running in the guest. TrustVisor [17] extended this idea with a much smaller trusted computing base (TCB). Flicker [18] uses nascent hardware support to effectively protect trusted applications. XOMOS [13] achieves the same goal, albeit with a new ABI and an ISA that has not yet been implemented in real hardware. InkTag [6] and Virtual Ghost [4] both aim to further defend these trusted applications from a small subset of potential Iago attacks [2], a new class of attacks in which a malicious kernel crafts return values from system services to trick a trusted application into following a code path intended by the attacker. However, these systems not only lack support for trusted *kernel components*, they also leverage existing protection domains and do not consider the protection of a trusted component from attacks originating in *the same address space*.

Kernel-space Isolation A large portion of previous work on kernel-space isolation is intended for isolating an entire kernel from *untrusted*, external components. LeVasseur's work on using virtual machines as vehicles for commodity driver reuse and fault isolation [12] shows promise, but these techniques involve using driver code residing in a completely separate virtual machine.

Swift showed, with *Nooks* [22], that code wrappers can isolate faulty code in Linux kernel extensions, improving the reliability of the core kernel. While Nooks provides an illustrative example of defining boundaries between driver and kernel code, it requires modifications to the kernel in which the drivers reside. Our system requires no such modifications. Further, Nooks does not

consider the situation in which a trusted module/extension requires protection *from* an untrusted kernel—our primary area of concern.

Both LXFI [16] and SecVisor [20] explore isolation in terms of guaranteeing kernel integrity. LXFI mitigates the potential for privilege escalation attacks against kernels by requiring that programmers annotate their modules. SecVisor insulates kernels from untrusted code by only allowing VMM-authorized code to execute, preventing a broad class of code-injection attacks against the kernel. Protecting the kernel against both malicious attacks and faulty software components are important problems, but they are orthogonal to our concerns. Our system guarantees the integrity of kernel *modules* that enjoy both a higher level of trust and privilege than the rest of the OS.

VM Introspection There have been several examples of leveraging the guest-host relationship to improve VM monitoring and resource management, especially in the context of autonomic computing [26, 15, 23, 7]. However, as far as we are aware, the only existing use case for trusted, isolated components within a guest kernel is for security monitors in which the only protected state is the code and data of the monitor itself, not higher-privilege state such as that required to access the hardware such as we outlined in Section 1.

IntroVirt [8] allows a VMM to invoke code in the guest, but does not deal with enforcing separate levels of trust within the *same* guest.

SYRINGE [1] provides a mechanism by which secure monitoring code can leverage functions in an untrusted guest. This system employs a secure VM along with an untrusted VM. When the monitoring code in the secure VM needs to call a function in the untrusted VM, the hypervisor forwards the call, managing control-flow and data integrity such that the secure VM is not compromised. However, this system is more akin to a secure, cross-core RPC facility that does not address border crossings within the same address space—a major component of our work.

Secure in-VM monitoring, or SIM [21], addresses performance issues raised by previous VM introspection techniques by allowing monitoring code to run directly in the guest while ensuring the monitor's integrity. While SIM touches on the border crossings that are our focus, it largely sidesteps the issue by using a completely separate address space for the trusted monitor code. We do not have this option as we seek to guard modules that reside in the same address space as the untrusted kernel.

As far as we are aware, the guarded module system we present is the first of its kind that guarantees both control-flow and data integrity for modules that share the same address space as an untrusted OS kernel. Guarded modules require no specialized hardware and no modifications to the guest OS in which they execute.

3 Trust and threat models; invariants

We assume a completely untrusted guest kernel. A developer will add to the VMM selective privilege mechanisms that are endowed with the same level of trust as the rest of the core VMM codebase. A module developer will assume that the relevant mechanism exists. The determination of whether a particular module is allowed access to a particular selective privilege mechanism is made at run-time by an administrator. The central relationship we are concerned with is between the untrusted guest kernel and the module. A compilation process transforms the module into a guarded module. This then interacts with run-time components to maintain specific invariants in the face of threats from the guest kernel.

Control-flow integrity The key invariant we provide is that the privilege on a given virtual core will be enabled if and only if that virtual core is executing within the code of the guarded module and the guarded module was entered via one of a set of specific, agreed-upon entry points. The privilege will be disabled whenever control flow leaves the module, including for interrupts and exceptions.

The guarded module boasts the ability to interact freely with the rest of the guest kernel. In particular, it can call other functions and access other data within the guest. A given call stack might intertwine guarded module and kernel functions, but the system guards against attacks on the stack as part of maintaining the invariant.

A valid entry into the guarded module is not checked further. Our system does not guard against an attack based on function arguments or return values, namely Iago attacks. The module author needs to validate these himself. Note, however, that the potential damage of performing this validation incorrectly is limited to the specific privilege the module has.

Code integrity Disguising the module's code is not a goal of our system. The guest kernel can read and even write the code of the guarded module. However, any modifications of the code by any virtual core will be caught and the privilege will be disabled for the remainder of the module's lifetime in the kernel. The identity of the module is determined by its content, and module

insertion is initiated external to the guest with a second identifying factor, guarding against the kernel attempting to spoof or replay a module insertion.

Data integrity Data integrity, beyond the registers and the stack, is managed explicitly by the module. The module can request private memory as a privilege. On a valid entry, the memory is mapped and is usable , while on departing the module, the memory is unmapped and rendered invisible and inaccessible to the rest of the kernel.

4 Design and implementation

The specific implementation of guarded modules we describe in this paper applies to Linux kernel modules. Our implementation fits within the context of the Palacios VMM and takes advantage of code generation and linking features of the GCC and GNU binutils toolchains. The VMM-based elements leverage functionality commonplace in modern VMMs, and thus could be readily ported to other VMMs. The code generation and linking aspects of our implementation seem to us to be feasible in any C toolchain that supports ELF or a similar format. The technique could be applicable to other guest kernels, although we do assume that the guest kernel provides runtime extensibility via some form of load-time linking.

In our implementation, a guarded Linux kernel module can either be voluntarily inserted by the guest or involuntarily injected into the guest kernel using the GEARS framework. The developer of the module needs to target the specific kernel he wants to deploy on, exactly as in creating a Linux kernel module in general.

The guarded module is a kernel module within the guest Linux kernel that is allowed privileged access to the physical hardware or to the VMM itself. The nature of this privilege, which we will describe later, depends on the specifics of the module. We refer to the code boundary between the guarded module and the rest of the guest kernel as the *border*.

Border crossings consist of control flow paths that traverse the border. A *border-out* is a traversal from the module to the rest of the kernel, of which there are three kinds. The first, a *border-out call* occurs when a kernel function is called by the guarded module, while the second, a *border-out ret*, occurs when we return back to the rest of the kernel. The third, a *border-out interrupt* occurs when an interrupt or exception is dispatched. A *border-in* is a traversal from the rest of the kernel to the guarded module. There are similarly three forms here. The first, a *border-in call* consists of a function call from the kernel to a function within the guarded module, while the second, a *border-in ret* consists of a return from a *border-out call*, and the third, a *border-in rti* consists of a return from a border-out interrupt. Valid border-ins should raise privilege, while border-outs should lower privilege. Additionally, any attempt to modify the module should lower privilege.

The VMM contains a new component, the *border control state machine*, that determines whether the guest has privileged access at any point in time. The state machine also implements a registration process in which the injected guarded module identifies itself to the VMM and is matched against validation information and desired privileges. This allows the administrator to decide which modules, by content, are allowed which privileges. After registration, the border control state machine is driven by hypercalls from the guarded module, exceptions that occur during the execution of the module, and by interrupt or exception injections that the VMM is about to perform on the guest.

The VMM detects attempted border crossings jointly through its interrupt/exception mechanisms and through hypercalls in special code added to the guarded module as part of our compilation process. Figure 1 illustrates how the two interact.

4.1 Compile-time

Our compilation process, *Christoization*[1], automatically wraps an existing kernel module with new code needed to work with the rest of the system. Two kinds of wrappers are generated. *Exit wrappers* are functions that interpose on the calls from the guarded module to the rest of the kernel. An exit wrapper, added using link-time processing, signals the VMM by a hypercall to lower privilege just before the underlying function call is made. When the function returns, it signals the VMM to validate the stack and raise privilege. *Entry wrappers* are functions that interpose on calls from the kernel into the guarded module. Entry wrappers, which are introduced by source preprocessing, use hypercalls to signal the VMM to raise privilege when called, and then lower privilege when the call returns to the kernel. The precise positions of the hypercall instructions in the wrappers are used by the VMM to validate the requests.

We designed our compile-time tool chain so that module developer effort is minimized when generating a guarded module. The requisite knowledge and materials are the same as what would be required of a developer writing a Linux kernel module. The necessary inputs to our toolchain are the guest Linux Makefile and kernel

[1] Named after the famed conceptual artist, Christo, who was known for wrapping large objects such as buildings and islands in fabric.

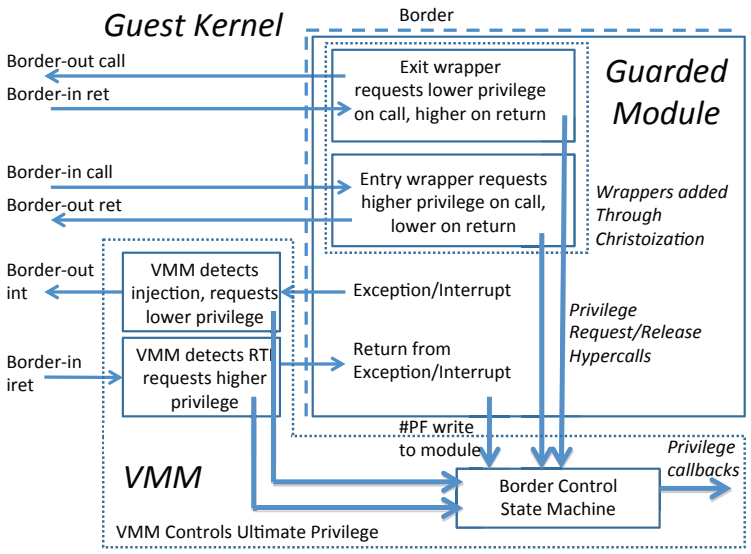


Figure 1: Guarded modules, showing operation of wrappers and interaction of state machine on border crossings.

headers, as well as the source and Makefile for the module to be Christoized. Additionally, the privilege names required by the module are passed as command-line parameters. Access to the guest Linux source tree may also be required if the developer wishes to use external functions that use non-standard calling conventions.

The first stage of the Christoization process is module source analysis. We scan the source files of the module, looking for functions that are assigned as callbacks. These functions represent entry points into the module, as the kernel will invoke them asynchronously. In order to effectively identify all of these functions, we must run a preprocessing pass over the module to make sure that external inlined functions and macros are accounted for. Once the entry callbacks are identified, we must search the source for the function that the module developer registers using Linux's `module_init` macro. This function will serve as the initial gateway into the module and must be intercepted by the VMM.

In the source annotation stage, each entry callback assignment in the source is changed to a macro that will expand to an entry wrapper function particular to that callback. These wrappers are added to the source file automatically and are depicted in Figure 2. The key idea here is that a hypercall is inserted both before and after the call to the original entry point. The remaining instructions are there to preserve the environment in such a way that the original function is not aware that it has been wrapped. The `module_init` routine is then similarly wrapped with a registration hypercall that notifies the VMM when it has been inserted into the guest kernel.

The linker wrapping stage takes the output of the annotation stage (a compiled object) and identifies undefined function references. These represent exits to the kernel. They are wrapped with exit wrappers, which are assembly stubs similar to entry wrappers. Exit wrappers lower privilege before the original call and raise it on return. They are added using `ld`'s function wrapping capability. The result of this linking step is that the module's original unresolved external references are resolved to the exit wrappers, while the exit wrappers contain references to the original unresolved symbols. As a result, any external call from the original module goes through an exit wrapper.

The final stage of the Christoization process is metadata generation. Here, information collected in the previous stages is aggregated into a formatted file with which the administrator can later register the guarded module. The essential metadata consists of the module's name, its required privileges, and the offsets in the compiled object of the identified valid entry points. This list can later be further restricted or expanded by the module developer. Additionally, to ensure module integrity at load-time, a cryptographic content hash of the code segment is performed and recorded. This metadata is later passed by the administrator to the VMM during the guarded module registration process, and it is used from then on by the border control state machine to validate the hyper-

```
entry_wrapped:
      popq  %r11
      pushq %rax
      movq  $border_in_call, %rax
(a)   vmmcall
      popq  %rax
      callq entry
      pushq %rax
      movq  $border_out_ret, %rax
(b)   vmmcall
      popq  %rax
      pushq %r11
      ret   (to rest of kernel)
```

Figure 2: An entry wrapper for a valid entry point. Exit wrappers are similar, except they invoke border out on a call, and border in after returning.

calls and other events it receives.

4.2 Run-time

The run-time element of our system is based around the border control state machine. As Figure 1 illustrates, the state machine is driven by hypercalls originating from the guarded module, and by events that are raised elsewhere in the VMM. As a side-effect of the state machine's execution, it generates callbacks to other components of the VMM that implement specific privilege changes, notifying them when valid privilege changes occur. The state machine also handles the initialization of a guarded module and its binding with these other parts of the VMM. We now describe guarded module execution with respect to the state machine.

Module initialization The guarded module is injected into the guest, either voluntarily by the user, or involuntarily by the administrator using GEARS's code injection facility. The module's initialization code immediately calls the guarded module registration function that was generated by Christoization. This function makes an initialization hypercall, providing a claimed hash as its argument. In response, the state machine validates the module using the metadata associated with the claimed hash. First, the address of the initialization hypercall instruction, combined with the known offset of the instruction in the text segment stored in the metadata, allows us to determine the load address of the module's text segment. The metadata includes the length of the text section. With this information, the state machine then marks the text segment as unwritable in the shadow or nested page tables, making it impossible for the guest to change it. The next step is to compute the hash over the text segment memory and compare it to the hash stored in the metadata.[2] If the hashes match, the state machine notifies the selective privilege-enabled component that privilege should be raised, transitions to the privileged state, enables interception of exceptions, and returns to the guest. At this point, the guarded module can complete the remainder of its initialization. In effect, module initialization is treated as the first border-in call.

Border-in call to border-out ret A valid entry into the guarded module results in a hypercall from the entry wrapper (Figure 2(a)) that requests a privilege raise. The address of this hypercall instruction is then validated against the list of addresses where such instructions were placed, which is stored in the metadata. If it is in the list, the state machine invokes a privilege-raising callback, and transitions to the privileged state. Before returning, it also enables interception of exceptions. Before exiting from a valid entry, the entry wrapper similarly invokes another hypercall (Figure 2(b)), which requests a lowering of privilege. When privilege is lowered, exception interception is returned to its nominal state.

Border-out call to border-in ret A call from the guarded module to the rest of the kernel results in a hypercall from the exit wrapper that requests a lowering of privilege. As a side-effect of lowering privilege, exception interception is returned to its nominal state. When the call returns, a second hypercall requests a raising of privilege. After sanity checking the address against the metadata, privilege is raised, and exception and interrupt interception are again enabled.

Border-out int to border-in rti The purpose of intercepting exceptions that occur when executing with privilege is to assure that we can lower privilege when these events trigger an interrupt handler dispatch and raise it once execution resumes in the guarded module. More generally, we must trap *any* switch from the guarded module code to kernel context. When the guest is not executing in the guarded module, nominal exception handling is sufficient. Our handler for exception intercepts simply causes the VMM to re-inject the exceptions alongside its normal injection of interrupt events.

Because we need to be aware of every interrupt/exception *dispatch*, we have modified the Palacios VM entry code so that, just before such an entry, if the guest is executing with privilege, we determine if an interrupt

[2] A direct comparison of the text segment content is also possible.

or exception injection will occur on the entry. If so, we lower privilege, switch back to nominal interception of exceptions, and enable interception of the `rti` instruction, which will be executed when the interrupt or exception handler completes. We also note the current `%rip` and other information related to this interrupt dispatch.

At this point, we allow the VM entry to complete, and interrupt dispatch ensues. We emulate `rti` instructions when they occur, looking for any `rti` that will return control to the instruction at which the original interrupt/exception was injected. When we discover a match, we raise privilege, re-enable exception interception, disable `rti` interception, and resume execution with privilege in the guarded module.

We note that one privilege that could be granted to a module is the ability to disable interrupts while it executes. In this case, this code path could be entirely avoided.

Internal calls The entry wrapper shown in Figure 2 and the exit wrappers are linked such that they are only invoked on border crossings. Calls internal to the guarded module do not have any additional overhead. The same applies for calls internal to the kernel.

Nesting and stack checking Although it is convenient to think of (and generate code for) border-crossings in matched pairs, it is important to realize that an execution path may involve multiple border-crossings. For example, the kernel might invoke a callback function on the module, which requires privilege, but which in turn calls a kernel function, which *should not* have privilege, and that subsequently makes another callback into the module, which *should*. The sequence of events for that example would be: border-in call, border-out call(*), border-in call, border-out ret, border-in ret(**), border-out ret. While border-ins and border-outs must eventually all be matched, they can nest. This nesting of border crossings introduces an opportunity to subvert the guarded module through the stack. Our primary concern is the protection of the `ret` in the border-out wrapper. If the border-out call(*) had its return address modified on the stack, the border-in ret(**) would return to that address with privilege raised!

To address this, the border control state machine tracks the nesting level and the stack state, and validates the stack state on any border-in. When a border-in occurs with a nesting level of zero, the state machine captures the starting point of this "first border-in" stack frame (i.e., `%rsp` and `%rbp`). When a border-out occurs, the state machine captures the ending point of this "last border-out" stack frame, and computes and stores a hash of the stack content from the first entry to this last exit. On any border-in whose nesting level is greater than zero, the actual stack is again hashed and compared with the last border-out hash. If they do not match, privilege is not granted.

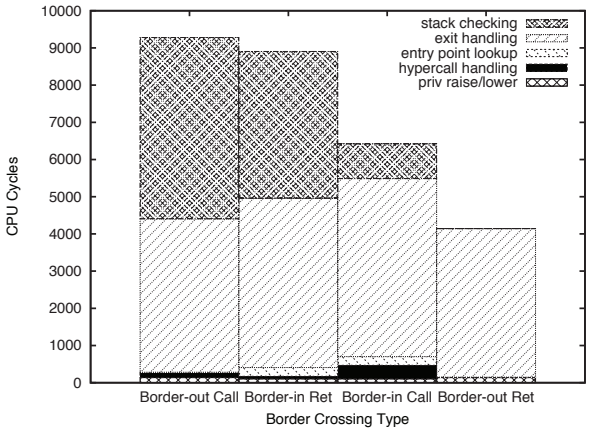


Figure 3: Privilege change cost with stack integrity checks.

Deinitialization The Christoization processing inserts a deinitialization hypercall as the last thing the module executes. After validating the hypercall's location, the state machine lowers privilege, removes any special interception that is active, and remaps the module with guest-specified writability. Privilege will not change again unless the initialization hypercall is executed.

Suspicious activity The state machine detects suspicious activity by noting privilege changing hypercalls at invalid locations, shadow or nested page faults indicating attempts to write the module code, and stack hash mismatches. Our default behavior is simply to lower privilege when these occur, and continue execution. Other reactions are, of course, possible.

5 Evaluation

We now consider the costs of the guarded module system, independent of any specific guarded module that might drive it, and any selective privilege-enabled VMM component it might drive. We focus on the costs of border crossings and their breakdown. The most important contributors to the costs are VM exit/entry handling and the stack validation mechanism.

All measurements were conducted on a Dell PowerEdge R415. This is a dual-socket machine, each socket comprising a quad-core, 2.2 GHz AMD Opteron 4122,

giving a total of 8 physical cores. The machine has 16 GB of memory. It runs Fedora 15 with a stock Fedora 2.6.38 kernel. Our guest environment uses a single virtual core that runs a BusyBox environment based on Linux kernel 2.6.38. The guest runs with nested paging, using 2 MB page mappings, with DVFS control disabled.

Figure 3 illustrates the overheads in cycles incurred at runtime. All cycle counts were averaged over 1000 samples. There are five major components to the overhead. The first is the cost of initiating a callback to lower or raise privilege. This cost is very small at around 100 cycles. The second cost, labeled "hypercall handling", denotes the cycles spent inside the hypercall handler itself, not including entry validations, privilege changes, or other processing involved with a VM exit. This cost is also quite small, and also typically under 100 cycles. "entry point lookup" represents the cost of a hash table lookup, which is invoked on border-ins when the instruction pointer is checked against the valid entry points that have been registered during guarded module initialization. The cost for this lookup is roughly 240 cycles. "exit handling" is the time spent in the VMM handling the exit outside of guarded module runtime processing. This is essentially the common overhead incurred by any VM exit. Finally, "stack checking" denotes the time spent ensuring control-flow integrity by validating the stack. This component raises the cost of a border crossing by 5000 cycles, mostly due to stack address translations and hash computations. Border-in calls are less affected due to the initial translation and recording of the entry stack pointer, while border-out rets are unaffected. Reducing the cost of this validation is the subject of on-going work.

The guarded module codebase consists of the compile-time tools, which comprise 223 lines of Perl, 260 lines of Ruby and the run-time elements added to the VMM. The latter are generally concentrated in an optional extension of 1007 lines of C that could be ported to other VMMs. Some changes to the VMM core were made to facilitate interrupt and exception interception and dispatch to the GEARS guarded module system. These include 178 lines of C.

6 Examples

We now consider two examples of using the guarded module functionality, drawn from the list in the introduction. In the first example, selectively-privileged PCI passthrough, the guarded module, and only the guarded module, is given direct access to a specific PCI device. We illustrate the use of this capability via a guarded version of a NIC driver. In our second example, selectively-privileged `mwait`, the guarded module, and only the guarded module, is allowed to use the `mwait` instruction. We illustrate the use of this capability via guarded module that adaptively replaces the kernel idle loop with a more efficient `mwait` loop when it is safe to do so.

We conducted all measurements in this section with the configuration described in Section 5.

6.1 Selectively privileged PCI passthrough

Like most VMMs, Palacios has hardware passthrough capabilities. Here, we use its ability to make a hardware PCI device directly accessible to the guest. This consists of a generic PCI front-end virtual device ("host PCI device") , an interface it can use to acquire and release the underlying hardware PCI device on a given host OS ("host PCI interface"), and an implementation of that interface for a Linux host.

A Palacios guest's physical address space is contiguously allocated in the host physical address space. Because PCI device DMA operations use host physical addresses, and because the guest programs the DMA engine using guest physical addresses it believes start at zero, the DMA addresses the device will actually use must be offset appropriately. In the Linux implementation of our host PCI interface, this is accomplished using an IOMMU: acquiring the device creates an IOMMU page table that introduces the offset. As a consequence, any DMA transfer initiated on the device by the guest will be constrained to that guest's memory. A DMA can then only be initiated by programming the device, which is restricted to the guarded module. This restriction also prevents DMA attacks on the module that might originate from the guest kernel.

A PCI device is programmed via control/status registers that are mapped into the physical memory and I/O port address spaces through standardized registers called BARs. Each BAR contains a type, a base address, and a size. Palacios's host PCI device virtualizes the BARs (and other parts of the standardized PCI device configuration space). This lets the guest map the device as it pleases. For a group of registers mapped by a BAR into the physical memory address space, the mapping is implemented using the shadow or nested page tables to redirect memory reads and writes. For a group of registers mapped into the I/O port space, there is no equivalent to these page tables, and thus the mappings are implemented by I/O port read/write hooks. When the guest executes an IN or OUT instruction, an exit occurs, the hook is run, and the handler simply executes an IN or OUT to the corresponding physical I/O port. If the host and guest mappings are identical, the ports are not inter-

cepted, allowing the guest to read/write them directly.

Direct guest access to network hardware is not a new idea. However, the focus of recent work in this area is on providing protection between guests [25, 24]. We allow protection of a *VMM-provided* driver *within* a guest.

We extended our host PCI device to support selective privilege; in the terminology of Section 4.2, it is now a selective privilege-enabled VMM component. In this mode of operation, virtualization of the generic PCI configuration space of the device proceeds as normal. However, at startup, BAR virtualization ensures that the address space regions of memory and I/O BARs are initially hooked to stub handlers. The stub handlers simply ignore writes and supply zeros for reads. This is the *unprivileged mode*. In this mode, the guest sees the device on its PCI bus, and can even remap its BARs as desired, but any attempt to program it will simply fail because the registers are inaccessible. In selectively privileged operation, the host PCI device also responds to callbacks for raising and lowering privilege. Raising privilege switches the device to *privileged mode*, which is implemented by remapping the registers in the manner described earlier, resulting in successful accesses to the registers. Lowering privilege switches back to unprivileged mode, and remaps the registers back to the stubs. Privilege changes happen on a per-core basis.

While the above description is complex, it is important to note that only about 60 lines of code were needed to add selectively privileged operation to our existing PCI passthrough functionality. Combined with the rest of the guarded module system, the selectively privileged host PCI device permits fully privileged access to the underlying device within a guarded module, but disallow it otherwise.

Making a NIC driver into a guarded module As an example, we used the guarded module system to generate a guarded version of an existing NIC device driver within the Linux tree, specifically the Broadcom BCM5716 Gigabit NIC. No source code modifications were done to the driver or the guest kernel. We Christoize this driver, creating a kernel module that we can later inject into the untrusted guest. The border control state machine in Palacios pairs this driver with the selectively privileged PCI passthrough capability. Recall that Christoization is almost entirely automated, so the result is an unmodified device driver, executing in the guest, having direct access to the NIC, while nothing else in the guest does.

The NIC uses exactly one BAR to define a 32 MB region of the memory address space. Raising and lowering privilege amounts to editing the shadow or nested page tables to remap these addresses. Assuming 2 MB superpages and suitable alignment, the system will adjust 16 page table entries when changing privilege.

Packet Sends	
Border-in	1.06
Border-out	1.06
Border Crossings / Packet Send	**2.12**
Packet Receives	
Border-in	4.64
Border-out	4.64
Border Crossings / Packet Receive	**9.28**

Figure 4: Border crossings per packet send and receive for the NIC example.

Overheads Compared to simply allowing privilege for the entire guest, a system that leverages guarded modules incurs additional overheads. Some of these overheads are system-independent, and were covered in Section 5. The most consequential component of these overheads is the cost of executing a border-in or border-out, each of which consists of a hypercall or exception interception (requiring a VM exit) or interrupt/exception injection detection (done in the context of an in-progress VM exit), a lookup of the hypercall's address, a stack check or record, conducting a lookup to find the relevant privilege callback function, and then the cost of invoking that callback.

We now consider the system-dependent overhead for the NIC. There are two elements to this overhead: the cost of changing privilege and the number of times we need to change privilege for each unit of work (packet sent or received) that the module finishes. The cost of raising privilege for the NIC is 4800 cycles (2.2 μs), while lowering it is 4307 cycles (2.0 μs).

Combining the system-independent and system-dependent costs, we expect that a typical border crossing overhead, assuming no stack checking will consist of about 3000 cycles for VM exit/entry, 4000 cycles to execute the border control state machine, and about 4500 cycles to enable/disable access to the NIC. These 11500 cycles comprise 5.2 μs on this machine. Stack checking would add an average of about 4500 cycles, leading to 16000 cycles (7.3 μs).

To determine the number of these border crossings per packet send or receive, we counted them while running the guarded module with a controlled traffic source (ttcp) that allows us to also count packet sends and/or receives. Dividing the counts gives us the average. There is variance because the NIC does interrupt coalescing.

Figure 4 shows the results of this analysis for the NIC. Sending requires on the order of 2 border crossings (privilege changes) per packet, while receiving requires on the

order of 9 border crossings per packet. Note that many of the functions that constitute border crossings are actually leaf functions defined in the kernel. This indicates that we could further reduce the overall number of border crossings per packet by pulling the implementations of these functions into the module itself.

6.2 Selectively privileged mwait

Recent x86 machines include a pair of instructions, `monitor` and `mwait`, that can be used for efficient synchronization among processor cores. The `monitor` instruction indicates an address range that should be watched. A subsequent `mwait` instruction then places the core into a suspended sleep state, similar to a `hlt`. The core resumes executing when an interrupt is delivered to it (like a `hlt`), or when another core writes into the watched address range (unlike a `hlt`). The latter allows a remote core to wake up the local core without the cost of an inter-processor interrupt (IPI). One example of such use is in the Linux kernel's idle loop.

In Palacios, and other VMMs, we cannot allow an untrusted guest to execute `hlt` or `mwait` because the guest runs with physical interrupts disabled. A physical interrupt is intended to cause a VM exit followed by subsequent dispatch of the interrupt in the VMM. If an `mwait` instruction were executed in the guest under uncontrolled conditions, it could halt the core indefinitely. This precludes the guest using the extremely fast intercore wakeup capability that `mwait` offers.

Under controlled conditions, however, letting the guest run `mwait` may be permissible. When no other virtual core is mapped to the physical core (so we can tolerate a long wait) and we have a watchdog that will eventually write the memory, the guest might safely run an `mwait`. To achieve these controlled conditions requires that we limit the execution of these instructions to code that the VMM can trust and that this code only execute `mwait` when the VMM deems it safe to do so. A malicious guest could use an unrestricted ability to execute `mwait` to launch a denial-of-service attack on other VMs and the VMM. We enforce this protection and adaptive execution by encapsulating the `mwait` functionality within the safety of a guarded module.

Adding selectively-privileged access to `mwait` to Palacios was straightforward, involving only a few lines of code. We then implemented a tiny kernel module that interposes on Linux's default idle loop, specifically modifying `pm_idle`, a pointer to the function that points to the idle implementation. Our module points this to a function internal to itself that dispatches either to an `mwait`-based idle implementation within the module or to the original idle implementation, based on a flag in protected memory that is shared with Palacios. Palacios sets this flag when it is safe for the module to use `mwait`. In these situations, the guest kernel enjoys much faster wake-ups of the idling core.

To assure that only our module can execute `mwait` we transform it into a guarded module using the techniques outlined earlier in the paper. A border-in to our module occurs when Linux calls its idle loop. If the border-in succeeds, Palacios stops intercepting the use of `mwait`. When control leaves the module, a border-out occurs, and Palacios resumes intercepting `mwait`. If code elsewhere in the guest attempts to execute these instructions, they will trap to the VMM and result in an undefined opcode exception being injected into the guest.

This proof-of-concept illustrates how the VMM can use guarded modules to safely adapt the execution environment of a VM to changing conditions.

7 Conclusions and future work

We presented the design, implementation, and evaluation of a system for guarded modules. The system allows the VMM to add modules to a guest kernel that have higher privileged access to physical hardware and the VMM while protecting these guarded modules and access to their privileges from the rest of the guest kernel. Our system is based on joint compile-time and runtime techniques that bestow privilege only when control flow enters the guarded module at verified locations. We demonstrated two example uses of the guarded module system. The first is passthrough access to a PCI device, for example a NIC, that is limited to a designated guarded module (a device driver). The guest kernel can use this guarded module just like any other device driver. We further demonstrated selectively privileged use of the `monitor` and `mwait` instructions in the guest, which could wreak havoc if their use was not constrained to a guarded module that cooperates with the VMM.

Our ongoing and future work lies along two lines. First, we will explore methods that can further enhance the performance of this system. Building upon the analysis of Section 6, we plan to further study methods by which we can reduce the cost and number of border crossings needed for a specific module. As previously mentioned, we are investigating an expansive linking process in which kernel functions invoked by the guarded module are incrementally incorporated into the module itself. Our second line of investigation is in designing other virtualization services that could be simplified or enabled by employing guarded modules.

References

[1] CARBONE, M., CONOVER, M., MONTAGUE, B., AND LEE, W. Secure and robust monitoring of virtual machines through guest-assisted introspection. In *Proceedings of the 15th International Conference on Research in Attacks, Intrusions, and Defenses (RAID 2012)* (September 2012).

[2] CHECKOWAY, S., AND SHACHAM, H. Iago attacks: Why the system call api is a bad untrusted rpc interface. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2013)* (March 2013).

[3] CHEN, X., GARFINKEL, T., LEWIS, E. C., SUBRAHMANYAM, P., WALDSPURGER, C. A., BONEH, D., DWOSKIN, J., AND PORTS, D. R. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2008)* (March 2008).

[4] CRISWELL, J., DAUTENHAHN, N., AND ADVE, V. Virtual ghost: Protecting applications from hostile operating systems. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2014)* (March 2014).

[5] HALE, K., XIA, L., AND DINDA, P. Shifting GEARS to enable guest-context virtual services. In *Proceedings of the 9th International Conference on Autonomic Computing (ICAC 2012)* (September 2012).

[6] HOFMANN, O. S., KIM, S., DUNN, A. M., LEE, M. Z., AND WITCHEL, E. Inktag: Secure applications on an untrusted operating system. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2013)* (March 2013).

[7] HU, L., SCHWAN, K., GULATI, A., ZHANG, J., AND WANG, C. Net-cohort: Detecting and managing vm ensembles in virtualized data centers. In *Proceedings of the 8th International Conference on Autonomic Computing (ICAC 2012)* (September 2012).

[8] JOSHI, A., KING, S. T., DUNLAP, G. W., AND CHEN, P. M. Detecting past and present intrusions through vulnerability-specific predicates. In *Proceedings of the 20th ACM Symposium on Operating System Principles (SOSP 2005)* (October 2005).

[9] LANGE, J., DINDA, P., HALE, K., AND XIA, L. An introduction to the palacios virtual machine monitor—release 1.3. Tech. Rep. NWU-EECS-11-10, Department of Electrical Engineering and Computer Science, Northwestern University, October 2011.

[10] LANGE, J., PEDRETTI, K., DINDA, P., BRIDGES, P., BAE, C., SOLTERO, P., AND MERRITT, A. Minimal overhead virtualization of a large scale supercomputer. In *Proceedings of the 2011 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2011)* (March 2011).

[11] LANGE, J., PEDRETTI, K., HUDSON, T., DINDA, P., CUI, Z., XIA, L., BRIDGES, P., GOCKE, A., JACONETTE, S., LEVENHAGEN, M., AND BRIGHTWELL, R. Palacios and kitten: New high performance operating systems for scalable virtualized and native supercomputing. In *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2010)* (April 2010).

[12] LEVASSEUR, J., UHLIG, V., STOESS, J., AND GÖTZ, S. Unmodified device driver reuse and improved system dependability via virtual machines. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2004)* (December 2004).

[13] LIE, D., THEKKATH, C. A., AND HOROWITZ, M. Implementing an untrusted operating system on trusted hardware. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP 2003)* (October 2003).

[14] LIU, J., HUANG, W., ABALI, B., AND PANDA, D. High performance vmm-bypass i/o in virtual machines. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC 2006)* (May 2006).

[15] LU, L., ZHANG, H., JIANG, G., CHEN, H., YOSHIHIRA, K., AND SMIRNI, E. Untangling mixed information to calibrate resource utilization in virtual machines. In *Proceedings of the 8th International Conference on Autonomic Computing (ICAC 2011)* (June 2011).

[16] MAO, Y., CHEN, H., ZHOU, D., WANG, X., ZELDOVICH, N., AND KAASHOEK, M. F. Software fault isolation with api integrity and multi-principal modules. In *Proceedings of the 23rd ACM Symposium on Operating System Principles (SOSP 2011)* (October 2011).

[17] MCCUNE, J. M., LI, Y., NING, Q., ZHOU, Z., DATTA, A., GLIGOR, V., AND PERRIG, A. Trustvisor: Efficient tcb reduction and attestation. In *Proceedings of the 31st IEEE Symposium on Security and Privacy (SP 2010)* (May 2010).

[18] MCCUNE, J. M., PARNO, B., PERRIG, A., REITER, M. K., AND ISOZAKI, H. Flicker: An execution infrastructure for tcb minimization. In *Proceedings of the 3rd ACM European Conference in Computer Systems (EuroSys 2008)* (April 2008).

[19] RAJ, H., AND SCHWAN, K. High performance and scalable i/o virtualization via self-virtualized devices. In *Proceedings of the 16th IEEE International Symposium on High Performance Distributed Computing (HPDC 2007)* (July 2007).

[20] SESHADRI, A., LUK, M., QU, N., AND PERRIG, A. Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP 2007)* (October 2007).

[21] SHARIF, M. I., LEE, W., CUI, W., AND LANZI, A. Secure in-vm monitoring using hardware virtualization. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS 2009)* (November 2009).

[22] SWIFT, M. M., BERSHAD, B. N., AND LEVY, H. M. Improving the reliability of commodity operating systems. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP 2003)* (October 2003).

[23] WANG, L., XU, J., AND ZHAO, M. Application-aware cross-layer virtual machine resource management. In *Proceedings of the 9th International Conference on Autonomic Computing (ICAC 2012)* (September 2012).

[24] WILLMANN, P., RIXNER, S., AND COX, A. L. Protection strategies for direct access to virtualized i/o devices. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC 2008)* (June 2008).

[25] WILLMANN, P., SHAFER, J., CARR, D., RIXNER, S., COX, A., ZWAENEPOEL, W., AND ZWAENEPOEL, W. Concurrent direct network access for virtual machine monitors. In *Proceedings of the 13th IEEE International Symposium on High Performance Computer Architecture (HPCA 2007)* (February 2007).

[26] XU, J., ZHAO, M., FORTES, J., CARPENTER, R., AND YOUSIF, M. On the use of fuzzy modeling in virtualized data center management. In *Proceedings of the 4th International Conference on Autonomic Computing (ICAC 2007)* (June 2007).

Active Control of Memory for Java Virtual Machines and Applications

Norman Bobroff, Peter Westerink, Liana Fong
IBM T.J. Watson Research Center
Yorktown Heights, NY 10598, USA
{*bobroff, peterw, llfong*}*@us.ibm.com*

Abstract

A controller is implemented to manage memory as an elastic resource similar to computing cycles for Java applications. The controller actively arbitrates constrained memory between collocated JVMs in response to demand. A key aspect of the work is that JVM metrics are used as proxies for application KPIs so that application performance instrumentation and modeling are not required. A metric corresponding to the *allocation rate* of memory is derived from the JVM metrics and established as the measure of application performance and is used as the effective feedback mechanism to the controller. The controller is based on a fair share policy in which memory is distributed to equalize the marginal performance value to all JVMs. The design is tested for effectiveness and stability using the suite of SPECjvm2008 and SPECjbb2005 benchmarks.

1 Introduction

Matching real memory and CPU resources to the time varying memory-processor demand footprint of applications is an important element in systems performance management. Active sharing of processors between applications within and across virtual machines (VMs) in response to demand is a mature feature of the operating systems and hypervisors. Active sharing of memory (ASM) is the analogous capability where physical memory pages move seamlessly between applications and across virtual machines to satisfy demand. This improves system wide memory utilization, or alternatively increases the application density or workload intensity hosted on a compute system. ASM is sometimes referred to as logical memory overcommit as it reduces the total amount of memory necessary in a system with time varying workloads from the sum of the maximum demand of each workload to the maximum of the sum of the workloads. ASM is distinguished from paging which requires saving and restoring state in order to reuse pages from processes or VMs. Exploiting ASM requires the ability to identify unused memory in applications and operating systems and (re)map those pages to collocated applications, or move them to another VM on a common hypervisor. This function is widely available at the VM-hypervisor layer in the commercial space. But support at the application layer has been lagging as traditional application design and coding practice has not emphasized the need to dynamically return memory from the process space to the OS.

Widespread use of the Java Virtual Machine (JVM) as a server application platform creates an opportunity to extend the scope of ASM into the application layer. Emerging JVM technologies such as heap ballooning [2, 3] and dynamic heap sizing [4] provide mechanisms to release committed memory from the virtual heap space. Given these advances it is an appropriate time to visit the architecture and control functions required for an automatic ASM solution that focuses on Java applications.

This paper describes two novel aspects of JVM memory management: JVM metrics are shown to be suitable proxies for application based key performance indicators (KPI); and JVM heap memory is actively sized in response to resource changes and workload variability by equalizing the value of memory as indicated by the JVM metrics. Memory intensive benchmarks from the SPECjvm2008 [7] suite and SPECjbb2005 [6] are used to correlatefJVM metrics and application KPIs, and to evaluate the control system.

2 Background and Related Work

Figure 1 shows the platform used to investigate active memory sharing (AMS) in a virtual environment. From a logical perspective, the figure is a tree with application JVMs at the top, and the hypervisor memory pool of a physical machine (PM) at the root. One or more

collocated JVMs is hosted by an operating system (OS). Each OS apportions its memory pool to processes (JVMs and other applications), free pools, and system cache. In turn, the operating systems share the common physical platform memory in the hypervisor pool. Memory flows slowly down the tree to the OS and hypervisor pools on the non-critical path, while the flow of memory up the tree is on the critical path. This paper focuses on the upper layer, fairly apportioning memory between collocated JVM based applications in response to workload changes, memory demand, and changes in OS memory.

Commercial methods are becoming available to release unused pages backing the JVM heap memory: VMWare's EM4J [8] applies a balloon mechanism that plugs into the JVM and is leveraged in the memory control work of Ginko [2]. Direct heap resizing is available in the IBM J9 JVM since version 7.0 [4]. Section 3.2 describes in detail how we leverage this JVM control knob, *MaxHeapSize* to actively control heap memory.

Recent work has studied active JVM memory sizing. Ginkgo [2] implements an application driven memory overcommitment system. Salomie [3] designs an application level ballooning controller in Xen-based environment. CRAMM [9] enables dynamically choosing of JVM heap sizes to meet workload demand, while avoiding latency in paging. QoE-JVM [5] uses an economic model for active heap sizing in the Jikes research JVM.

3 Approach

3.1 JVM metrics as proxies for application performance

There are several reasons to use JVM metrics as proxies for application performance as developed in Section 4. Most Java processes and applications do not maintain internal measures of their rate of progress. When available the interpretation of the KPI's often requires domain specific knowledge. Processing services often support a mix of incoming request types each with its own resource requirements. A shift in workload composition can change KPI's in a way that needs to be understood by the controller logic.

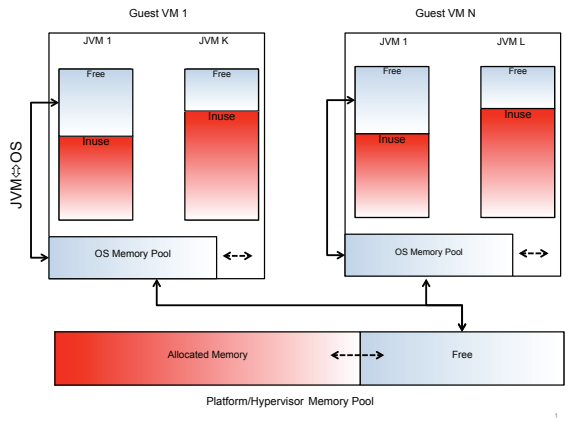


Figure 1: System view of elastic memory

End-to-end application performance depends components other than the JVM. For example, the database tier may be slowed because of insufficient OS system buffers. Note that the effect of a slow database on a Java application tier is manifest in JVM metrics such as rate of object allocation since the application can't make progress. Here, the local controller gives the JVM less memory than if the JVM tier is running at full load. This released memory makes its way to the database server via the flow of Figure 1.

3.2 JVM direct page releasing mechanism

The JVM memory control knob leveraged in this work is the *MaxHeapSize* parameter of IBM's J9 JVM. At startup J9 reserves a contiguous region of virtual process space for its heap sized by the command line argument *-Xmx* which is exposed through JMX as the immutable *MaxHeapSizeLimit* . The J9 JVM maintains a second, soft, heap maximum setting called the *MaxHeapSize* whose operation is described in Sciampacone [4]). Basically, MaxHeapSize can be set via JMX at any time during JVM execution to a value less than MaxHeapSizeLimit. When actual heap used drops below MaxHeapSize the JVM attempts to resize the heap using MaxHeapSize as the new limit.

4 JVM Metrics and Application Performance

This section analyzes the correlation between JVM metrics and workload intrinsic performance (e.g., business operations per second-bops) for memory intensive benchmarks culled from SPECjvm2008 and SPECjbb2005. The goal is to utilize the JVM metrics as proxies for application KPI's to correctly size or arbitrate JVM memory.

4.1 Metrics collected from the JVM

Several JVM metrics are exposed through the JMX API, providing a measure of how the application benefits from memory.

- Mem-freed - Cumulative number of bytes collected by the GC since a JVM startup.
- Heap-inuse - Current amount of the heap memory containing objects with live references.

Benchmark	Commit(MB)	Alloc Rate(MB/s)
SPECjbb2005	2000	2400
compiler.compiler	5000	400
derby	2000	1300
scimark.lu.large	2000	20
scimark.sort.large	1000	7
scimark.sparse.large	2000	20
scimark.fft.large	1800	20
crypto.aes	400	300
xml.validation	150	600
xml.xform	200	600
serial	420	300

Table 1: Max memory and allocation rate in benchmarks

- Heap-committed (hpCom) - Physical memory mapped to the virtual heap.
- GC CPU - The fraction of the system CPU cycles spent in GC. A decrease in GC CPU often provides an indication of whether adding memory is benefiting the application.
- Collection rate (coll-rate) - The number of GCs reported by the JVM over the sampling interval. The algorithm that determines when GC occurs is internal to the JVM.
- Allocation rate (alloc-rate) - This measure is is derived from the inuse-heap and mem-freed metrics. It is the rate of memory allocated during an interval and is computed in the sample interval $[t_1, t_2]$ using the allocated bytes;
  alloc-rate = [heap-inuse(t_2) - heap-inuse(t_1) + mem-freed(t_2) - mem-freed(t_1)] / [($t_2 - t_1$]

Allocation rate depends jointly on application demand, and the ability to satisfy the demand. Furthermore, it is a complementary measure to the GC CPU and GC collection rate metrics. If the JVM heap allocator is slowed down because of low memory, that latency translates at the application code to time spent in the Java 'new' memory allocation operator.

4.2 Memory intensive workloads

Two benchmark groups are used to establish the correlation of the JMX metrics and workload performance. SPECjvm2008 contains over 20 individual benchmarks that cover a wide range of applications. Of these, the 10 which use more than 128MB of committed heap are considered memory intensive. The excluded set in this group use less than 50MB opt committed heap. SPECjbb2005 is representative of a traditional transactional workload.

The benchmarks selected are summarized in Table 1. They cover a range of committed heap size from 128MB to 5GB, and allocation rates from 10MB/s to over 1GB/s. All benchmarks are CPU intensive and multithreaded.

4.3 Results

The correlation between the SPECjvm2008 benchmark KPIs and the JVM metrics is explored as a function of the MaxHeapSize (MB) parameter as the JVM control knob. Figures 2, 3, 4, and 5, are representative data sets the workloads. In each figure, the SPECjvm2008 performance number (bops) recorded during the runs is shown in (a). The corresponding JVM metric averages are displayed in subplots: (b) - GC CPU (%); (c) - collection-rate (ct/s); (d) - allocation rate (MB/s); and (e) - committed physical memory (MB). These data also indicate the open loop response the MaxHeapSize input control.

The *derby* database benchmark of Figure 2(a) typifies workloads exhibiting the *threshold* memory pattern. Here, most of the gain in application performance occurs within a critical heap size, after which the value of adding memory is low.

Derby also illustrates an interesting behavior with respect to committed memory within the threshold pattern. The region at the right hand side of Figure 2(e) shows that as the heap maximum is increased beyond the critical region, the JVM continues to commit real memory and grow the heap well into the low benefit region. Doubling the memory by incrementing the MaxHeapSize control value 1GB provides less than 2% performance improvement. This *memory greedy* pattern is also observed in the scientific benchmarks grouped together in Table 1. For example, in the large FFT benchmark of Figure 3, the JVM commits about 1GB of memory beyond the point of improving performance. Systems executing this workload pattern benefit from limiting MaxHeapSize to avoid consuming physical memory.

The *xml.validation* benchmark (Figure 4(a)) also typifies the *threshold pattern*, but is not memory greedy. Figure 4 shows that the JVM only committed 150MB.

In contrast, the compiler benchmark of Figure 5 benefits proportionally to the maximum heap memory control parameter. The behavior is monotonic, but there is

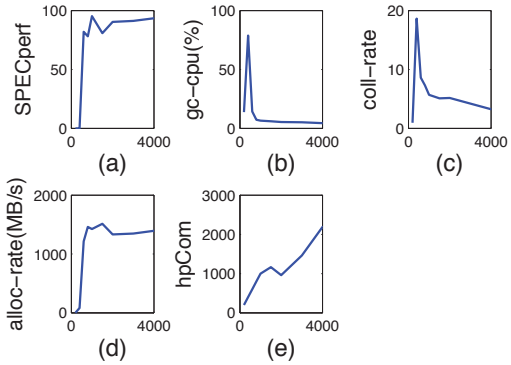


Figure 2: The SPECjvm derby workload typifies the threshold pattern and is memory greedy

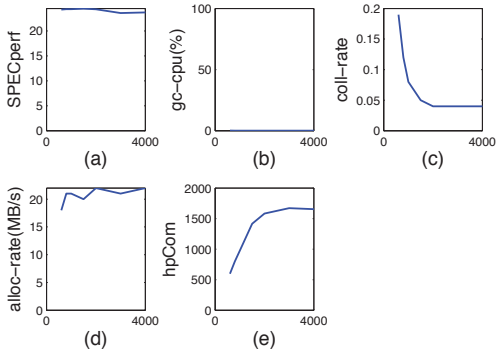


Figure 3: Scientific benchmarks such as this FFT tend to fit the memory greedy pattern.

a region of less performance gain between around 1GB and 3GB sandwiched between larger slopes of the performance - heap memory relation. The compiler performance plateaus at about 4.8GB from our experiments, just beyond the right end of the plot.

Additional insight is obtained by correlating the JVM metrics with the SPECjvm2005 and SPECjvm2008 KPI (e.g.bops). The correlation coefficients are shown in Figure 6 for the alloc-rate, gc-cpu, and coll-rate measures. The weakest correlations for all three JVM metrics are observed for the scientific benchmarks typified in the FFT benchmark of Figure 3. The lower correlation does not imply that the JVM metrics are not suited as input data to the active memory control system. In the case of the *scimark.fft.large* benchmark, the data of Figure 3 are flat and so the jitter contributes significantly to the correlation calculation.

These experiments suggest that decisions about the benefit to the application of additional memory be made on the basis of the observed change in JVM metrics as memory is added to the JVM, rather than on the metric values themselves. Consider the *threshold pattern* of Figure 2. Adding memory clearly benefits the application, as indicated by the concurrent improvement in allocation rate.

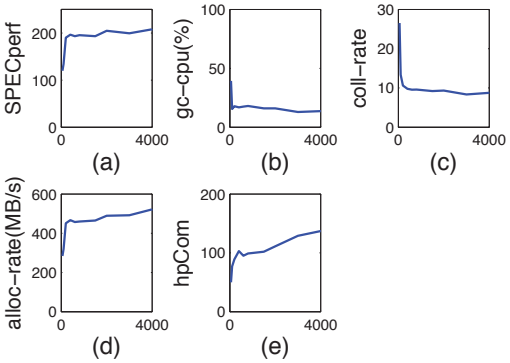


Figure 4: The SPECjvm2008 xml.validation benchmark exhibits a threshold pattern, but is not memory greedy

Figure 6 suggests the *allocation rate* is the single most consistent indicator of workload performance. The only benchmark where its correlation is significantly depressed relative to the other metrics is for the sci.lu.lg benchmark. Closed loop studies in Section 5 further validate the choice of *allocation rate* as the best single proxy for application performance.

5 Active JVM Memory Control

The objective of the memory controller is to leverage the JVM performance metrics of the prior section (esp. allocation-rate) to *fair share* the available memory between collocated JVMs. The *fair sharing* condition defines the distribution of memory between JVMs at a given workload so that: equal changes in the MaxHeapSize of each JVM result in equal changes in the relative performance of each JVM. The *relative performance slope* (S) for each JVM (j) is defined as the slope of the curve of the application performance (P_j) against MaxHeapSize, normalized by the performance value:

$$S_j = \frac{\Delta P_j}{\Delta MaxHeapSize_j} \times \frac{1}{P_j}.$$

The controller attempts to actively set $MaxHeapSize_j$ such that S_j is the same for all j.

JVM metrics are used to measure the application performance P_j. Section 4 identified *allocation rate* as a strong candidate for an application performance proxy. The open loop and offline data are now used to evaluate the JVM metrics in our slope equalizing algorithm. For example, the Specjvm2008 benchmark KPI data in Figures 5(a) and 4(a) is compared against the JVM metrics of Figures 5(b-d) and 4(b-d) in the algorithm to establish which single metric compares best to the fair sharing point given by the actual benchmark KPI numbers.

Figure 7 illustrates the equalization of relative performance slopes using data from the SPECjvm2008 *xml.validation* and *compiler.compiler* benchmarks (Figures 4 and 5). The horizontal and vertical axes are the

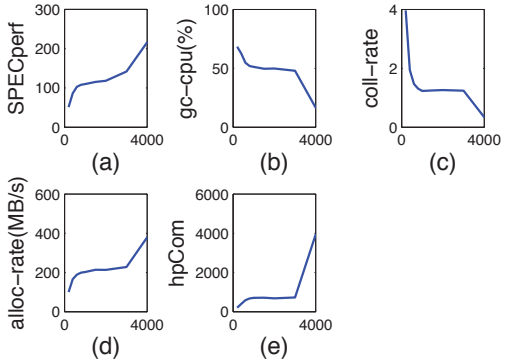


Figure 5: SPECjvm2008 compiler.compiler benefits up to about 5GB of memory

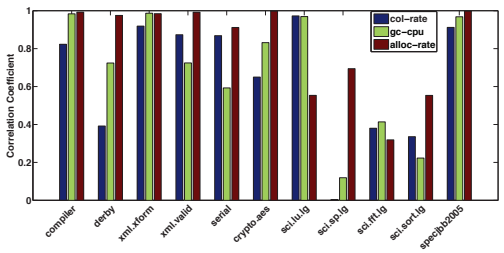


Figure 6: Correlation between SPEC performance and

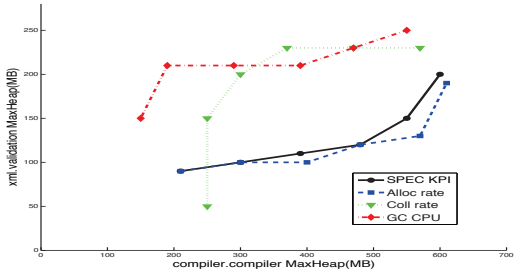


Figure 7: Memory balancing results using measured data for 3 JVM metrics and the SPECjvm performance values.

target value of the MaxHeapSize control parameter computed by the algorithm for each application JVM. Each data point represents the MaxHeapSizes that equalize the S_j for a given total memory. The total memory varies from 200MB on the left to 800MB on the right. The solid line labeled *SPEC KPI* is the reference curve showing the ideal apportionment using the application (i.e.SPECjvm2008) KPIs. The three other lines correspond to the GC-CPU, collection rate, and allocation rate metrics. The data show that the allocation-rate metric produces the closest agreement with the SPEC KPIs. This result supports the correlation analysis of Figure 6. Similar results are achieved using other workloads of Table 1.

5.1 Controller architecture

Figure 8 is a component diagram of the measure-analyze-control cycle that tracks workload memory demand and actively sets the *MaxHeapSize* parameter of each JVM. On the right of the figure is the *data collector* which uses JMX to poll the data from the JVM. The typical polling interval is 5 seconds.

The JVM metrics are fed into the control module on the left which has three logical components: the slope evaluator; the *Compute Next MaxheapSize* module that estimates the next MaxHeapSize value based on the current state; and the dither function. The data collector and controllers for collocated JVMs run in a single lightweight JVM process use less than 0.1% CPU and 20MB of memory.

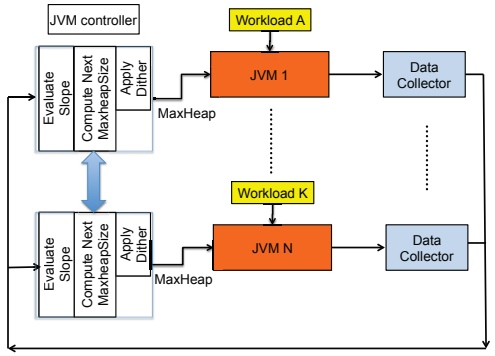


Figure 8: Main components of the memory controller.

5.2 Evaluating relative performance

A key function of the controller is to measure the slope of the allocation rate against the MaxHeapSize parameter at the current workload. To accomplish this, the controller modulates the MaxHeapSize parameter about its current target value. In this *Dithering* [10] technique, the MaxHeapSize is varied faster than the system response through a range about the target MaxHeapSize. The JVM metrics are sampled at limit points of this dithering range. Expanding the dither range too far results in oscillations in JVM performance, while reducing it yields a slower control speed. Empirically, a reasonable tradeoff is achieved with the lower point at 80% of the current target MaxHeapSize, and the upper point at 120%.

Figure 9 illustrates operation of the collection of data using dithering when running the *derby* benchmark as memory is removed from the system. Each subfigure is a snapshot showing the allocation rates measured at the ends of the dither range, and at the current MAxHeapSizeTarget. The three dither points are acquired on sequential measurement cycles about 5 seconds apart. This means that in a snapshot the three points are not necessarily at 80%, 100% , and 120% of the target MaxHeapSize as the target may have changed at each measurement cycle. Consequently, the three measured dither-points in the curve window may not lie on a locally convex curve. This situation is improved by relying more on the latest measurements than on older ones.

Figure 9 shows there are critical and noncritical regions of control. In the critical region, at the bottom of the figure, the slope is steep indicating the high value of additional memory to the application. In the noncritical region at the top of the figure, memory is not as valuable. Fortunately, the main difficulties caused by noise and jitter in measuring slope occur in the noncritical region of controller operation where the slope is low.

5.3 Memory balancing methodology

The *'Compute next MaxHeapSize'* module of Figure 8 determines the next set of target MaxHeapSize values to input into the JVMs based on the current system state based on the following procedure:

1. Check the available OS memory - If it has been modified, that memory is apportioned to the JVMs according the principle of equalizing the slopes.
2. Adjust the target MaxHeapSize - The current algorithm uses an iterative, greedy procedure to estimate the new set of HeapSizeMax values that equalize the slopes. At each step, memory is moved from the JVM with the lowest slope to the JVM with the steepest slope. The iterative computation is ended under either of two conditions: i) for any JVM, memory is only changed when within the upper and lower dither points; ii) the deviation from the equal slope condition no longer improves.
3. Select the dither points for each JVM- The direction and value of the dither is chosen for each JVM so that at any time the sum does not exceed the total available memory. Figure 11 shows the phase offset between the dithering pattern two located JVMs.
4. Execute the new MaxHeapSize target for each JVM.

5.4 Experimental results

The controller is evaluated using collocated JVMs running the SPECjvm2008 derby and the SPECjbb2005 transactional benchmarks. Figure 10 shows the allocation rate of each benchmark. The total memory constraint for the two JVMs is 1.5 GB.

The system state is held constant for 580 seconds with SPECjbb2005 using 10 warehouses. The variability in allocation rate during this period is due to different phases in the underlying workload. At 580s, the number

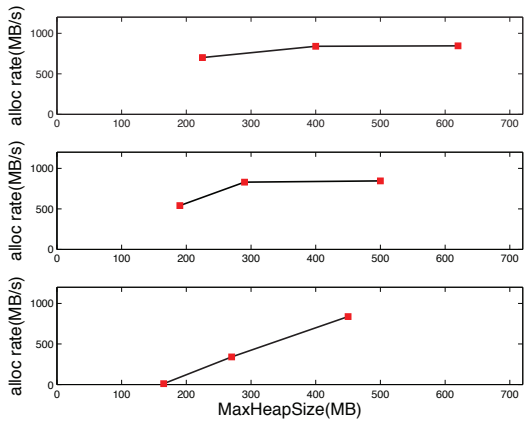


Figure 9: Snapshots of the dithering points.

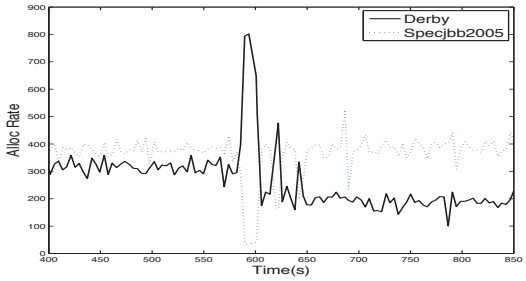


Figure 10: The allocation rate metric for collocated Derby and SPECjbb workloads.

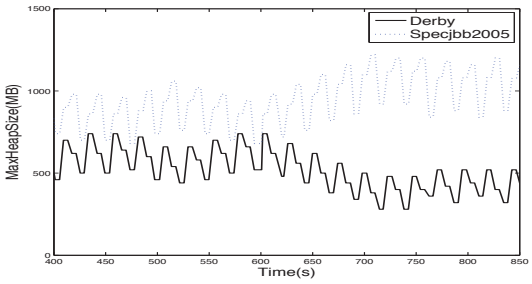


Figure 11: The MaxHeapSize control parameter including dithering for the Derby and SPECjbb workloads.

of SPECjbb2005 warehouses doubles to 20, introducing a step like change in the demand for memory. An 80s period of exponentially decaying oscillatory behavior in the control system occurs during the transition to the new operating point. The process CPU followed a similar pattern (not shown in the figure), with about a 13% shift from Derby to SPECjbb2005.

Figure 11 shows the corresponding control signal of MaxHeapSize sent to JVMs during runs. The dither signal is clearly seen imposed on the average MaxHeapSize control signal. Comparing the strength of the dither to the allocation rate data, Figure 10 indicates the dither does not affect the application performance, as desired. Experiments using the SPECjvm2008 compiler.compiler and SPECjbb2005 yielded comparable results.

6 Conclusion

JVM metrics are shown to work well as proxies for application KPIs so that application performance instrumentation and modeling are not required. This expands the applicability and ease of resource arbitration between collocated Java applications.

The control system of Section 5 is successfully applied in actively apportioning memory between collocated Java applications whose internal functions are largely unknown. Results show the response time to a step in workload intensity is of order of 80 seconds.

References

[1] CORDERO, M., CORREIA, L., AND ET EL. IBM PowerVM Virtualization Introduction and Configuration, June 2013.

[2] HINES, M. R., GORDON, A., SILVA, M., DA SILVA, D., RYU, K. D., AND BEN-YEHUDA, M. Applications know best: Performance-driven memory overcommit with ginkgo. In *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on* (2011), IEEE, pp. 130–137.

[3] SALOMIE, T.-I., ALONSO, G., ROSCOE, T., AND ELPHINSTONE, K. Application level ballooning for efficient server consolidation. In *Proceedings of the 8th ACM European Conference on Computer Systems* (2013), EuroSys '13, ACM, pp. 337–350.

[4] SCIAMPACONE, R., BURKA, P., AND MICIC, A. Garbage collection in WebSphere Application Server V8, Part 2: Balanced garbage collection as a new option. In *IBM WebSphere Developer Technical Journal* (August 2011).

[5] SIMAO, J., AND VIEGA, L. Qoe-jvm: An adaptive and resource-aware java runtime for cloud computing. *LNCS*, 7566 (2012), 566–583.

[6] STANDARD PERFORMANCE EVALUATION CORPORATION. SPECjbb2005 benchmark. http://www.spec.org/jbb2005.

[7] STANDARD PERFORMANCE EVALUATION CORPORATION. SPECjvm2008 benchmark. http://www.spec.org/jvm2008.

[8] VMWARE VFABRIC5 DOC CENTER. Elastic Memory for Java. http://pubs.vmware.com/vfabric5/index.jsp#em4j/about.html.

[9] YANG, T., BERGER, E. D., KAPLAN, S. F., AND MOSS, J. E. B. Cramm: Virtual memory support for garbage-collected applications. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (2006), OSDI '06, USENIX Association, pp. 103–116.

[10] ZAMES, G., AND SHNEYDOR, N. Dither in nonlinear systems. *IEEE TRANSACTIONS ON AUTONATIC CONTROL AC-21*, 5 (1976), 660–667.

Is Your Web Server Suffering from Undue Stress due to Duplicate Requests?

Fahad A. Arshad, Amiya K. Maji, Sidharth Mudgal, Saurabh Bagchi,
Purdue University
{*faarshad, amaji, smudgals, sbagchi*}*@purdue.edu*

Abstract

An important, if not very well known, problem that afflicts many web servers is duplicate client browser requests due to server-side problems. A legitimate request is followed by a redundant request, thus increasing the load on the server and corrupting state at the server end (such as, the hit count for the page) and at the client end (such as, state maintained through a cookie). This problem has been reported in many developer blogs and has been found to afflict even popular web sites, such as CNN and YouTube. However, to date, there has not been a scientific, technical solution to this problem that is browser vendor neutral. In this paper, we provide such a solution which we call GRIFFIN. We identify that the two root causes of the problem are missing resource at the server end or duplicated Javascripts embedded in the page. We have the insight that dynamic tracing of the function call sequence creates a signature that can be used to differentiate between legitimate and duplicate requests. We apply our technique to find unreported problems in a large production scientific collaboration web service called HUBzero, which are fixed upon reporting the problems. Our experiments show an average overhead of 1.29X for tracing the PHP-runtime on HUBzero across 60 unique HTTP transactions. GRIFFIN has zero false-positives (when run across HTTP transaction of size one and two) and an average detection accuracy of 78% across 60 HTTP transactions.

1 Introduction

The affliction of duplicated web requests: A duplicate web request occurs when the client web browser sends two requests for the same web page, the second being a redundant duplicate request. This affliction does not affect poorly run web sites alone. It afflicts two of the top 10 most visited sites — CNN and YouTube [15]. Our tests (with Chrome) show that at least 22 out of top 98 (on April 4, 2014) globally ranked Alexa [1] web sites give a duplicate request on accessing their homepages. On the academic side, we found that it affects HUBzero, a widely used open source software platform (originating from Purdue) for building powerful Web sites that support scientific discovery, learning, and collaboration [14].

Why do duplicate web requests happen? There are two root causes for the problem of duplicate web requests, which have been separately pointed out in many developer forums and blog posts [3, 4, 17]. The first cause is the incorrect way in which browsers handle missing component names, or empty tags, such as, `<img src="">`, `<script src="">`, and `<link href="">`. Equivalently, this could be caused by JavaScript which dynamically sets the `src` property on either a newly created image or an existing one: The most readable and comprehensive treatment of this first cause can be found in [3]. We will refer to this first root cause as *missing resource cause*. The second cause is the same Javascript being included in the page twice, or more number of times [15]. This is the root cause behind the duplicate web requests in CNN and YouTube. Two main factors increase the odds of a script being duplicated in a single web page: team size and number of scripts. It takes a significant amount of resources to develop a web site, especially if it is a top destination. In addition to the core team building the site, other teams contribute to the HTML in the page for things such as advertising, branding, and data feeds. With so many people from different teams adding HTML to the page, it is easy to imagine how the same script could be added twice, e.g., CNN and YouTube's main pages have 11 and 7 scripts respectively. We will refer to this second root cause as *duplicate script cause*.

How to fix the problem? The "missing resource cause" happens because the HTML specification, version 4 [5][1] is silent on this seemingly esoteric aspect. Even though the specification indicates that the `src` attribute should

[1] HTML4 is the latest version of the specification, except for a W3C "Candidate Recommendation" for HTML5 dated 04 February, 2014.

contain a Uniform Resource Identifier (URI), it fails to define the behavior when `src` does not contain a URI. Consequently, different browsers behave in different ways. For example, Internet Explorer (IE) sends the duplicate request to the directory of the page rather than the page itself, while Firefox and Chrome send the duplicate request to the page itself. Further, the behavior of different browsers for handling different missing resources is different, *e.g.*, IE does not initiate a duplicate request with missing `script` while Firefox and Chrome do. The overall approach to handling this could be to write server-side code that will catch a similar request arising close in time to the original request and correlated with finding a missing URI in a tag. However, due to the differences in browser behaviors and for different tags, this would lead to ungainly code, with case statements for a large number of different cases. An indirect evidence comes from the fact that though this problem has been known for a while (since at least 2009), this solution is seldom deployed. The "duplicate script cause" of course has no easy solution available currently. The solution is mainly process-based — enabling better communication and coordination between developers writing or using scripts to create web pages.

Our solution approach: In this paper, we present a general-purpose solution to the above problem, in a system called GRIFFIN[2]. By "general-purpose", we mean that the solution applies unmodified to all kinds of resources and browsers. The solution has at its heart the observation that the duplicate web requests cause a repeated signal, for some definition of "signal". The signal should be defined such that it can be easily traced in a production web server, without impacting computation or storage resources and without needing specialized code insertion. We find that the *function call depth* is the signal that satisfies these conditions, while preserving enough fidelity that the repeated sequence can be easily and automatically discerned. To automatically discern the repeated pattern, we use the simple-to-calculate autocorrelation function for the signal and at a lag, equal to the size of the web request (in terms of number of HTTP commands), GRIFFIN sees a spike in autocorrelation which it uses to flag the detection.

When tested over a wide range of buggy and non-buggy behavior, we find that GRIFFIN performs well with respect to both the detection and the false positive. We find that GRIFFIN has no false positive and an 80% detection accuracy. To make GRIFFIN feasible in real production settings, we adopt a mix of synchronous and asynchronous approaches, both without modifying the application's source code, or even needing access to the source code. Synchronously we capture the call stack depth, using a built-in functionality, in the tracing tool called SYSTEMTAP. Then, asynchronously, GRIFFIN calculates the autocorrelation function for various lags, filters the values, and flags a detection when the value exceeds a threshold. In addition to detection, GRIFFIN also provides some diagnostic insight, *i.e.*, gives an idea of the module where the root cause lies.

[2] GRIFFIN is a mythical creature with the front legs, wings, and head of a giant eagle, and the body, hind legs, and tail of a lion. It is often used to guard treasures.

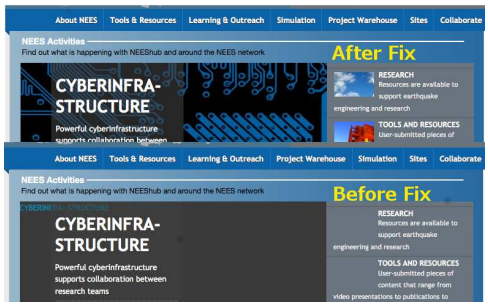


Figure 1: Duplicate bug-manifestation (with missing images) before and after the fix

2 Example Bug Case

Here we present a bug-case that was observed for the beta release of the main web portal of our NSF center called NEEScomm, meant for providing a cyberinfrastructure for earthquake engineers and scientists throughout the US `www.nees.org`. GRIFFIN was able to detect it before the code update made it to the production site, and thus avoided the duplicate request problem. On accessing the homepage, the images that appear as part of background were missing (Figure 1). Listing 1 presents the code modifications that fixed the problem (no duplicate requests seen from client). In Listing 1, `$slide->mainImage` variable does not resolves to the image `XYZ.jpg` location. Instead, it resolves to the NUL character. Manual inspection revealed that the images were missing. To verify, we hard-coded a valid image location and it fixed the duplicate problem. Listing 2 shows the runtime state of the rendered HTML in Firefox browser. On lines 3 and 10, the empty `url()` is observed, while on line 4, the `src` field in `<img>` tag having a value of "/" pinpoints the root cause for the duplicate request to the base URL.

To understand how current browser versions (Chrome 32, Firefox 26) behave under unexpected input, we did a synthetic injection in `HTML` tags: `<span style:background=X>`, `<img src=X>`, `<script src=X>`, `<iframe src=X>`, `<link href=X>`. Here `X`, the injected character, had ASCII codes in the range 32-126 excluding alphanumeric characters. We found that, in addition to duplicate requests due to empty strings which have been reported before [3], the characters '?' and '#' also resulted in duplicate requests.

`<span style:background=SPACE,EMPTY>` resulted in a duplicate request for both browsers. For Firefox, `<img src=SPACE>`, `<script src=SPACE,EMPTY>`, and `<link href=SPACE>` created duplicate requests. These injections provide evidence that browsers do behave differently and erroneously under unexpected special characters for URIs.

```
1 --- a/modules/mod_fpss/tmpl/Movies/default.php
2 +++ b/modules/mod_fpss/tmpl/Movies/default.php
3 - <span style="background:url(<?php echo $slide->mainImage; ?>) no-
      repeat;">
4 + <span style="background:url(media/system/images/XYZ.jpg) no-repeat;">
5 - <img src="<?php echo $slide->mainImage; ?>" alt="<?php echo $slide->
      altTitle; ?>" />
6 + <img src="media/system/images/XYZ.jpg" alt="<?php echo $slide->
      altTitle; ?>" />
7 - <span class="navigation-thumbnail" style="background:url(<?php echo
      $slide->thumbnailImage; ?>) no-repeat;"> </span>
8 + <span class="navigation-thumbnail"style="background:url(media/system/
      images/XYZ.jpg) no-repeat;"> </span>
```

Listing 1: Code modification to fix unnecessary duplicate requests

```
1 <div class="slide" style="position: absolute; opacity: 0; z-index: 89;"
      >
2   <a class="slide-link" href="/fpss/track/35/L3Jlc291 ,,">
3     <span style="background:url() no-repeat;">
4       <img alt="NEEShub Release 5.0" src="/">
5     </span>
6   </a>
7 .
8 .
9 .
10 <span class="navigation-thumbnail" style="background:url() no-repeat;">
       </span>
```

Listing 2: Runtime state of generated HTML as observed by Firebug

3 Design

Here we detail the design of GRIFFIN to detect duplicate web requests. At a high level, it comprises three steps: model application behavior at the web server (in terms of the function calls and returns), create a signal of the function call depths, and compute the auto-correlation of the signal to trigger detection. Figure 2 shows these steps in GRIFFIN.

3.1 Synchronous Tracing

We leverage SYSTEMTAP [12], a tracing/probing framework that can provide synchronous tracing data on Linux hosts. To enable tracing, SYSTEMTAP allows to write probe-point scripts. Probe-point scripts tell SYSTEMTAP two things. *(1). What event do you want to trace? (2). What do you want to print at the traced event-location?*. GRIFFIN logs both function-entry and function-return events and prints timestamp, thread-id, function call depth, funcation name, file name, line number, and class name, if available. Further tracing implementation details are available in [7].

3.2 Modeling Application Behavior

For modeling purposes, we define a numeric metric called *function call-depth* that represents the runtime function call-depth. At every function-call, the call-depth is incremented and at every return, it is decremented. *Our foundational intuition for modeling application behavior is that the flow of an application can be roughly represented by how function call depth changes.* The function call depth sequence for a given high-level web operation can be considered as a fingerprint of the high-level operation. For further exploration of this intuition, let us first define some terms: *web-request, web-click, http-transaction*. Starting from the lowest level, a web request is the HTTP request sent by the web browser, such as, GET and POST. A web click is a human user clicking in the browser to send web requests. A single web click can generate multiple web requests. A set of web clicks done in a particular sequence, as permitted by the workflow in the website, is called an http transaction. An http transaction can consist of one or more web clicks; in typical usage this will be more than one web click. An example of an http-transaction of size two is going to the homepage followed by going to the login page (HomePage→Login).

Now coming back to our intuition for detecting duplicate web requests, consider that a duplicate web request will create a duplicated signal of the function call depths. It is easy to concoct a synthetic example where this intuition is violated. For example, consider two legitimate consecutive web clicks and the corresponding web requests: (a (b (c c') b') a') (d (e (f f') e') d') giving a call-depth sequences of (1 2 3 3 2 1) (1 2 3 3 2 1). This would give the appearance to GRIFFIN of duplicated web requests. However, we find that for real web pages, the length of web clicks in terms of the number of function calls and returns tends to be much larger. This kind of accidental matching of the function call depth signal happens only very rarely for these real situations.

To get the call-depth at runtime, we add a function called `thread_indent_depth(long)` to SYSTEMTAP 's native scripts. This function returns a number corresponding to the depth of nesting. We call this function `thread_indent_depth(1)` in the probe-point SYSTEMTAP script. Here, the argument one means that at every function-call, increment the depth by one. We submitted this function to the SYSTEMTAP repository and it has been merged into SYSTEMTAP 's master-branch and is available out-of-the-box after SYSTEMTAP is installed [6].

3.3 Duplicate Detection Algorithm

With the function call-depth sequence captured, the next goal is to detect whether the sequence has a repetitive pattern and to do this efficiently with respect to time. To do this, we use a common signal analysis technique to detect repeating patterns, *auto-correlation* [19] of the function call-depth signal. Auto-correlation of a signal x is defined by R_{xx} (Equation 1) as a function of lag-value t, where t varies from zero (perfect signal match with R_{xx}=1) to n, the sequence length in terms of the number

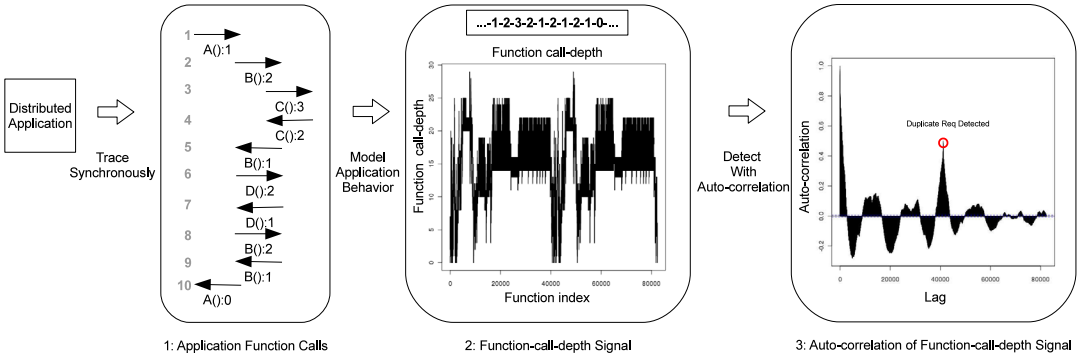


Figure 2: Overview of the duplicate-detection workflow.

of function calls and exits. Ideally for GRIFFIN to detect duplicate web requests resulting from a single user web click, it would be possible to segment the web requests for each web click. But that is not always possible in practice, as we discuss in [7]. Auto-correlation can be viewed as a sequences of *shift, multiply, sum* operations for all lag values on function call-depth signal. Intuitively, we are using auto-correlation to estimate the similarity between the signal and its time shifted versions for various values of the time shift. If the function-depth signal is exactly repeated twice, we expect to see a peak of 0.5 around the lag value of *n/2*.

$$R_{xx}[t] = \frac{C_t}{C_0} \text{ where t=0,...,n}$$
$$C_t = \frac{1}{n} \sum_{s=max(1,-t)}^{min(n-t,n)} [X_{s+t} - \bar{X}][X_s - \bar{X}] \quad (1)$$

After auto-correlation computation for all lag-values, we find the index at which the auto-correlation first becomes negative, call this t_0. For values of auto-correlation beyond t_0, we find if there is any value greater than a threshold value τ. If yes, we flag a duplicate-detection. For the duplication of a set of web requests once, we expect ideally an auto correlation peak of 0.5. But to tolerate the normal variation in function call-depth signal, we set the threshold τ to be a little lower than 0.5. We report on our sensitivity empirical study in Section 5.3. The reason for starting the search beyond t_0 is that then we eliminate the high values of autocorrelation that we will see due to the original signal being correlated with itself with small time lags. The pseudocode for GRIFFIN's detection algorithm is available in [7].

3.4 Usage Modes

We envision GRIFFIN to work in two scenarios, pre-production *testing* and *in-production*. In testing, developer's have control of the environment and trace segmentation is not an issue. Here, a possible concern by developers could be GRIFFIN's detection latency, which is in order of seconds. For in-production mode, operators' main concern could be the overhead of configuring and tuning GRIFFIN and the application tracing overhead, which is incurred in the critical path of all web requests and responses. GRIFFIN's configuration is minimal with only one threshold parameter for which we provide a recommendation (threshold=0.4) with our sensitivity analysis. To further minimize the tracing overhead, an operator can run GRIFFIN in time intervals of low load on the web server .

4 Experimental Setup

4.1 Configurations: Hardware, Software, Tracing

NEEShub infrastructure is running Apache/2.2.16 (Debian) web server in Prefork MPM (Multi-Processing Module) [2] mode, *i.e.*, with multiple processes and one thread per process, on a VM with Intel(R) Xeon(R) CPU E5-2643 0 @ 3.30GHz with 6GB RAM. The PHP-runtime (`libphp5.so`) version is 5.3.3 and is compiled with `--enable-dtrace` option in order for SYSTEMTAP (ver 2.4) to be able to intercept PHP-function calls and returns with its probes.

4.2 Evaluation Metrics

We evaluate GRIFFIN's detection performance with traditional definitions of accuracy and precision. Accuracy is defined as the percentage of true positives and true negatives. Precision is defined as the percentage of true positives out of all detections. We establish the ground truth through manual verification, at client-end, by checking duplicate requests for each web-click using browser debugging tools, Firebug and Chrome-dev-tools. We measure the overhead of GRIFFIN in two areas, tracing overhead and detection overhead. Tracing-overhead is the fraction of total time, taken by SYSTEMTAP 's probes while processing a given web-click. Detection overhead or detection latency is measured in the standard way as the time elapsed for all the detection steps.

5 Evaluation

5.1 Experimental Workload

GRIFFIN's testing was conducted on a replica of the production site (www.nees.org), technically referred to as a "staging machine" where developers merge their code after doing the unit testing on their own development box. We made no modifications or synthetic error injections. Therefore, we expected to find few, if any, problems with the website.

We tested GRIFFIN's duplicate-detection performance by sending a total of 60 HTTP transactions of varying sizes. The size of a transaction is measured by the number of web clicks incorporated within the transaction. Thus, the transaction HomePage→Login has a size of two. Also, for the analysis (autocorrelation computation), the signal is considered the entire transaction. We used 20 transactions for each of the sizes 1,2,3. These 60 HTTP transactions were executed following different possible user workflows as enabled by the web portal. We tried to cover all the workflows that a typical user would follow while visiting the website.

Ideally, the analysis in GRIFFIN will consider the traces corresponding to a single web click from a single user. Within a single user, we expect that different web clicks are handled by threads of different IDs. We empirically validated that this is *always* the case for all our transactions.

5.2 Accuracy and Precision Results

Out of the 7 duplicate request problems (among the 60 HTTP transactions), GRIFFIN was able to correctly find 4 duplicated requests i.e., *HomePage, Topics-page, SimulationWiki-page and Wiki-page*. SimulationWiki page was due to a Javascript-based duplication, while the other three were due to missing-resources. GRIFFIN missed 3 cases of duplicated requests, *warehouse, simulation and education* pages.

GRIFFIN's accuracy and precision with different HTTP transaction sizes is presented in Table 1. GRIFFIN provides an average accuracy of 80% across HTTP transactions of size one and two with no false positives. With three web clicks, GRIFFIN's performance degrades– here 0% precision is misleading in the sense that out of the 20 HTTP transactions of size three, only one (HOMEPAGE→LOGIN→LOGGINGIN (Figure 3)) had a duplicate request which GRIFFIN did not detect. GRIFFIN falsely flagged 4 out of 20 transactions giving a false positive rate of 20% for HTTP transactions of size three. The reason why GRIFFIN did not detect HOMEPAGE web-click within HOMEPAGE→LOGIN→LOGGINGIN transaction is due to the significant difference of LOGGGINGIN function call-depth signal from the signals of HOMEPAGE and LOGIN web clicks (see the increase in function call-depth signal between index 100K to 150K in Figure 3). Here, HOME and LOGIN web clicks have an average function call-depth of 15.61 and 15.47 respectively while LOGGINGIN has an average of 32.42 making it significantly different. With HTTP transaction of size 3, GRIFFIN is performing its analysis after combining these three signals into one. Thus, the divergence in the single combined signal means that the autocorrelation values, even with one duplication, tend to be low, and stay below the threshold. In practice, the HTTP transactions of size 3 will be very rare because of the discrimination that GRIFFIN will be able to do using the thread ID [7].

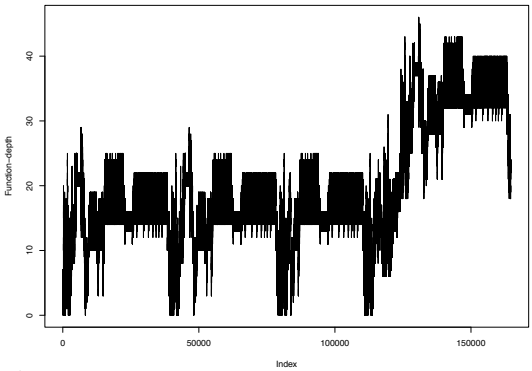
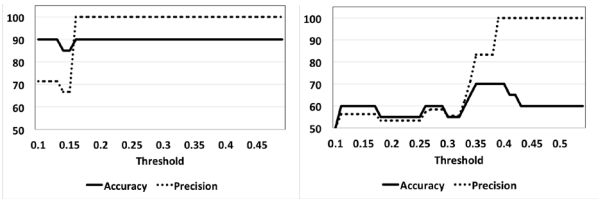


Figure 3: HOME→LOGIN→LOGGINGIN: Function call-depth signal for three web clicks from browser

	Accuracy	Precision
one-click	$90\% = \frac{18}{20}$	$100\% = \frac{3}{3}$
two-clicks	$70\% = \frac{14}{20}$	$100\% = \frac{4}{4}$
three-clicks	$75\% = \frac{15}{20}$	$0\% = \frac{0}{4}$

Table 1: Summary of Performance results

With the ideal (and practically common) case of analysis over HTTP transaction of size 1, GRIFFIN shows 90% accuracy and 100% precision. As an example, the function call-depth and autocorrelation for HOME web-transaction is presented in Figure 2. We see that the autocorrelation has a clear peak value of 0.4998 near a lag-value of 40,000 which is detected by GRIFFIN (with a threshold set at 0.4). Manual checking, both at user-end and at server-end revealed that HOME web-request ("/") is being sent twice by the user's browser. Further inspection on the server revealed that a field called `hits` in the back-end database is incremented on every HOME web-transaction. We reported this hitherto unknown problem to the web developer at NEES, and it was subsequently fixed and not pushed into the production environment. Testing GRIFFIN with HTTP transactions of size 2, we observe a drop in accuracy (to 70%). This happens due to the significant variability in the basic signal due to the very different nature of the function call invocations in the two web clicks. Expectedly, autocorrelating a divergent signal gives low autocorrlation values, which sometime fall below the GRIFFIN threshold (0.4).



(a) One-click (b) Two-click

Figure 4: Sensitivity of GRIFFIN for one and two-clicks

	Tracing Overhead (Avg)	Tracing Overhead (Std. Dev)	Sequence Length (Avg)	Sequence Length (Std. Dev)
one-click	24.0%	6.6%	67,071	54,165
two-clicks	32.8%	11.6%	131,511	76,630
three-clicks	29.1%	9.1%	141,427	33,727

Table 2: Tracing Overhead

5.3 Sensitivity and Overhead

GRIFFIN's sensitivity to different parameters, sequence length, threshold and number of traced contiguous web clicks is critical from a usability perspective. With an increasing number of contiguous web clicks, GRIFFIN's accuracy and precision drop. The pattern of accuracy decreasing with increasing number web clicks holds true with increasing sizes of the traces. We present GRIFFIN's sensitivity with different thresholds in Figure 4.We set GRIFFIN threshold to 0.4 as the default value for GRIFFIN to provide us zero false positives, *i.e.*, 100% precision. The user can decrease the threshold for fine tuning her system, but we suggest to not go below 0.35 (based on Figure 4b) as that can result in possible false positives.

The detection latency as a function of the sequence length (*i.e.*, the number of trace events due to SYSTEMTAP probes) shows the expected behavior of greater latency with increasing sequence length [7]. This is due to a larger number of autocorrelation computations for a longer trace length. However, the upper range of the sequence length is typically about 100K and with that we have a detection latency of about half a minute, which should be fast enough to be useful for the subsequent manual process of fixing the problem. The average tracing overhead across the 60 tested HTTP transactions is 28.6% with a standard devitation of 10.0%. The overhead for HTTP transactions for each size is presented in Table 2. The tracing overhead is independent of the length of the sequence and the differences seen are due to statistical variations.

5.4 Diagnostic-context

When GRIFFIN detects duplicate web-requests, a diagnostic-context about the detection would help the developers as a starting point for debugging. At detection-time, in addition to the autocorrelation value, we also have the lag when this autocorrelation value exceeded the threshold, call this t_{max}. We use t_{max} alongwith the information provided by an additional SYSTEMTAP probe that records the HTTP-request going from `apache-core` to PHP-runtime, to provide the *diagnostic-context*. With the t_{max}, we get the nearest next fired `apache-core` to PHP event. We then extract a high-level component (module name) from the file name. For the duplicate bug of Figure 1, this simple scheme is able to correctly flag `mod_fpss` module in Joomla, the Content Management System, on which HUBzero is built.

6 Related Work

Most of the existing approaches to handle duplicate requests are *not* at the application-level. TCP [9] is the classic example that uses sequence numbers along with a windowing-based mechanism to do duplicate detection of IP packets. Stateless protocols like HTTP have to deal with the request-response nature and maintain state at the application-level. Application-level works include similarity detection [16] deployed at web-proxy caches to eliminate redundant network traffic, duplicate-content detection [18] with clustering and similarity metrics [11]. These are directed at generic payloads and are therefore less accurate than GRIFFIN in general.

Finding relevant system events to detect and diagnose failures is often equated to the problem of finding a needle in a haystack. Over the last decade, several researchers have proposed solutions to this challenging problem [10, 20, 8, 13]. The high-level objective here is to mine vast amounts of system data to find relevant signatures for failures. Our work falls within this broad umbrella. We automate the process of detecting duplicated web requests by looking at a compressed signal from system events, specifically function calls and returns.

7 Conclusion

In this paper, we have presented a systematic method and an automated tool called GRIFFIN for detecting an important problem that afflicts many web servers, namely, duplicate client browser requests. This causes an artificially high load on servers and corrupts server and client state. Culling together many blog posts and developer forum reports, we identify the two fundamental root causes of the problem and come up with a solution that handles both, without needing special case logic for the two root causes or for different browsers. We use GRIFFIN for detecting the problem in a production web portal for an NSF center at Purdue and identify that the problem is more widespread than previously identified. Our evaluation on the production site revealed no false positive. The dynamic system tracing using SYSTEMTAP is lightweight and the detection latency small enough (less than half a minute) as to be useful in practice. Our contributions were considered significant enough that the problem was fixed in the web portal and our addition to the dynamic tracing facility was accepted in its official release.

References

[1] Alexa Internet, Inc. http://www.alexa.com/.

[2] Apache MPM prefork. http://httpd.apache.org/docs/2.2/mod/prefork.html.

[3] Empty image src can destroy your site. http://www.nczonline.net/blog/2009/11/30/empty-image-src-can-destroy-your-site/.

[4] Empty SRC And URL() Values Can Cause Duplicate Page Requests. http://www.bennadel.com/blog /2236-Empty-SRC-And-URL-Values-Can-Cause -Duplicate-Page-Requests.htm.

[5] HTML 4.01 Specification. http://www.w3.org/TR/html4/.

[6] Systemtap call-depth feature request. https://sourceware.org/bugzilla/show_bug.cgi?id=16472.

[7] ARSHAD, F., MAJI, A.K. MUDGAL, S., AND BAGCHI, S. Is your web server suffering from undue stress due to duplicate requests? http://docs.lib.purdue.edu/ecetr/458, Apr. 24 2014. Technical Report, School of Electrical and Computer Engineering, Purdue University.

[8] BODIK, P., GOLDSZMIDT, M., FOX, A., WOODARD, D. B., AND ANDERSEN, H. Fingerprinting the datacenter: Automated classification of performance crises. In *Proceedings of the 5th European Conference on Computer Systems* (New York, NY, USA, 2010), EuroSys '10, ACM, pp. 111–124.

[9] CERF, V., AND KAHN, R. A protocol for packet network intercommunication. *Communications, IEEE Transactions on 22*, 5 (May 1974), 637–648.

[10] COHEN, I., ZHANG, S., GOLDSZMIDT, M., SYMONS, J., KELLY, T., AND FOX, A. Capturing, indexing, clustering, and retrieving system history. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2005), SOSP '05, ACM, pp. 105–118.

[11] COSKUN, B., AND GIURA, P. Mitigating sms spam by online detection of repetitive near-duplicate messages. In *Communications (ICC), 2012 IEEE International Conference on* (June 2012), pp. 999–1004.

[12] EIGLER, F. C., PRASAD, V., COHEN, W., NGUYEN, H., HUNT, M., KENISTON, J., AND CHEN, B. Architecture of systemtap: a linux trace/probe tool.

[13] FU, Q., LOU, J.-G., LIN, Q.-W., DING, R., ZHANG, D., YE, Z., AND XIE, T. Performance issue diagnosis for online service systems. In *Reliable Distributed Systems (SRDS), 2012 IEEE 31st Symposium on* (2012), IEEE, pp. 273–278.

[14] MCLENNAN, M., AND KENNELL, R. Hubzero: A platform for dissemination and collaboration in computational science and engineering. *Computing in Science & Engineering 12*, 2 (2010), 48–53.

[15] SOUDERS, S. High-performance web sites. *Commun. ACM 51*, 12 (Dec. 2008), 36–41.

[16] SPRING, N. T., AND WETHERALL, D. A protocol-independent technique for eliminating redundant network traffic. *SIGCOMM Comput. Commun. Rev. 30*, 4 (Aug. 2000), 87–95.

[17] STEVE W. Monkey Code. http://code.alittlegoofy.com/2008/12/i-found-something-peculiar-about.html.

[18] VALLÉS, E., AND ROSSO, P. Detection of near-duplicate user generated contents: The sms spam collection. In *Proceedings of the 3rd International Workshop on Search and Mining User-generated Contents* (New York, NY, USA, 2011), SMUC '11, ACM, pp. 27–34.

[19] VENABLES, W. N., AND RIPLEY, B. D. *Modern Applied Statistics with S*. Springer Publishing Company, Incorporated, 2010.

[20] XU, W., HUANG, L., FOX, A., PATTERSON, D., AND JORDAN, M. I. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles* (New York, NY, USA, 2009), SOSP '09, ACM, pp. 117–132.

A Model-Based Namespace Metadata Benchmark for HDFS

Cristina L. Abad
Escuela Superior Politécnica del Litoral

Yi Lu Roy H. Campbell
University of Illinois, Urbana-Champaign

Nathan Roberts
Yahoo, Inc.

Abstract

Efficient namespace metadata management is increasingly important as next-generation storage systems are designed for peta and exascales. New schemes have been proposed; however, their evaluation has been insufficient due to a lack of an appropriate namespace metadata benchmark. We describe MimesisBench, a novel namespace metadata benchmark for next-generation storage systems, and demonstrate its usefulness through a study of the scalability and performance of the Hadoop Distributed File System (HDFS).

1 Introduction

There are no metadata-intensive benchmarks with realistic workloads for next-generation storage systems [1, 13]. A few existing tools [7, 9] are useful as microbenchmarks, but do not reproduce realistic workloads.

The storage community has long-acknowledged the need for benchmarks based on realistic workloads, and programs like SPECsfs2008 [12] and Filebench [6] are extensively used for this purpose in traditional storage systems. However, emerging Big Data workloads like MapReduce [2] have not been properly synthesized yet.

We present MimesisBench, a novel metadata-intensive storage benchmark suitable for Big Data workloads. MimesisBench consists of a workload modeling tool, a workload generator, and a workload profile from a large cluster at Yahoo. More workloads will be added in the future.

The model on which MimesisBench is based [3] allows it to generate type-aware workloads, in which specific types of file behavior can be isolated or modified for 'what-if' and sensitivity analysis. These types of files are modeled autonomically, using unsupervised statistical clustering. The model also supports multidimensional workload scaling.

MimesisBench's Hadoop-based implementation allows it to be used in any storage system that is compatible with Hadoop (e.g., HDFS, Ceph, CassandraFS, Lustre). We have released the benchmark and workloads as open source so that other researchers can benefit from it[1].

This paper makes two contributions. First, we extend a model for temporal locality and popularity in object request streams [3] to: (1) include other operations in addition to regular accesses to objects (opens), (2) support pre-existing files (created before the benchmark), and (3) support a realistic hierarchical namespace. Second, we use this model to implement a metadata-intensive storage benchmark to issue realistic workloads on distributed storage systems. This benchmark can be used to evaluate the performance of storage systems without having to deploy a large cluster and its applications.

[1] Available: `http://sites.google.com/site/cristinaabad`

2 Model

We extend a model we proposed for generating accesses to objects (e.g., opens to files) [3], which is able to reproduce temporal correlations in object request streams that arise from the long-term popularity of the objects and their short-term temporal correlations. This model is suitable for Big Data workloads because it supports highly dynamic populations, it is fast and scalable to millions of objects, and it is workload-agnostic so it can be used to model emerging workloads.

Objects or files in a stationary segment of a request stream are modeled as a set of *delayed renewal processes* [11] (one per object). Each object in the stream is characterized by its time of first access, its access interarrival distribution, and its active span (time during which an object is accessed). With this approach, the system-wide popularity distribution asymptotically emerges through explicit reproduction of the per-object request arrivals and active span [3]. However, this model is unscalable, as it models each object independently.

To reduce the model size, a lightweight version uses unsupervised statistical clustering (k-means) to identify groups of objects with similar behavior and significantly reduce the model space by modeling "types of objects" instead of individual objects. As a result, the clustered model is suitable for synthetic workload generation.

2.1 Extensions to the model

The model described above cannot be used directly to test a storage system since it only reproduces accesses (e.g., opens) to an object (i.e., file) and not other operations that are also critical in a storage system like creates and deletes. We propose the following extensions to the

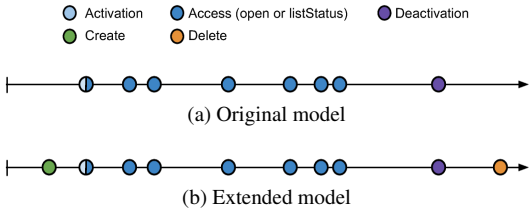
original model to make it suitable for namespace metadata benchmarking: (1) additional storage system operations, (2) pre-existing files, and (3) realistic namespaces.

2.1.1 Extension 1: Additional operations

We first extend our model to include file creations, deletions and list operations in addition to regular opens. We focus on these operations because together they constitute more than 95% of the namespace metadata operations in MapReduce clusters [2], thus accounting for the vast majority of the workload.

The **list** operations are like opens and only *read* or access the namespace. Thus, we model both list and opens together as *accesses* to files. A parameter keeps track of the percentage of read operations that constitute opens and those that are list.

On the other hand, **creates** and **deletes** *write* or modify the namespace, and are characterized independently.



(a) Original model

(b) Extended model

Figure 1: Operations on a single file, with the original and the extended model. The file's activation time is given by its creation time plus the delay to its first access; the deactivation time is given by its activation time plus its active span.

Figure 1 shows how the original and extended models generate operations on a file of a particular type (cluster). The extended model requires the following additional per-cluster statistical information: distribution of create interarrivals, distribution of delay to deletion (relative to the object's last access in the stream, which is given by the time of first access + active span), and percentage of accesses that are open operations (list are the remaining percentage). In addition, the activation time is relative to the creation stamp of a file (or to the beginning of the test, for files that already exist at t_0).

Table 1 lists the parameters required by the extended model to generate events to files.

This extension does not affect the access patterns preserved by the original model, which are characterized by the per-cluster access interarrival distribution and the per-cluster distribution of active spans.

2.1.2 Extension 2: Pre-existing files

When benchmarking storage systems, we must consider that their performance depends on the state of the underlying file system [1, 4], namely the pre-existing files and the structure of the hierarchical namespace (the latter is discussed in the next subsection).

We extend our model to keep track of the number of files (within each file type) that were created sometime before the beginning of the modeled trace.

Table 1: Parameters used to generate events on files; O_i is the object or file i, $i \in \{1,...,n\}$; K_j is the cluster or file type j, $j \in \{1,...,k\}$.

Symbol	Description
n	number of files in the trace or request stream
k	number of clusters or types of files
F_j	interarrival distribution of accesses to an object in K_j
C_j	interarrival distribution of creations for objects in K_j
G_j	distribution of delay to first access to objects in K_j (relative to creation or t_0); $C_j + G_j = \text{Activation}_j$
H_j	active span distribution $\forall O_i \in K_j$; $\text{Activation}_j + H_j = \text{Deactivation}_j$
D_j	distribution of delay to delete event of objects in K_j; $\text{Deactivation}_j + D_j = \text{Deletion}_j$
p	percentage of accesses that are opens ($1-p$: list)
w_j	percentage of objects in K_j ; $\sum_{j=1}^{k} w_j = 1$
t_{end}	duration of request stream (in milliseconds)
pre_j	number of pre-existing files in K_j
PN	Pre-existing namespace conf. file (generated with NGM)
FD	Array with percentage of files at each depth in hierarchy

We could infer if a file exists prior to the captured trace of namespace events by making the assumption that any file accessed in the trace, but not created during it, is a pre-existing file. However, this approach would lead to an inaccurate model if the trace contains many operations on files that do not exist (e.g., due to users incorrectly entering the name of a file).

To avoid this problem we can use a *namespace metadata trace* that contains a snapshot of the namespace (file and directory hierarchy), in addition to the set of events that operate atop that namespace (e.g., open a file, list directory contents) [1]. The traces we analyzed consist of access logs obtained by parsing the name node audit logs, plus namespace snapshots obtained with Hadoop's Offline Image Viewer tool.

2.1.3 Extension 3: Realistic namespaces

Earlier, we proposed [1] a statistical model for generating realistic namespace hierarchies, and implemented a *namespace generation module* (NGM) based on it. To the best of our knowledge, this is the only available tool that can generate large realistic namespaces[2].

Prior to issuing the workload, the NGM is used to generate a realistic directory structure, which preserves the following statistical properties of the original: number of directories, distribution of directories at each depth, and distribution of subdirectories per directory.

To integrate the files to this directory structure, we add a parameter to the model, the *percentage of files at each depth* of the hierarchy, and proportionally assign files to each depth according to this parameter.

[2]We tested the only other alternative system, the *Impressions* framework [4], and were not able to generate the large namespaces observed in Big Data storage deployments since it was designed to model smaller (more traditional) namespaces. Furthermore, at the time of this writing, the Impressions framework is no longer available for download.

2.2 Assumptions and limitations

Our model can be used to generate realistic synthetic workloads to evaluate the *namespace metadata management subsystems*. Dimensions related to data input/output behavior—like a correlation between file size and popularity or the length of data read/write operations—are out of the scope of our model.

We model **stationary segments** of object request streams. Workloads consisting of a few stationary segments can be divided using the approach in [15]. However, in practice, benchmark runs tend to issue the workload of a period of one hour or less, so the need to consider non-stationary segments is not critical.

We represent arrivals with a sequence of **interarrival times** $X = (X_1, X_2, ...)$, where X is a sequence of *independent, identically distributed random variables*. In practice, there could be autocorrelations in the arrival process, leading to bursty behavior at different timescales. In [3] we briefly discuss how ON/OFF processes can be used to capture burstiness.

We assume that each arrival process of the accesses to a file in the storage system is **ergodic**, so its interarrival distribution can be deduced from a single realization of the process (i.e., the trace of events representing the segment being modeled). In addition, instead of forcing a fit to a particular interarrival distribution, we use the empirical distribution of the interarrivals inferred from the arrivals observed in the trace being modeled.

Finally, there may be unknown, but important, behavior in the original workloads that our model does not capture. Some of these dimensions, like spatial locality, may be added to our model in the future. However, there is a trade-off of increased complexity due to adding these dimensions.

3 Design

Similar to Hadoop's DFSIO [16] and S-live [9], MimesisBench is a MapReduce job in which a set of mappers simultaneously issue requests to the storage layer. Each mapper is in charge of issuing the operations on files of a particular type (as encapsulated by the model). The cluster used to run MimesisBench must have at least k nodes available to run a mapper task each, so that the full workload can be issued simultaneously and the mappers do not interfere with each other during their operation.

A run of MimesisBench has two phases: First, the pre-existing files are created; next, the workload is issued.

In each phase, a *job coordinator* parses the parameters and generates configuration files for each worker. In addition, the job coordinator of the first phase creates the hierarchical namespace based on an input parameter file that has been pre-generated with the Namespace Generation Module (NGM).

In the **first phase**, the workers create the target number of files for each type. A parameter tells the workers to use a flat namespace (create all files in a single, configurable path) or a hierarchical one. Files of each type are created at different levels of the namespace hierarchy, proportionally to the configuration parameter of *files at each depth* (which indicates what percentage of files are located at each depth). The subdirectories at each depth are assigned to a file type, proportionally to the weight of the cluster (w_j) to which the file belongs.

In the **second phase**, the workers issue the load. Each *load generator* worker reads the configuration for the specific file type that it has been assigned and waits in a time-based barrier[3] to start issuing the load corresponding to the files that belong to the type it is in charge of. Two data structures are used to keep track of the files and events: a `PriorityBlockingQueue` of files (sorted by the timestamp of the next event—create, open, etc.—of that file) and a FIFO `BlockingQueue` of events to be issued[4]. Three threads coordinate access to these data structures: a file introduction thread adds files to the priority queue (using the cluster's create interarrival distribution)[5], another thread continuously polls the priority queue and adds details of the next event to be issued to the back of the FIFO queue. Finally, a consumer thread pulls the information of the next event to be issued from the FIFO queue and schedules it to be issued at the proper time, using Java's `ScheduledExecutorService`.

All file create operations create files of size zero. This allows us to ignore the effect of writing bytes to a file, handled by the data nodes, and concentrate on evaluating the namespace metadata server (name node in HDFS) performance.

A configurable maximum allowed drift is used to abort a run of the benchmark if the events are falling behind from their original schedule. In that case, more mappers would be needed to issue the workload.

A collector reducer task gathers the stats from the workers and generates aggregate results (see Table 2).

Table 2: Statistics reported by MimesisBench.

Description	Unit
Throughput	ops/sec
Active time (workers)	*msecs*
Operations issued	*ops*
Successful creates/deletes/opens/list	*ops*
Average latency: create/delete/open/list	*msecs*

3.1 Scaling workloads

A workload can be scaled across several dimensions:

[3] The clocks of the nodes in a Hadoop cluster are typically synchronized to support Kerberos authentication.

[4] The size of these queues in the current implementation is set to be the number of files of that particular type for the former, and 1 500 000 for the latter.

[5] Pre-existing files of a type are added to the queue upon loading.

Number of files. A workload profile comes with a configured number of files, n, as observed in the original trace on which the model is based. One can increase or decrease the number of files to emulate a larger load.

Number of files of a particular type. The user can increase the number of files of only one particular type by increasing n and doing a transformation on the w_j weights. For example, to double the number of objects of type 1 while keeping the number of objects of types 2 to k unchanged, we can obtain the values of n_{new} and $\{w_{1_{\text{new}}}, w_{2_{\text{new}}}, ..., w_{k_{\text{new}}}\}$ by solving:

$$n_{\text{new}} = n + w_1 \times n \quad (1)$$

$$w_{1_{\text{new}}} \times n_{\text{new}} = 2(w_1 \times n) \quad (2)$$

$$w_{j_{\text{new}}} \times n_{\text{new}} = w_j \times n, \quad j \in \{2, ..., k\} \quad (3)$$

Time. Interarrivals can be *accelerated* by multiplying the random variable by a scaling factor between 0 and 1. A scaling factor of 1 reproduces the original workload, while a scaling factor of 0 provides maximum stress on the system by issuing all the operations as fast as possible (0 millisecond wait between operations). Interarrivals can also be slowed down by multiplying the random variable by a constant greater than 1.

Active span. The active span random variable can also be multiplied by a user-defined constant. Modifying the active span has the effect of modifying the number of accesses of the files, thus modifying their popularity.

3.2 Isolating workloads

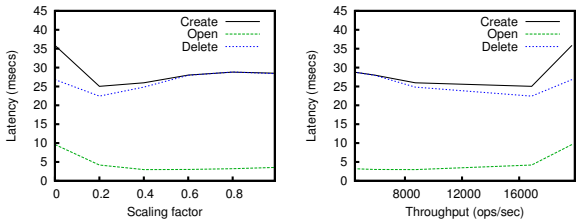
The user can choose to isolate the workload of a particular file type or cluster (i.e., turn-off the other types of files). This can be used to analyze how a particular type of file affects the performance of the system.

Summary Table 3 shows a summary of the features of MimesisBench and other related tools.

4 Evaluating a Big Data storage system

We demonstrate the usefulness of MimesisBench by using it to evaluate the performance and scalability of the HDFS name node across several dimensions.

We modeled a 1-day (12/1/2011) *namespace metadata trace* from a Hadoop cluster at Yahoo. The trace came from a 4100-node production cluster [10]. It contains 60.9 million opens events that target 4.3 million distinct files, and 4 PB of used storage space. For a detailed workload characterization of this cluster see [2]. We modeled the trace using 30 file types or clusters (i.e., $k = 30$ for the k-means clustering algorithm). We chose this value of k because it was the smallest for which we could obtain a close approximation of the file popularity



(a) Operation latency (b) Throughput vs. latency

Figure 2: *Left:* Operation latency for varying interarrival speeds, averaged across five runs. In each run, the interarrival random variable, X, was multiplied by a constant, c, shown in the x-axis. When $c = 0$, the operations are issued as fast as possible. When $c = 1$ the operation interarrival mimics the interarrivals of the original trace. For all data points, the standard deviation is very small (< 0.7 msecs). *Right:* Throughput vs. operation (open) latency.

distribution (Kolmogorov-Smirnov distance between the real and synthetic CDFs $< 4\%$).

Our performance testbed had 32 nodes, each with 2 Xeon 2.4GHz quad-core processors and 24GB RAM.

4.1 Latency of opens, creates and deletes

Figure 2 shows the effect on the operation latency as interarrivals are accelerated. In general, read operations are faster than write operations because acquiring an exclusive lock is not necessary to read the namespace.

The latency of the operations is not initially affected by issuing them faster. This shows, that the performance of the name node is not degraded as more operations are issued per second. However, when the clients issue more than 16,800 operations per second ($c \geq 0.2$), the name node's performance starts degrading rapidly. This information can be used to determine whether the name node can properly support an increased workload in the future.

4.2 Flat versus hierarchical namespaces

Figure 3 shows the impact a hierarchical namespace (versus a flat one) has on the name node. The performance degrades significantly faster on a flat namespace than on a hierarchical one. The hierarchical namespace can serve up to 19,696 ops/sec versus 10,284 ops/sec for the flat namespace.

These results show that using benchmarks that create files in a flat namespace (see Table 3) is not desirable as they place a heavier and unrealistic burden on the locking mechanisms of the metadata server. In this case, the problem is with the locking used to log namespace write operations used for auditing purposes.

4.3 Isolating workloads

We isolated the workload of two file types with different access patterns and observe the effects on latency and throughput (see Table 4). We chose these two clusters because they represent two extreme, read-mostly (cluster 29) and write-heavy (cluster 17), yet realistic, workloads.

We ran several tests with events issued at normal speed

Table 3: Features of MimesisBench and other related tools. See Section 5 for a description of these tools.

Feature	NNBench	DFSIO	S-live	Filebench	SPECsfs	mdtest	MimesisBench
For next-generation storage	✓	✓	✓				✓
Metadata-intensive	✓					✓	✓
Realistic workloads				✓	✓		✓
Type-aware workload				✓			✓
Autonomic type-awareness							✓
Hierarchical namespace	Semi-flat$^+$		Semi-flat$^+$	Limited*	Fixed		✓
Issues I/O load		✓	✓	✓	✓		

$^+$ Only multiple directories at depth 1 are supported.

* Creates hierarchies with a given depth and width. Characteristics like subdirectories per directory and directories per depth, are not supported.

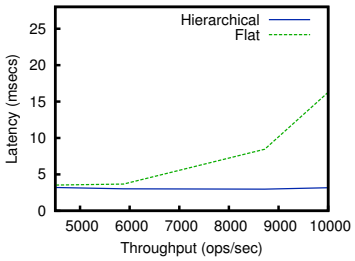


Figure 3: Throughput vs. open latency in a hierarchical and a flat namespace. Five runs; standard deviation < 1.5 msecs.

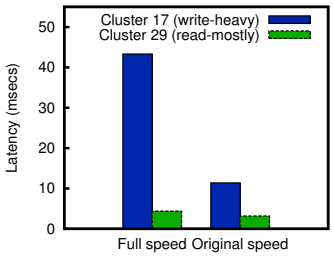


Figure 4: Latency of open events, with two interarrival scaling factors (0 or operations at full speed, and 1 or operations issued at their original speed), averaged across five runs, for two clusters with different read/write ratios (see Table 4). Standard deviation < 0.5 msecs.

(interarrival scaling factor $= 1$) and at full speed (interarrival scaling factor $= 0$). To ensure a fair comparison, we adjusted the number of mappers issuing the workload so that the total number of operations issued in both tests was roughly the same. At 30 mappers for cluster 17, and 12 mappers for cluster 29, the number of operations issued in the tests was $3\,559\,101 \pm 0.9\%$.

Figure 4 shows the latency of open events. The latency of the open events degrades significantly more in a write-heavy workload: When operations are issued at full speed, the latency of opens in the write-heavy workload increases 3.9x in cluster 17 versus 1.4x in cluster 29. In addition, at maximum issuing speed (interarrival scaling = 0) the name node can serve 8 times more operations when the workload constitutes only reads: 53,233 vs. 6,453 ops/sec for clusters 29 and 17, respectively.

Table 4: Characteristics of the two clusters whose workload was isolated. We chose these two clusters because they represent two extreme, read-mostly and write-heavy, workloads.

	Cluster 17	Cluster 29
Mean interarrivals (regular accesses)	179.61 msecs	4.92 msecs
Mean creates interarrivals	40.64 msecs	87,355.00 msecs
Mean active span	3.33 mins	8.33 mins
Percentage of read operations	69%	$\approx 100\%$
Percentage of write operations	31%	$\approx 0\%$

4.4 Evaluating a proposed enhancement

A common use for benchmarks is to evaluate a new design against an old design. In this subsection, we show the results of one example of this type of evaluation.

The HDFS-5239 [8] Jira allows the namespace lock fairness to be changed from fair (default) to unfair. This change may be desirable in large clusters, as an unfair lock has a higher throughput than a fair lock.

Table 5: Throughput improvement obtained when using an unfair namespace lock (HDFS-5239), for four different workloads.

	Fair lock	Unfair lock	Improvement
Hierarchical	19 696 ops/sec	21 899 ops/sec	11%
Flat	10 255 ops/sec	11 299 ops/sec	10%
Read-mostly	53 324 ops/sec	103 331 ops/sec	94%
Write-heavy	8 694 ops/sec	8 701 ops/sec	1%

Table 5 shows the name node throughput improvement for the following scenarios: full workload on a hierarchical namespace, full workload on a flat namespace, a realistic read-mostly workload (cluster 29), and a realistic write-heavy workload (cluster 17). The improvement when the full workload is issued is around 10%. However, for a read-mostly workload, the improvement is as high as 94%. This is a result of the bottleneck that exists in the logging mechanism used for write operations.

4.5 Stability of the results

Benchmarks are often used to evaluate new designs against an old design, or against another competing new design. An improvement of 10% in performance may be desirable to push in a large system, as long as we can trust the results of the benchmark. For this reason, it is important to have a small variability in the results produced by a benchmark. For example, it is not reasonable to trust a 10% performance improvement if the coefficient of variation, c_v, of the results is 8%.

Our experiments had a very small standard deviation and corresponding coefficient of variation (or the ratio

of the standard deviation to the mean). For all the sets of experiments we ran, the greatest observed coefficient of variation was 2.38%, with an average coefficient of variation of 1.08%. These results suggest that MimesisBench is suitable as a tool to evaluate expected performance improvements of new designs.

5 Related work

Some prior tools provide a subset of the features of MimesisBench, as summarized in Table 3.

Mdtest issues metadata intensive workloads. However, it does not support realistic workloads or namespaces. In addition, mdtest interfaces with the storage system via system calls and has not been ported to the interfaces of next-generation storage systems, nor does it allow distributed workload generation.

Filebench [6] uses a POSIX-compliant interface and includes traditional workloads like web server, file server, and database server. Emerging Big Data workloads have not been included as pre-defined workloads.

SPECsfs2008 [12] is used as a standard to enable comparison of file server throughput and response times across different vendors and configurations. It supports NFSv3 and CIFS APIs and comes with a pre-defined workload based on enterprise NFS and CIFS workloads.

For HDFS, the Hadoop developer community has designed two benchmarks that can test I/O operations and do simple stress-tests on storage layer: *DFSIO* [16] and *S-live* [9]. However, these benchmarks do not reproduce realistic workloads and can only be used as microbenchmarks. Another Apache tool, *NNBench* [7] was created to benchmark the HDFS name node; however, it can only issue one type of operation at a time, and does not work atop a realistic namespace.

Tarasov et al. [14] found that traditional workloads, like those provided with Filebench, are a poor replacement for virtual machine workloads. We consider another emerging workload: MapReduce clusters.

Impressions [4] generates realistic file system images; however, it is not readily coupled with a workload generator to easily reproduce workloads that operate on the generated namespace. Furthermore, the generative model used by Impressions to create the file system hierarchy is not able to reproduce the distributions observed in our analysis, nor is it able scale to the large hierarchies observed in the Big Data systems we have studied.

Chen et al. [5] proposed the use of multi-dimensional statistical correlation (k-means) to obtain storage system access patterns and design insights in user, application, file, and directory levels. However, the clustering was not leveraged for workload generation or benchmarking.

In earlier work, we developed Mimesis [1], a synthetic trace generator for namespace metadata traces. However, it was too CPU-intensive to issue operations at real-time and as such is inapplicable for benchmarking real systems (though its synthetic traces could be used in trace-based evaluations). In addition, its model does not reproduce file popularity.

6 Conclusions

We presented MimesisBench, a metadata-intensive storage benchmark suitable for Big Data workloads. MimesisBench consists of a workload-generating software and a workload from a Yahoo Big Data cluster.

MimesisBench is extensible and more workloads can be added in the future. It is based on a novel model that allows it to generate type-aware workloads, in which specific type of file behavior can be isolated or modified for 'what-if' and sensitivity analysis. In addition, it supports multi-dimensional workload scaling.

MimesisBench is implemented on top of the Apache Hadoop framework, which allows it to be used in any storage system that is compatible with Hadoop. We have released the benchmark and workload as open source.

A study of the performance and scalability of HDFS was presented to show the usefulness of metadata-intensive benchmarking using MimesisBench.

Acknowledgments

We thank our anonymous reviewers for their many insightful comments and suggestions. This work was partially completed during C. Abad's PhD studies at UIUC and during her internship at Yahoo. R. Campbell and C. Abad were supported in part by AFRL grant FA8750-11-2-0084. Y. Lu is partially supported by NSF grant CNS-1150080.

Bibliography

[1] ABAD, C. L., LUU, H., ROBERTS, N., LEE, K., LU, Y., AND CAMPBELL, R. H. Metadata traces and workload models for evaluating Big storage systems. In *Proc. UCC* (2012).

[2] ABAD, C. L., ROBERTS, N., LU, Y., AND CAMPBELL, R. H. A storage-centric analysis of MapReduce workloads: File popularity, temporal locality and arrival patterns. In *Proc IISWC* (2012).

[3] ABAD, C. L., YUAN, M., CAI, C., ROBERTS, N., LU, Y., AND CAMPBELL, R. H. Generating request streams on Big Data using clustered renewal processes. *Perf. Eval. 70*, 10 (Oct. 2013).

[4] AGRAWAL, ARPACI-DUSSEAU, AND ARPACI-DUSSEAU. Generating realistic Impressions for file-system benchmarking. *ACM TOS 5* (2009).

[5] CHEN, Y., SRINIVASAN, K., GOODSON, G., AND KATZ, R. Design implications for enterprise storage systems via multi-dimensional trace analysis. In *Proc. SOSP* (2011).

[6] FileBench. sourceforge.net/projects/filebench, Mar. 2013. Last accessed: April 15, 2013.

[7] NOLL, M. G. Benchmarking and stress testing an Hadoop cluster with TeraSort, TestDFSIO & co. www.michael-noll.com/blog/2011/04/09/benchmarking-and-stress-testing-an-hadoop-cluster-with-terasort-testdfsio-nnbench-mrbench, Apr. 2011. Last accessed: January 18, 2014.

[8] SHARP, D. Allow FSNamesystem lock fairness to be configurable. issues.apache.org/jira/browse/HDFS-5239, 2013. Last accessed: March 26, 2014.

[9] SHVACHKO, K. A stress-test tool for HDFS (Apache JIRA HDFS-708). issues.apache.org/jira/browse/HDFS-708, 2010. Last accessed: December 30, 2013.

[10] SHVACHKO, K., KUANG, H., RADIA, S., AND CHANSLER, R. The Hadoop Distributed File System. In *Proc. MSST* (2010).

[11] SIEGRIST, K. Chapter 14: Renewal processes. In *Virtual Laboratories in Probability and Statistics*. 2013. Last accessed: March 23, 2013.

[12] SPECsfs2008. http://www.spec.org/sfs2008/, Apr. 2013. Last accessed: December 30, 2013.

[13] TARASOV, V., BHANAGE, S., ZADOK, E., AND SELTZER, M. Benchmarking file system benchmarking. In *HotOS* (2011).

[14] TARASOV, V., HILDEBRAND, D., KUENNING, G., AND ZADOK, E. Virtual machine workloads: The case for new benchmarks for NAS. In *Proc. Usenix FAST* (2013).

[15] TARASOV, V., KUMAR, S., MA, J., HILDEBRAND, D., POVZNER, A., KUENNING, G., AND ZADOK, E. Extracting flexible, replayable models from large block traces. In *Proc. Usenix FAST* (2012).

[16] VAVILAPALLI, V. K. Delivering on Hadoop .Next: Benchmarking performance. hortonworks.com/blog/delivering-on-hadoop-next-benchmarking-performance, 2012. Last accessed: December 30, 2013.

Towards Combining Online & Offline Management for Big Data Applications

Brian Laub, Chengwei Wang, Karsten Schwan, Chad Huneycutt
College of Computing, Georgia Institute of Technology, Atlanta, GA 30318, USA
{*brian.laub, flinter, karsten.schwan, chadh*}*@cc.gatech.edu*

Abstract

Traditional data center monitoring systems focus on collecting basic metrics such as CPU and memory usage, in a centralized location, giving administrators a summary of global system health via a database of observations. Conversely, emerging research systems are focusing on scalable, distributed monitoring capable of quickly detecting and alerting administrators to anomalies. This paper outlines VStore, a system that seeks to combine fast online anomaly detection with offline storage and analysis of monitoring data. VStore can be used as a historical reference to help guide administrators towards quickly classifying and fixing anomalous behavior once a problem has been detected. We demonstrate this idea with a distributed big streaming data application, and explore three common fault scenarios in this application. We show that each scenario exhibits a slightly different monitoring history, which may be undetectable by online algorithms that are resource-constrained. We also offer a discussion of how historical data captured by VStore can be combined with online monitoring tools to improve troubleshooting efforts in the data center.

1 Introduction

Today's data center applications are increasingly concerned with fast and efficient processing of huge amounts of data. Web companies use data analytics to track and analyze user actions and provide real-time business decisions to improve user experience. Applications that store and process such vast amounts of data are no longer composed of single codes running on a few machines, but rather a complex set of many interconnected distributed systems, often across hundreds of machines in a data center. With cost savings and flexibility afforded by services like Amazon's EC2 and Elastic MapReduce [1, 2], these applications are now being deployed in virtualized environments as well.

As data scale increases and the push towards real-time data analytics continues, efficient monitoring and troubleshooting of these systems has become more important. Each component of a system may have different performance characteristics and failure scenarios. In traditional data centers, the scale of machines required to process data fast enough means hardware failures are common [10]. In cloud environments, performance can often be erratic due to virtualized resource contention with other cloud tenants [9]. To combat these issues, monitoring is often used as a tool for administrators to detect, diagnose, and fix failures quickly.

This paper outlines a method for combining fast and scalable monitoring of distributed applications with a scalable database backend for offline storage and analysis of monitoring data. Traditional monitoring applications use polling interfaces and centralized collection, providing a snapshot of overall system health via metrics collected over some period of time. However, these systems can have scalability issues, and lack fast anomaly detection required for real-time streaming data applications that are deeply ingrained in business logic. In our previous work, VScope [17], we developed a system that scales to thousands of machines and allows dynamic, real-time monitoring using online analytics for fast anomaly detection. In this paper we outline an extension to VScope, which we call *VStore*, that enables offline archiving of monitoring data gathered by VScope for historical analysis. This approach allows for online algorithms to be deployed dynamically to quickly detect problems, and to compare current observations to a rich set of historical data for targeted debugging.

The remainder of this paper is organized as follows: section 2 discusses problems with current solutions; section 3 outlines our system design; section 4 explores use cases from three common anomaly scenarios in a distributed weblog application, and discusses how VStore can aid troubleshooting efforts; and in section 5 we conclude and discuss future work.

2 Problem Description

Traditional data center monitoring systems such as such as Ganglia [5], Nagios [6], and OpenTSDB [8] provide for basic collection of simple metrics like CPU, memory, and network utilization from machines in a datacenter. These metrics offer a summary of global system health, but sometimes this view can be limited. In the case of Ganglia, for instance, a round-robin database limits the amount of data stored in a centralized place. OpenTSDB provides a scalable data storage backend, but still relies on simple polling of metrics. Although Nagios supports alerting, methods are typically based on heuristics such as pre-defined thresholds for each monitored metric.

Conversely, emerging methods for monitoring systems in the research literature such as Monalytics [15] and VScope [17] use a more targeted approach by aggregating data over a very large scale of machines, and performing online analysis for rapid detection of anomalies. These systems provide an opportunity to dynamically adjust the monitoring and analysis functions within the datacenter, but lack support for a rich history of monitoring data to compare against.

Each of these monitoring systems provides an invaluable set of information to data center administrators. System-wide monitoring applications provide a global overview of application and system health over longer time windows, whereas online algorithms provide extremely fast detection of anomalies and targeted debugging when problems arise. However, we lack a coherent bridge between these two paradigms. An administrator working on troubleshooting an issue in a complex distributed system would like to not only know *when* an anomaly is detected, but what *type* of anomaly the application is experiencing. Consider a distributed system like HBase, which is composed of other complex components (namely Hadoop and ZooKeeper). As noted in [14, 4], a number of different fault types can lead to irregular behavior in an HBase cluster. These range from a failing disk causing excessive I/O wait times at HDFS nodes, imbalanced key spaces causing ”hot spots” at region servers, or a failing NIC causing network requests to be delayed. Each scenario may exhibit subtly different anomalies at each tier, making classification difficult if data has been aggregated or summarized.

Bridging this gap will allow for faster detection and correction of faults in large, distributed systems. Anomalous behavior can first be detected with an online algorithm, then classified by gathering enough data from a set of monitored nodes and comparing it with historical data. This comparison guides the administrator down the right path for further debugging.

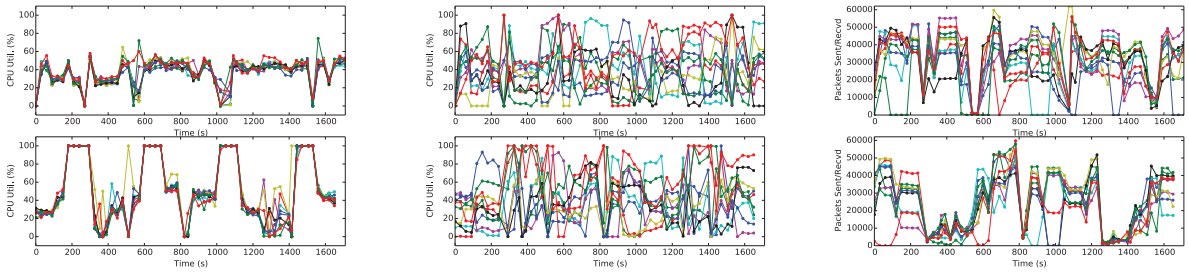
3 System Overview

We are working on integrating both offline storage and analysis of monitoring data with fast, scalable, online monitoring. We have built an extension to VScope [17] which adds a scalable database backend built on OpenTSDB [8], which we call *VStore*. This database allows us to bridge the gap between VScope's targeted, online analytics over short time windows, and the extended view of long-term interactions between application tiers provided by a database. An administrator can use VStore to draw upon a rich history of monitoring data to fine-tune online algorithms, and uncover anomalies that otherwise might be missed by VScope.

VStore integrates with VScope by hooking in to the online monitoring and analysis performed by Distributed Processing Graphs (DPG). First outlined in our previous works [19, 17], DPGs are essentially scalable overlay networks capable of being dynamically deployed and reconfigured with customized monitoring functions. For example, a ”watch” DPG may be deployed across all nodes to coarsely track low level metrics such as CPU and memory utilization, while a ”query” DPG might deploy on a subset of nodes to perform fine-grained application-level monitoring. To scale monitoring functions, DPGs support arbitrary aggregations, such as trees that aggregate values observed from child nodes in the monitoring graph. Additionally, VScope can run fast analytics on data aggregated in a DPG, which can be used to detect anomalies over a short time window. For instance, a DPG may gather 5 minutes worth of data and perform an entropy calculation (see [16, 20, 21]) to quickly identify CPU spikes or other anomalies, and alert an administrator of an impending fault.

To complement this, VStore can be deployed at the root of a DPG, where values are aggregated and analyzed. The root uses VStore to archive each individual observation before aggregation. Whereas an online process will throw away data after aggregating and analyzing over a short time window, VStore saves this history completely for later analysis. As a simple example, consider a DPG that aggregates CPU utilization at a number of nodes. The root of the DPG collects observations (perhaps once per second) from each node and buckets them over a sliding time window to perform an online analysis. At the same time, when each observation is received at the root, it sends it to VStore for archiving.

VStore acts as an OpenTSDB client, and provides a communications layer between custom monitoring code running inside DPGs and a scalable key-value store using HBase, the backbone of OpenTSDB. Monitoring code executing within a DPG that uses VStore can store any type of numeric metric that is being collected. To scale archiving at different points in the DPG, VStore can be



(a) CPU usage at HBase Region Servers **(b)** CPU usage at HDFS nodes **(c)** Network traffic at HDFS nodes

Figure 1: A periodic, high-CPU background task running at the HBase tier. The top graph shows a normal workload, while the bottom shows a periodic process starting and stopping every 5 minutes. A periodic high-CPU load at the HBase tier has relatively low impact on HDFS, making this type of anomaly more difficult to detect using a global online algorithm.

integrated at multiple levels. Each aggregation point in a DPG can thus use VStore to archive observations before either forwarding them or collapsing into a single value. Because it is backed by HBase, VStore can scale well to concurrent clients archiving a large amount of monitored data. This is critical to integrating with VScope's goal of providing low-overhead and scalable troubleshooting for over 1000 nodes.

4 Use Cases

To demonstrate how collection and offline analysis can be combined with online monitoring systems, we experimented with three typical scenarios seen in modern, multi-tiered data center applications: a periodic background process that interferes with normal processing, a misconfiguration in the software application itself, and a fault or misconfiguration at the network level. Our experiments are designed around a big data application, similar to that of [17], to collect and analyze web logs for micro-marketing purposes. We constructed a distributed log collection workload using Hadoop, HBase, and Apache Flume [3, 18].

Our experimental testbed is deployed on a local research cloud at Georgia Tech using the OpenStack platform [7], with virtual machines running Ubuntu Linux 12.10. We run an HDFS tier containing a Hadoop Master server, plus 10 VMs running data and task tracker processes. The HBase tier contains a Master server, plus 10 VMs acting as region servers, and is not colocated with HDFS nodes. 3 VMs are dedicated to a ZooKeeper quorum. Finally, 70 VMs serve as workload generators and aggregators for Flume. A workload generator runs a simple log generation process simulating a web server processing requests. These log records are sent via a Flume pipeline to sinks that connect directly to HBase to archive log records.

Each of the graphs in our results show two plots: the top plot shows data collected by VStore under normal conditions, while the bottom shows data for the same experiment with an anomaly introduced.

4.1 Periodic Processes

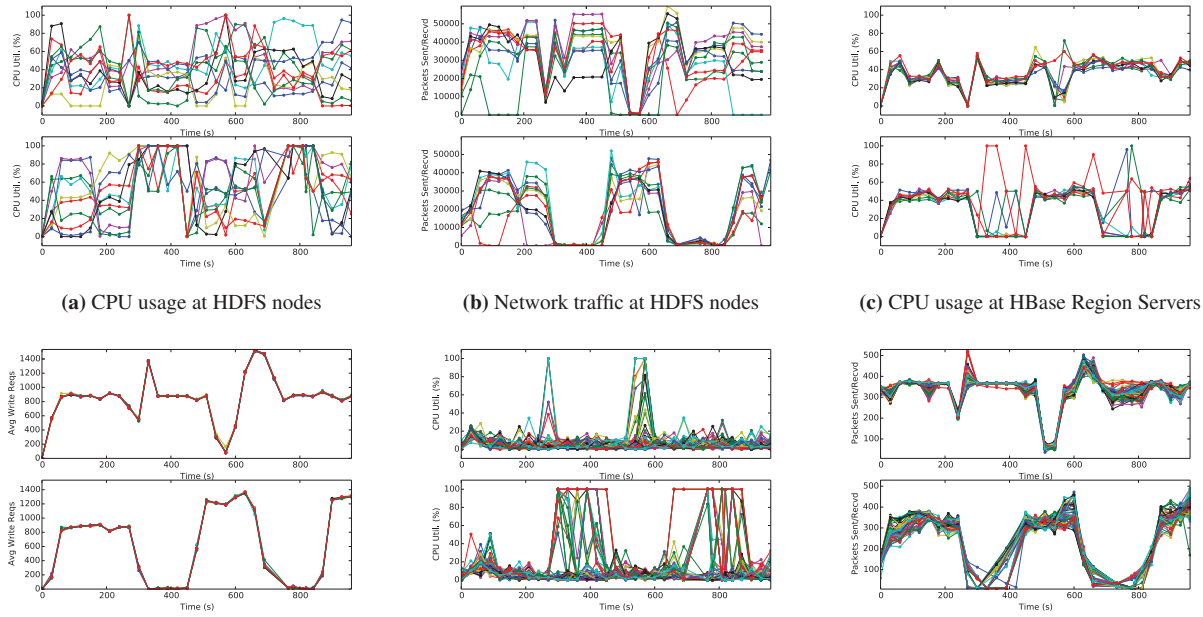
A common task for multi-functional servers in a data center is to periodically perform some short-lived operation. For example, a cron job may be used to clean up and archive system logs or application data once per day. In the case of our weblog application, HBase may be manually configured to perform major compactions on a periodic basis. Such tasks can be resource-intensive, and are typically scheduled at off-peak hours so as not to disturb other running applications. However, a misconfiguration or other fault may lead to such processes running at incorrect times.

Unfortunately, online monitoring algorithms may not detect and easily classify such cycles. As mentioned before, an online algorithm might only observe a short period of time during which a periodic process is not running. Furthermore, the observed window may not fully classify periodic behavior, for instance if only the beginning or end of a spike is observed.

To simulate a periodic process, we used the Linux "stress" utility to impose a load on certain servers, and measured the impact on metrics collected during this time. Each such stress test process runs every 5 minutes, for a duration of 2 minutes. We run 2 concurrent "stress" processes for a single resource, such as CPU or disk I/O.

4.1.1 CPU-intensive Periods

The first experiment simulates a periodic, high-CPU workload at the HBase tier. This is potentially common for major compactions which require frequent communication with HDFS nodes. Figure 1a shows the CPU utilization for each HBase region server. Each period of high-CPU activity is clearly identified, but only over a



(a) CPU usage at HDFS nodes **(b)** Network traffic at HDFS nodes **(c)** CPU usage at HBase Region Servers

(d) Avg write reqs at HBase Region Servers **(e)** CPU usage at Workload Generators **(f)** Network traffic at Workload Generators

Figure 2: A periodic, I/O intensive background task running on HDFS nodes. High disk I/O on HDFS nodes tends to starve many resources, and leads to anomalous behavior at all tiers

brief period of time. To properly correlate this behavior, an online algorithm would need an extended time window to distinguish between periodic spikes (such as those under normal load), and a deterministic process. Additionally, we can see from figures 1b and 1c that CPU and network utilization at other tiers remains relatively unaffected. Thus, an online algorithm that aggregates monitoring data across tiers may not efficiently detect the problem at the HBase tier alone.

4.1.2 Intense Disk I/O

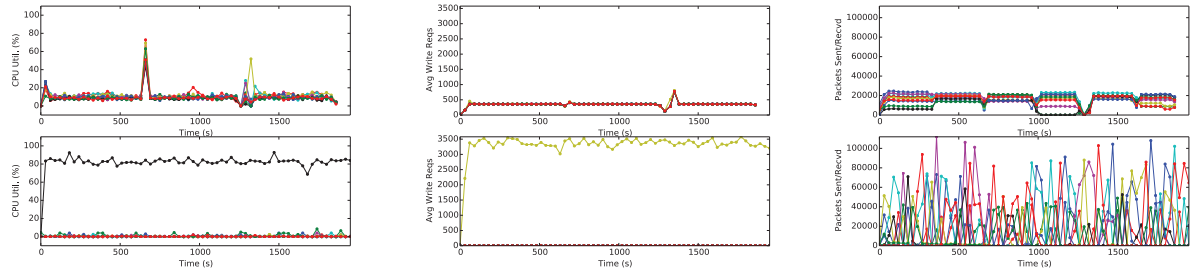
We also experimented with periods of high disk I/O at the Hadoop tier. This may be triggered by an external process (such as a periodic MapReduce Job), or an internal process (such as log cleanup). Additionally, as suggested in [13], anomalies in disk I/O could be caused by ”limpware,” failing or degraded hardware that causes excessive I/O latencies or a higher amount of disk reads compared to normal operation. For this test, we find that high disk I/O on HDFS nodes has a more drastic effect across tiers on different metrics.

Figures 2a and 2b compare CPU utilization and network flows at HDFS nodes. A spike in CPU utilization is visible during the period of high disk usage, but is difficult to distinguish from HDFS's normal CPU workload. However, we see a significant drop in packets sent and received during this time, indicating other tiers are being affected by the spike. Figures 2c and 2d show a significant change in CPU utilization and requests fielded by each region server, respectively, while figures 2e and 2f show a corresponding change at workload servers themselves. An online algorithm would need to observe both an extended time window and multiple metrics across tiers to detect this type of anomaly.

4.2 Software Misconfigurations

Another typical scenario involves a misconfigured or buggy software application which leads to load imbalance, crashes, or other anomalous behavior. In the case of HBase, one source of load imbalance stems from region splitting. By default, a client request is sent to the region server hosting the region that falls within a key range for the requested row. Tables are ”split” based on contiguous ranges of row keys. Although HBase will do this automatically, it is sometimes advantageous to pre-split tables to maintain a uniform distribution of requests across the cluster. For instance, if the row key is based on a hash function, each region server can be assigned portions of the hash space for balanced load distribution.

To emulate a misconfiguration under this scenario, we constructed a pre-split table in HBase using the `HexStringSplit` algorithm, which assumes that row keys are a uniformly distributed hexadecimal string. Flume agents write log records to HBase using a random UUID as the row key, providing even row key distribu-



(a) CPU Utilization at HBase Region Servers **(b)** Avg write reqs at HBase Region Servers **(c)** Network traffic at HDFS nodes

Figure 3: A software misconfiguration at the Flume tier. The HBase table is pre-split using `HexStringSplit`, and agents use a random hexadecimal string for row keys. Misconfigured agents use a timestamp instead, causing unbalanced regions.

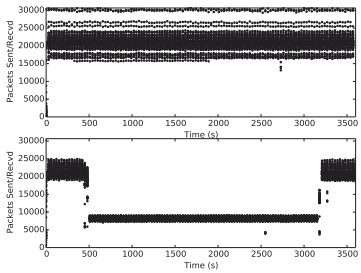
tion. An anomalous configuration uses a timestamp instead, leading to load imbalance as keys increase monotonically.

Figures 3a and 3b compare CPU utilization and write requests at HBase region servers, respectively. These figures show a clear imbalance among region server load when timestamps are used as the row key. From figure 3c, we also see how this can have a drastic effect on HDFS communications. This type of anomaly is distinct from periodic process interruptions, but could be missed by an algorithm that aggregates observations (for instance, calculating an average). However, the individual data captured by VStore clearly distinguishes the two scenarios.

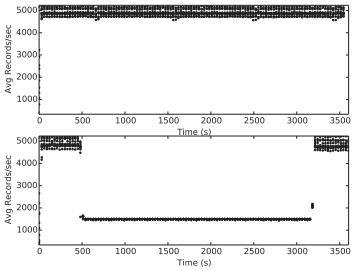
4.3 Network Faults

A faulty or misconfigured network interface is a common problem in data centers. Such a fault may stem from a variety of issues, such as a physical NIC incorrectly negotiating a link speed with a switch (see [4]), or a software-defined networking layer misconfiguring the bandwidth provisioned to a VM. As noted in [13], network hardware can also suffer ”limpware” anomalies - a faulty adapter may drop or corrupt packets periodically, forcing error correction mechanisms to be used and reducing overall bandwidth on the link. Sustained network bandwidth is important for a variety of applications. For instance, many web applications run realtime analytics on incoming requests at a high speed (such as calculating ”trending” topics). Faults at the network layer can thus directly affect an application’s business logic.

To simulate this, we configured 60 flume agents to generate an 8KB JSON object 5000 times per second (simulating web server requests). These requests are forwarded to 10 aggregator Flume agents, which count the number of records processed. Each server has a 1Gbps LAN connection; we simulate an anomaly by limiting bandwidth to 100Mbps at the 10 aggregation nodes.



(a) Network traffic at Flume agents



(b) Avg rate of records processed by Flume aggregation agents

Figure 4: Simulating a network misconfiguration. A real-time streaming aggregation application counts requests collected from a front-end tier. A VLAN misconfiguration or other network fault may limit bandwidth to a critical set of nodes, leading to reduced throughput throughout the application.

Figure 4a shows the network traffic for each server. With a faulty network link, there is a precipitous drop-off in network traffic across the cluster, as other agents become limited in the amount of work they can send to the aggregation tier. This situation also highlights how monitoring application metrics can provide useful anomaly classification information. Figure 4b shows the rate of records processed per second at each aggregation Flume agent. When bandwidth is scarce, agents are severely limited in the amount of records they can process.

4.4 Discussion

The data collected in our experiments indicate that there are uses for combined online and offline monitoring in a number of real-world debugging scenarios. Ideally, an operator troubleshooting a problem should be able to leverage both current and past data efficiently to rapidly identify and fix problems when the occur. Some opportunities for VStore to assist in these efforts include:

- *Targeted debugging with full history*: A strength of VScope is the ability to guide troubleshooting and monitoring efforts to the right servers when an anomaly is detected. VStore can integrate with VScope to offer automatic snapshots of historical data to aid in the troubleshooting process. A *baseline watch DPG* runs continuously in the datacenter and uses VStore to capture basic performance metrics, such as CPU and network utilization. When a problem occurs, the operator may use a combination of VScope's *watch*, *scope*, and *query* operations to locate the affected machines (refer to section 4.1 of [17] for an example). A new watch DPG is deployed on these machines to capture data about the current scenario. The *VShell* running this new watch presents the operator with current data, and automatically pulls snapshots of historical data from OpenTSDB for comparison - for instance data from the same time period for the past 10 days. The end result is that the operator has quickly gained knowledge about how a current problem relates to past events, which can help guide debugging efforts.

- *Detecting "limplock"*: The authors of [13] suggest that efficient detection through monitoring is one method of mitigating "limplock" - a situation where failing or degraded hardware causes systems to slow considerably, but does not trigger normal fail-stop mechanisms. VStore and VScope could be used as tools to detect limplock by providing a historical reference to help distinguish between overload-induced slowdown and failing hardware. VScope DPGs can be used to target monitoring on nodes and metrics known to cause limplock (such as HDFS write performance at Hadoop reducers). The online monitoring is then compared with historical data collected by VStore to determine if the system is limplocked, or simply experiencing a higher-than-usual load.

- *Classification of anomalies*: The rich set of historical data collected by VStore could be analyzed to help automatically classify anomalies, for instance using support vector machines [12] or statistical approaches like that of [11]. When combined with the targeted debugging approach outlined above, VScope could then query this data to attempt to automatically identify the type of anomaly currently taking place. If the current fault scenario can be classified using historical data, this helps the operator identify a debugging approach much more rapidly.

5 Conclusion and Future Work

In this paper we described VStore, an extension to our previous work VScope, which aims to bridge the gap between fast and flexible online monitoring systems with large-scale data collection for historical analysis. Our experiments show that VStore's capabilities in archiving fine-grained monitoring data across a large cluster of systems can help pinpoint hard-to-find anomalies that an online anomaly detection algorithm might miss. We have evaluated three common fault scenarios, and discussed how VStore's data archiving capabilities can be combined with VScope's online monitoring to complement debugging efforts for real-world scenarios.

A main goal of our future work involves a more complete integration of VStore with VScope's DPGs and online algorithms, to allow historical data to be captured dynamically at different points in the graph. This integration will allow us to begin exploring some of the use cases for fully integrated online and offline monitoring outlined in the discussion in section 4.4. In addition to monitoring and detecting application-level performance anomalies, we are also exploring ways to use VStore to detect infrastructure faults in cloud systems, such as a misconfigured SDN causing performance anomalies across cloud tenant applications.

A thorough performance evaluation of VStore is also needed. Results from [17] indicate that VScope can scale well to thousands of nodes, and we believe VStore also scales well as it is backed by HBase. Our experience with the experiments in this paper suggest the overhead of our system is small, but a detailed evaluation of VStore's scalability and perturbation when run along side real-time streaming data applications is required to quantify this.

Acknowledgements

We thank the anonymous reviewers for their insightful comments and feedback helping to improve this work.

References

[1] Amazon Elastic Compute Cloud (Amazon EC2). `http://aws.amazon.com/ec2`. Accessed: December 9, 2013.

[2] Amazon Elastic MapReduce (Amazon EMR). http://aws.amazon.com/elasticmapreduce. Accessed: December 9, 2013.

[3] Apache Flume. http://flume.apache.org. Accessed: December 7, 2013.

[4] The apache hbase reference guide. http://hbase.apache.org/book/book.html. Accessed: December 9, 2013.

[5] Ganglia Monitoring System. http://ganglia.sourceforge.net. Accessed: December 7, 2013.

[6] Nagios. http://www.nagios.org. Accessed: December 7, 2013.

[7] OpenStack Cloud Software. http://www.openstack.org. Accessed: December 7, 2013.

[8] OpenTSDB. http://opentsdb.net. Accessed: December 7, 2013.

[9] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, Apr. 2010.

[10] L. A. Barroso and U. Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis Lectures on Computer Architecture*, 4(1):1–108, 2009.

[11] P. Bodik, M. Goldszmidt, A. Fox, D. B. Woodard, and H. Andersen. Fingerprinting the datacenter: Automated classification of performance crises. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 111–124, New York, NY, USA, 2010. ACM.

[12] N. Cristianini and J. Shawe-Taylor. *An introduction to support vector machines and other kernel-based learning methods*. Cambridge university press, 2000.

[13] T. Do, M. Hao, T. Leesatapornwongsa, T. Patana-anake, and H. S. Gunawi. Limplock: Understanding the impact of limpware on scale-out cloud systems. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 14:1–14:14, New York, NY, USA, 2013. ACM.

[14] L. George. *HBase: the definitive guide*. O'Reilly Media, Inc., 2011.

[15] M. Kutare, G. Eisenhauer, C. Wang, K. Schwan, V. Talwar, and M. Wolf. Monalytics: Online monitoring and analytics for managing large scale data centers. In *Proceedings of the 7th International Conference on Autonomic Computing*, ICAC '10, pages 141–150, New York, NY, USA, 2010. ACM.

[16] C. Wang. Ebat: Online methods for detecting utility cloud anomalies. In *Proceedings of the 6th Middleware Doctoral Symposium*, MDS '09, pages 4:1–4:6, New York, NY, USA, 2009. ACM.

[17] C. Wang, I. A. Rayan, G. Eisenhauer, K. Schwan, V. Talwar, M. Wolf, and C. Huneycutt. VScope: Middleware for Troubleshooting Time-Sensitive Data Center Applications. In *ACM/IFIP/USENIX International Conference on Middleware (Middleware)*, 2012.

[18] C. Wang, I. A. Rayan, and K. Schwan. Faster, Larger, Easier: Reining Real-Time Big Data Processing in Cloud. In *ACM/IFIP/USENIX International Conference on Middleware (Middleware)*, 2012.

[19] C. Wang, K. Schwan, V. Talwar, G. Eisenhauer, L. Hu, and M. Wolf. A flexible architecture integrating monitoring and analytics for managing large-scale data centers. In *Proceedings of the 8th ACM International Conference on Autonomic Computing*, ICAC '11, pages 141–150, New York, NY, USA, 2011. ACM.

[20] C. Wang, V. Talwar, K. Schwan, and P. Ranganathan. Online Detection of Utility Cloud Anomalies Using Metric Distributions. In *IEEE/IFIP Network Operations and Management Symposium (NOMS)*, 2010.

[21] C. Wang, K. Viswanathan, L. Choudur, V. Talwar, W. Satterfield, and K. Schwan. Statistical Techniques for Online Anomaly Detection in Data Centers. In *IFIP/IEEE International Symposium on Integrated Network Management (IM)*, 2011.

An Enterprise Dynamic Thresholding System

Mazda A. Marvasti, Arnak V. Poghosyan, Ashot N. Harutyunyan, and Naira M. Grigoryan
Management BU
VMware Inc.
{mazda;apoghosyan;aharutyunyan;ngrigoryan}@vmware.com

***Abstract*— We demonstrate an enterprise Dynamic Thresholding System for data-agnostic management of monitoring flows. The dynamic thresholding based on data historical behavior enables adaptive and more accurate control of business environments compared to static thresholding. We manifest the main blocks of a complex analytical engine that is implemented in VMware vCenter Operations Manager as a principal foundation of the company's data-driven anomaly detection.**

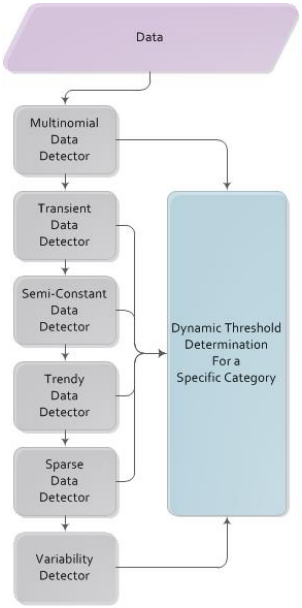
Keywords - monitoring; time series data; dynamic thresholding; data categorization; parametric and non-parametric statistics.

I. INTRODUCTION

Modern enterprise IT management becomes increasingly smart to proactively respond to the performance issues that complex infrastructures necessarily encounter. The management based on expert knowledge utilization is no longer efficient. Monitoring and data measuring of the processes governing the entire IT is a fundamental approach to gain insights from those sophisticated environments with complicated interrelation between the constituent components. The history of this approach goes back to statistical process control [1]. Since contemporary business infrastructures are highly dynamic (and out of classical Gaussian normalcy domain [2]), static thresholding of processes and performance indicators become inadequate. Hence problem diagnostics dictates a soft control of IT environments ([3-14]). In this paper, we follow a path where anomaly detection is based on prediction of upper and lower dynamic thresholds (normalcy range) of categorized data that vary over time [15-19].

VMware's vCenter Operations Manager (vC Ops) [20] is an industry-leading enterprise solution in area of IT management that encodes the bunker of monitoring data from customer IT system into a real knowledge for anomaly detection and problem root cause identification, as well as capacity planning for modern virtualized and cloud computing ecosystems. In this paper we introduce an enterprise data-agnostic dynamic thresholding system (EDTS) that enables vC Ops to act as an atomistic anomaly detection and forecasting of monitoring flows. The data-agnosticism (indicates that data analysis and determination of its normalcy behavior is performed without knowing the essence of the underlying physical and business service processes) enables an universal platform for processing of very large data sets, at the same time, it can lead to a deadlock if the statistical methods are not sufficiently powerful to handle diversity of monitored data types. Our analysis of customer data over several years show that deficiency of data-agnosticism can be compensated by appropriate data categorization, since a specific data category statistically characterizes the underlying process and empowers an efficient construction of relevant normalcy bounds (dominant behavior) thus reliably controlling the flow. This concept leads to an EDTS based on data categorization realized in vC Ops. A simplified and specific realization of EDTS adjusted for IT environments is presented in Flowchart 1. Although selection of the categories is adapted to some IT customer preferences, the overall approach is applicable widely (also out of the IT interests) with appropriate modification of categories and their definition parameters.

Experimental results justify EDTS's potential to effectively handle large infrastructures in terms of both accuracy and complexity. All ideas described in the sequel are filed as a patent [21].



Flowchart 1. A simplified principal scheme of EDTS.

EDTS sequentially utilizes different data categorization detectors that allow choosing the right algorithm for determination of data dynamic thresholds (DT's). The categorization order or the hierarchy is important as different orders of iterative checking and identification will lead to different categories with differently specified normalcy states. The system presented in Flowchart 1 categorizes data as Multinomial, Transient, Semi-constant, Trendy, Sparse, High-Variability or Low-Variability. In each of those cases the normalcy determination method is different. In all categorization scenarios the data additionally is verified against *periodicity* for efficient construction of its normalcy bounds.

Moreover, in each of the majority of categorization scenarios data is processed by category-specific *change detection* procedures. The functional meanings of the above mentioned detectors are as follows:

Multinomial Data Detector searches for Multinomial data which takes only integer values after checking errors introduced by the monitoring apparatus. If data is identified as multinomial then DT determination module calculates specific for this category DT's otherwise data transmits to the next detector.

Transient Data Detector looks for Transient Data which can be characterized as multimodal data. If data is identified as transient then DT calculation module performs DT calculation separately for each mode.

Semi-Constant Data Detector checks the data against its "almost constant" behavior. If data is not semi-constant but its latest portion satisfies the category specifications after a global change, then DT construction module performs DT calculation for the latter. Piecewise constant data is from this category.

Trendy Data Detector performs trend identification. If data is trendy then detector classifies trend as linear or non-linear and DT determination module executes a special DT calculation algorithm. If data is not trendy but its latest portion can be selected as trendy (change occurred), then DT determination module performs calculations for that portion.

Sparse Data Category Detector explores data gaps, their amount, and distribution in time. Data goes to the next detector for further analysis if overall gap duration is negligible. Data is classified as Sparse if gaps have uniform distribution in time. If gaps have some accumulation and remaining data is acceptable for further analysis then the selected portion goes to the next detector.

Variability Detector categorizes data either High-Variability or Low-Variability with specific DT calculation procedures. Before final categorization data passes through a change detection procedure for selection of the latest statistically stable portion for final DT determination.

II. Data Categorization

Multinomial Data Detector. This detector calculates some statistical parameters for comparison with the predefined measures. If the check is positive then data is classified as *Multinomial Data*. It is assumed that Multinomial data takes only integer values. Let p_j be the frequency of occurrences of the integer n_j

$$p_j = \frac{n_j}{N}100, \qquad j = 1, \dots, m$$

where N is the total number of integer values and m is the number of different integer values. Data is multinomial if it takes less than m different integer values and at least s of them have frequencies greater than parameter H_1.

Some integer values with small cumulative percentages can be discarded. This can be done by sorting the percentages p_j in descending order and by defining the cumulative sum c_j

$$c_1 = 100,\ c_j = p_j + \cdots + p_m,\ c_m = p_m.$$

Then, if $c_k < H_2(= 0.5\%)$, $c_{k-1} \geq H_2$ the integer values $n_k, n_{k+1}, \dots, n_m$ can be discarded.

Transient Data Detector. *Transient Data* is categorized by multimodality, modal inertia, and randomness of modes appearing along the time axis. *Transient Data* must have at least two modes. Modal inertia means that data points in each mode must have some inertia and they can't oscillate from one mode to the other "quickly". Actually the inertia can be associated with the time duration that data points remain in the selected mode. Categorization is performed by calculation of some transition probabilities. We omit the relevant details from [21]. A similar technique is applied in *Sparse Data Detector* (see below).

Figure 1 shows an example of a *Transient Data*.

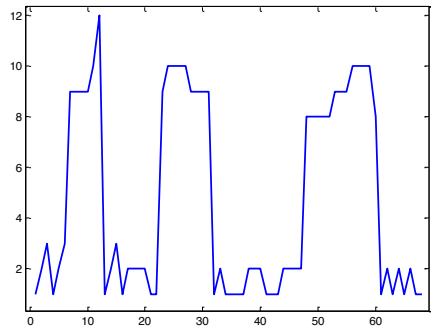


Figure 1. Example of a Transient Data.

Semi-Constant Data Detector. Data is categorized as *Semi-Constant* if

$$iqr(data) = 0$$

where iqr stands for the interquartile range of data. If data is not from the required category but the latest enough long portion satisfies the condition then it is selected for further dynamic threshold determination as *Semi-Constant Data*.

Figure 2 shows an example of *Semi-Constant Data*.

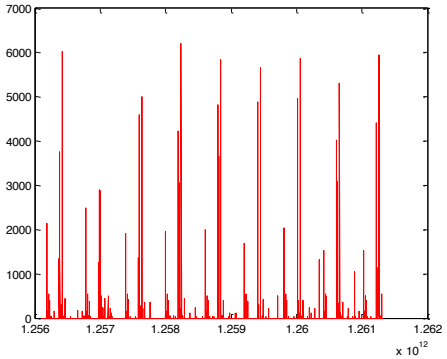


Figure 2. Example of Semi-Constant Data.

Trendy Data Detector. Different classical methods are known for trend determination. Mann-Kendall [22,23] test is appropriate for our purposes although other known tests are also possible to apply. The test categorizes data either *Trendy* or *Non-Trendy*. In case of *Trendy Data* further analysis categorizes the trend into linear and non-linear. Linearity can be checked by the well-known linear regression. If data is *Linear-Trendy* then DT determination module performs a specific DT calculation. If data is not *Linear-Trendy* but the

latest rather long portion of data has linear trend, then we select it for further DT determination.

Sparse Data Detector explores data in terms of gaps. If the total percentage of gaps is higher than some limit and they have non-uniform distribution in time (it means that gaps have some localization in time) then gap clean up (data selection) procedure will return a regular data for further categorization. If gaps in data have uniform distribution in time then data belongs to a *Sparse Data* category. If gaps in data have extremely high percentage that further analysis is impossible. Data categorization is based on the following measures: 1) percentage of gaps, 2) transition probabilities for gap-to-gap, data-to-data, gap-to-data and data-to-gap. Probability calculation is starting with data monitoring time (Δt) estimation $\Delta t = median(\Delta t_k)$, $\Delta t_k = t_{k+1} - t_k$. Time intervals with $\Delta t_k \le c\,\Delta t$ are normal data intervals while $\Delta t_k > c\,\Delta t$ are gaps. c is a parameter for gap definition. Let T_k be duration (in milliseconds, seconds, minutes, etc., but in the same measures as the monitoring time) of the k-th gapless data portion. For data without gaps we have only one such portion and $T_k = t_N - t_1$. The sum $T = \sum_{k=1}^{N_T} T_k$ is the duration of gapless data where N_T is the count of gapless data portions. Let G_k be duration (in the same measure as T_k) of the k-th gap. The sum $G = \sum_{k=1}^{N_G} G_k$ is the duration of all gaps in data and N_G is the count of gap portions. Obviously $G + T = t_N - t_1$. By ρ we define the percentage of gaps in data

$$\rho = \frac{G}{G+T}100\%.$$

Now, by $p_{11}, p_{10}, p_{00}, p_{01}$ we define the probabilities of data-to-data, data-to-gap, gap-to-gap and gap-to-data transitions, respectively

$$p_{11} = 1 - \frac{N_T}{\frac{T}{\Delta t}},\ p_{10} = 1 - p_{11},$$

$$p_{00} = 1 - \frac{N_G}{G/\Delta t},\ p_{01} = 1 - p_{00}.$$

Data with gaps non-uniformly distributed in time can be specified by the condition

$$\begin{cases} \rho > H_1 \\ \mathrm{p}_{10} < \varepsilon \\ \mathrm{p}_{01} < \varepsilon \end{cases}$$

where the following values of parameters can be reasonably chosen $H_1 = 25\%$ and $\varepsilon = 0.0005$. The main reason for smallness of p_{10} and p_{01} is the smallness of the numbers N_T and N_G while G and T are as big as ρ is assumed. Data from this category can be further processed via data selection procedure that will eliminate (if possible) concentration of gaps. This can be done as follows: calculate the total percentage of gaps in the series of data $\{x_k\}_{k=i}^{j}$, $j = j_1, j_2, \dots, j_s$, $i = i_1, i_2, \dots, i_r$, and select the portion for which $\rho \le H_1$. The selected data is ready for further analysis by sequential detectors.

Data with gaps uniformly distributed in time (*Sparse Data*) can be specified by the condition ($H_2 = 60\%$)

$$\begin{cases} \rho > H_2 \ge H_1 \\ \mathrm{p}_{10} \ge \varepsilon \\ \mathrm{p}_{01} \ge \varepsilon \end{cases}$$

The second and third conditions mean that gaps are uniform in time and technical cleanup is impossible.

Data is useless for further analysis if $\rho > H_3$ ($= 95\%$).

Figure 3 shows an example of *Sparse Data* with the corresponding measures for categorization.

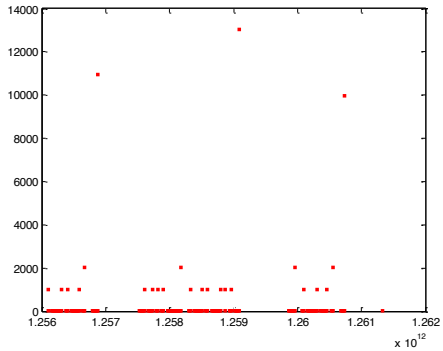


Figure 3. Sparse Data. Here $\rho = 68\%$, $p_{11} = 0.9957$, $p_{10} = 0.0043$, $p_{00} = 0.9979$, $p_{01} = 0.0020$.

Variability Detector calculates variability indicators and categorize data into High-Variability or Low-Variability. Based on the absolute jumps x'_k of data points

$$x'_k = |x_{k+1} - x_k|$$

the following measure R of variability is considered

$$R = \frac{iqr(\{x'_k\}_{k=1}^{N-1})}{iqr(\{x_k\}_{k=1}^{N})}100\%, \quad iqr(\{x_k\}_{k=1}^{N}) \neq 0.$$

Then, if $R \le V$ then data is Low-Variability, otherwise High-Variability. Figure 4 shows an example of Low-Variability data with $R = 0$.

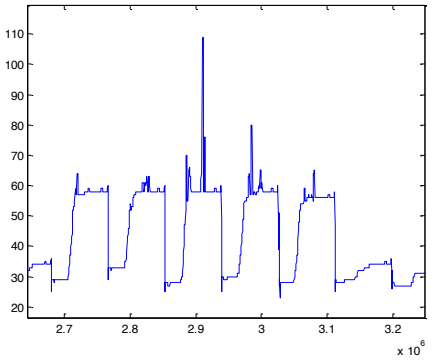


Figure 4. Low-Variability data with $R = 0\%$.

III. Category-Specific DT Determination

As we mentioned above, each data category preliminary passes through some period determination procedure which additionally categorizes data into *Periodic* and *Non-Periodic*. Details of this procedure are presented in [21]. We describe only a high level concept. The period determination is seeking similar patterns in the historical behavior of time series for setting the DT's based on the discovered cyclical information. The algorithm consists of two main steps:

1) Data *Footprint* calculation which provides with two-dimensional distribution of time series based on some predefined frame. First, we calculate percentages of data in each cell of the frame and then we get the corresponding

distribution by taking cumulative sums of those percentages for each column of the frame. More specifically, the range of data (or the range of preliminary smoothed data) is divided into non-uniform parts by quantiles q_k with $k = k_1, \dots, k_m$, $0 \le k_1 < \cdots < k_m \le 1$, with some m and k_j. Evidently, the grid lines are dense where data is dense. For division of data into parts along the time axis two parameters "$time\_unit$" and "$time\_unit\_parts$" are used. "$Time\_unit$" is a basic parameter that defines the minimal length of possible cycle that can be found. Moreover, any cycle can be a factor only of the length of the "$time\_unit$". Usual setting is $time\_unit = 1\,day$. Parameter "$time\_unit\_parts$" shows the number of subintervals (columns in the frame) that "$time\_unit$" must be divided. Actually this parameter is the measure of resolution. The bigger the value of "$time\_unit\_part$", the more sensitive is the footprint of the historical data.

2) *Pattern recognition* procedure which provides with *Cyclochart* of data by comparing different columns of the Footprint in terms of similarity. Figure 5 shows an example of a *Cyclochart* where y-axis shows the measure of confidence that data has some T-days cycle.

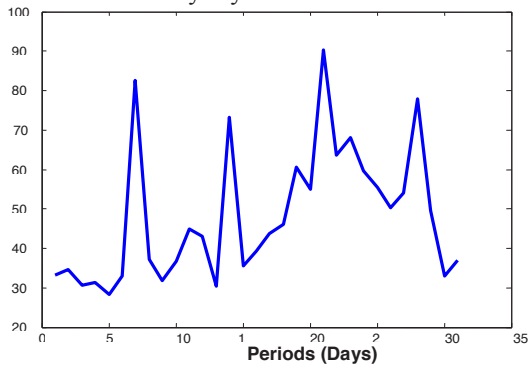


Figure 5. Example of a Cyclochart.

Further investigation of the *Cyclochart* categorizes data into *Periodic* or *Non-Periodic*. If data classifies as *Periodic* then the method provides with information on cycle length and outputs the frame columns in terms of similarity that are finally employed to quantify the time-based DT values. Figure 6 shows an example of a *Periodic Data* (blue curve) with the corresponding upper (red curve) and lower (green curve) DT's.

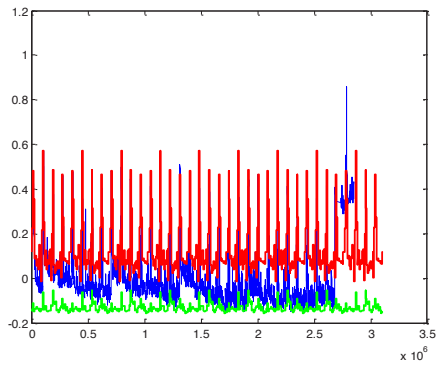


Figure 6. DT's of a Periodic Data.

Now we describe several category-specific DT construction mechanisms:

DT's of Multinomial Data. As mentioned, period determination investigates the cyclicality of data and classifies it into periodic and non-periodic categories. The general scheme for period determination in this case is specialized with the following modification while constructing the *Footprint* of data: instead of the percentages of data in every cell we are taking the values of c_k (see categorization of the *Multinomial Data*) in every column of the frame. If data is claimed *Periodic* then the normalcy set for similar columns are calculated as follows. Data points in similar columns are collected together and corresponding new values of the numbers c_k are calculated. If $c_{k+1} < H$, $c_k \ge H$ then the values $n_1, n_2, \dots, n_k$ constitute the most probable set (normalcy set) of similar columns. If *Multinomial Data* is determined as Non-Periodic then the numbers c_k are calculated for all data points and normalcy set is determined similarly.

DT's of Semi-Constant Data. For Semi-constant data every data point greater than $q_{0.75}$ (quantile) or less than $q_{0.25}$ is an outlier. If the percentage of outliers is greater than $p\%$ ($p = 15\%$), then we check for periodicity in outlier data by the procedure described above. For periodicity analysis data points equal to the median are excluded from the analysis.

DT calculation for *Non-Periodic Semi-Constant Data* is performed separately for upper (for data points that are greater or equal to median) and lower (for data points that are less than or equal to median) parts of data. Since the process of obtaining of both upper and lower bounds are similar, we'll explain the method only for the upper DT. The main principle is maximization of an objective function

$$g(P,S) = e^{aP}\frac{S}{S_{max}}$$

where $a > 0$ is a sensitivity parameter, for example $a = 0.9$; P is the percentage of data points within the median of data and any upper line higher than the median (see Figure 7);

$S_{max} = (t_{max} - t_{min})(Upper\ Line - Data\ Median)$

and S is the square of the area within data points and data median.

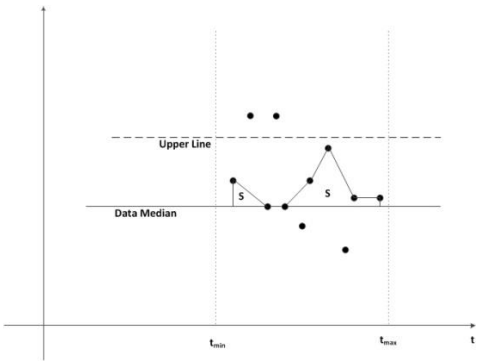


Figure 7. Auxiliary drawing for the objective function.

We consider two different approaches for determination of DT's via maximization of the objective function: data range and data variability based. In the data-range-based analysis, we divide the range within median and maximum of data

(while determining the upper bound) into m parts and for each level calculate the values $g_k, k = 1,2, \dots m$ of the objective function. Then, the level that corresponds to $\max(g_k)$ gives the appropriate upper bound. Instead of dividing the range into equal parts it is reasonable also to divide the range of data by corresponding quantiles that will give unequal division according to the density of data points along the range. Here preliminary abnormality cleaning of data can be performed. For this, removal of data points with abnormal concentration in the given time window is performed. Abnormal concentration can be detected by the following procedure. For the given time window (for example 10% of data length), we calculate the percentage of data points with values higher than 0.75-quantile. Then by moving this window along data, we calculate the corresponding percentages. Any percentage higher than the upper whisker indicates abnormal concentration and must be discarded from further calculation. We repeat the same abnormality cleaning procedure for data points lower than 0.25-quantile.

In the data-variability-based approach, we calculate the variability of data points x_k against median of data μ

$$v = \left(\frac{1}{N-1} \sum_{k=1}^{N} (x_k - \mu)^2 \right)^{1/2}$$

and consider the following set of upper lines

$$[\mu + z_j \, v], \quad j = 1,2, \dots$$

For each level, we calculate the corresponding values g_j of the objective function as described above and, we take the level that corresponds to $\max(g_j)$ as the appropriate upper DT. The following values can be used for z_j

$$z_1 = 1, z_2 = 1.5, z_3 = 2, z_4 = 3, z_5 = 4.$$

In case of periodic data the same procedure is applicable for each periodic column of the *Footprint* of data.

DT's of Transient Data can be obtained by similar procedure for each mode separately based on maximization of the objective function as we do it for $f(t)$ below.

DT's of Linear-Trendy Data. In case of *Linear-Trendy Data,* we perform decomposition of the original data $f_0(t)$ into the following form

$$f_0(t) = f(t) + kt + b$$

and perform DT calculation for $f(t)$based on the following objective function

$$g(P,S) = \frac{e^{a\,P} - 1}{e^{a} - 1} \frac{S}{S_{max}}$$

where S is the square of the area limited by t_{min}, t_{max} and some lower and upper lines (see Figure 8),

$$S_{max} = h(t_{max} - t_{min})$$

and P is the fraction of data within upper and lower lines and a is a user defined parameter. Then we calculate standard deviation $\sigma$of $f(t)$ and consider the following set of lower and upper lines

$$[kt + b - z_j \, \sigma, kt + b + z_j \, \sigma], \quad j = 1,2, \dots$$

Next we calculate g_j and take the level corresponding to $\max(g_j)$. We use the following values for z_j

$$z_1 = 1, z_2 = 1.5, z_3 = 2, z_4 = 3, z_5 = 4.$$

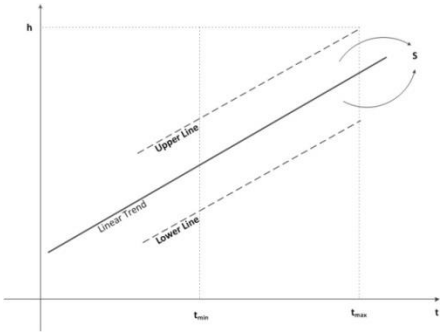


Figure 8. Auxiliary drawing for definition of the objective function.

Figures 9 shows an example of *Linear-Trendy Data* with the corresponding DT's.

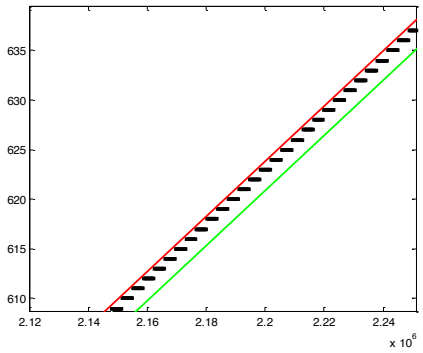


Figure 9. *Linear-Trendy Data* with the corresponding DT's**.**

DT's of Sparse Data. For period determination procedure, we put "*time_unit_parts*" $= \left[\frac{\text{"time\_unit"}}{median(T_k)+median(G_k)} \right]$. If data is classified as *Periodic* then DT calculation is performed according to the found cycles otherwise DT's can be determined based on the utilization of the objective function.

DT's of High- and Low-Variability Data. First data is checked for periodicity by setting different preliminary parameters while calculating the *Footprint* of data – less sensitive for High-Variability data, then DT determination is performed based on cycles or objective function utilization.

Figure 10 shows an example of *Low-Variability Data* with the corresponding normalcy bounds.

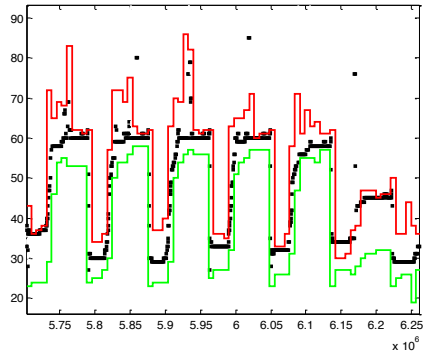


Figure 10. Data from Figure 4 with upper and lower DT's.

IV. System Validation

The results obtained for a specific customer data by EDTS are presented below. Note that Figures 6, 9, and 10 are also real enterprise examples. We selected some 3215 monitoring metrics with one month length and applied the categorization procedures. Table 1 shows the distribution along different data categories and Table 2 indicates the count of periodic and non-periodic data. In particular, for semi-constant data we observe the following picture. From 532 metrics (see Table 1) 267 have percentage of outliers less than 15% and they are claimed as non-periodic without any further checking. The remaining 235 metrics are checked for periodicity and in 212 cases periods are found. It is worth noting that results obtained for the specific customer data can't be in any manner generalized to other cases. The graphs below demonstrate several snapshots from the product with monitored time series data and their adaptively updated DT's by EDTS. We observe reliably detected DT's, data changes, periods, and relevant out-of-normal areas (yellow) reported as alarms.

Table 1. Distribution along the categories.

Data Category	Count (Percentage) of Metrics in the Category
Multinomial	724 (22.5%)
Trendy	165 (5.1%)
Semi-Constant	532 (16.5%)
Transient	102 (3.2%)
Sparse	88 (2.7%)
Low-Variability	826 (25.7%)
High-variability	669 (20.8%)
Corrupted	109 (3.4%)

Table 2. Count of periodic and non-period data.

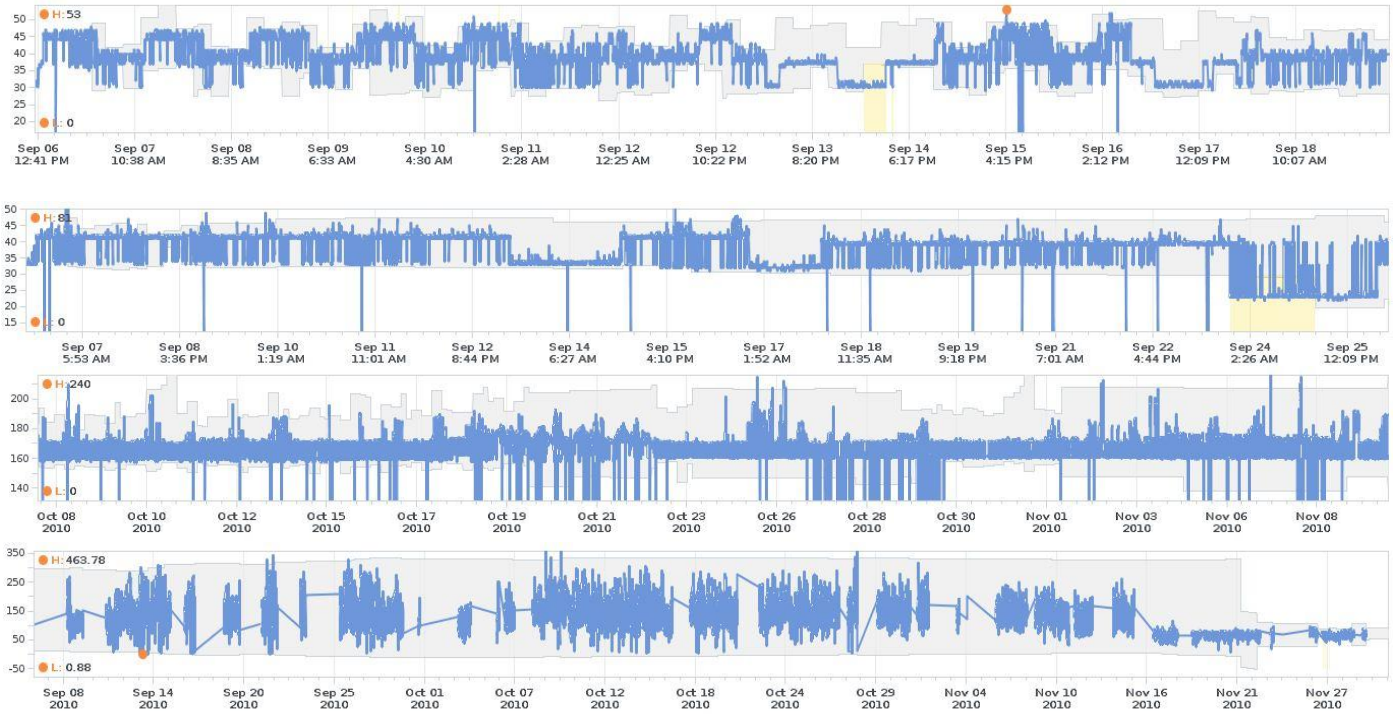
Periodic	Non-Periodic	Corrupted	Overall
1511	1595	109	3512

V. Related Work

In terms of our application, the performance of EDTS is estimated according to users experience on indicative and missed alarms, as well as the generated noise level that the useful information is embedded in. In this context, our categorization techniques allow achieving essentially better trade-off between the produced recommendation (alarm) noise and its accuracy in problem indication. That would not be possible with classical parametric approaches including Fourier transform, discrete Fourier transform [24-28], Prony's method [29,30]) as well as with other common purpose enterprise algorithms (including our algorithm of Section III that produces DT's based on data footprint even when cyclical patterns are not discovered). Moreover, the categorization in terms of those specific classes enables an efficient root cause analysis [31,32] based on the abnormality events (DT violation alarms) space that our system outputs. Furthermore, [19] reports about reliably predicted root causes of suddenly occurring influential outages at large enterprise infrastructures. This method relies on historically analyzed mutual impact factors of out-of-DT events.

Note that EDTS handles only structured monitoring data. For the unstructured data sets (like log files) we have developed a graph-based approach [33,34] that extracts the dominating correlation pattern between the main event types in data as dynamic normalcy structure and applies it to identification of "large"/abnormal deviations from that structure to determine performance anomalies.

Finally, we refer the reader to the papers [35,36] which outline the approaches and trends of the area of anomaly detection up to the recent days.



REFERENCES

[1] D.J. Wheeler, and D.S. Chambers, *Understanding Statistical Process Control*, Knoxville, TN: SPC Press, 1986.

[2] M.A. Marvasti, "How normal is your data?," *VMware technical white paper*, http://www.vmware.com/files/pdf/vcenter/-VMware-vCenter-Operations-How-Normal-Is-Your-Data-WP-EN.pdf, 2011.

[3] J.P. Buzen and A.W. Shum, "MASF: multivariate adaptive statistical filtering," in *Int. Computer Measurement Group (CMG) Conf.*, Nashville, TN, USA, Dec. 4-8, pp. 1-10, 1995.

[4] H. Kang, H. Chen, and G. Jiang, "PeerWatch: a fault detection and diagnosis tool for virtualized consolidation systems," in *ACM Int. Conf. on Automatic Computing (ICAC)*, Washington, DC, USA, June 7-11, pp. 119-128, 2010.

[5] C. Wang, V. Talwar, K. Schwan, and P. Ranganathan, "Online detection of utility cloud anomalies using metric distributions", in *IEEE/IFIP Network Operations and Management Symposium (NOMS)*, Osaka, Japan, April 19-23, pp. 96-103, 2010.

[6] C. Wang, K. Viswanathan, L. Choudur, V. Talwar, W. Satterelid, and K. Schwan, "Statistical techniques for online anomaly detection in data centers," in *IFIP/IEEE Int. Symp. Integrated Network Management (IM)*, Dublin, Ireland, May 23-27, pp. 385-392, 2011.

[7] M. Jiang, M. A. Munawar, T. Reidemeister, and P. A. Ward, "Automatic fault detection and diagnosis in complex software systems by information-theoretic monitoring," in *IEEE Conf. Dependable Systems and Networks (DSN)*, Lisbon, Portugal, June 29 – July 2, pp. 285-294, 2009.

[8] K. Ozonat, "An information-theoretic approach to detecting performance anomalies and changes for large-scale distributed web services," in *IEEE Conf. Dependable Systems and Networks (DSN)*, Anchorage, Alaska, June 24-27, pp. 522-531, 2008.

[9] K. Viswanathan, L. Choudur, V. Talwar, C. Wang, G. MacDonald, and W. Satterelid, "Ranking anomalies in data centers," in *IEEE/IFIP Network Operations and Management Symposium (NOMS)*, Maui, HI, April 16-20, pp. 79-87, 2012.

[10] G. Jiang, H. Chen, K. Yoshihira, and A. Saxena, "Ranking the importance of alerts for problem determination in large computer systems," in *ACM Int. Conf. Automatic Computing (ICAC)*, Barcelona, Spain, June 15-19, pp. 3-12, 2009.

[11] Y. Tan and X. Gu, "On predictability of system anomalies in real world," in *IEEE Int. Symp. Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, Miami Beach, FL, Aug. 17-19, pp. 133-140, 2010.

[12] X. Gu and H. Wang, "Online anomaly prediction for robust cluster systems," in *IEEE Int. Conf. Data Engineering (ICDE)*, Shanghai, March 29-April 2, pp. 1000-1011, 2009.

[13] Y. Tan, X. Gu, and H. Wang, "Adaptive system anomaly prediction for large-scale hosting infrastructures," in *ACM SIGACT-SIGOPS Symp. Principles of Distributed Computing (PODC)*, Zurich, Switzerland, July 25-28, pp. 173-182, 2010.

[14] L. Cherkasova, K. M. Ozonat, N. Mi, J. Symons, and E. Smirni, "Anomaly? Application change? or workload change? Towards automated detection of application performance anomaly and change," in *IEEE Conf. Dependable Systems and Networks (DSN)*, Anchorage, AK, June 24-27, pp. 452-461, 2008.

[15] D. Dang, A. Lefaive, J. Scarpelli, and S. Sodem, "Automatic determination of dynamic threshold for accurate detection of abnormalities," US Patent 20110238376, Published 2011.

[16] J. C. Jubin, V. Rajasimman, N. Thadasina, "Hard handoff dynamic threshold determination", US Patent 20100056149, published 2010.

[17] Ying-Ru Chen, "Method of motion detection using adaptive threshold," US Patent 8077926 B2, 2011.

[18] H.M. Sun, S.P. Shieh, "A construction of dynamic threshold schemes," *Electronic Letters*, pp 2023-2025, NSC 84-2213-E-009-081, Nov. 1994.

[19] H.M. Sun, S.P. Shieh, "On dynamic threshold schemes," *Information Processing Letters* 52, pp 201-206, NSC 84-2213-E-009-081, 1994.

[20] VMware vCenter Operations Manager. http://www.vmware.com/products/vcenter-operations-manager.

[21] A.V. Poghosyan, A.N. Harutyunyan, N.M. Grigoryan, and M.A. Marvasti, "Data-agnostic anomaly detection," applied US patent 13/853,321, Filed March 29, 2013.

[22] H.B. Mann, "Nonparametric tests against trend", *Econometrica* 13, pp. 245–259, 1945.

[23] M.G. Kendall, *Rank Correlation Methods*. Griffin, London, UK, 1975.

[24] R.N. Bracewell, *The Fourier transform and its applications* (3rd ed.), Boston: McGraw-Hill, ISBN 0-07-116043-4, 2000.

[25] B. Boashash, ed., *Time-Frequency Signal Analysis and Processing: A Comprehensive Reference*, Oxford: Elsevier Science, ISBN 0-08-044335-4, 2003.

[26] A.V. Oppenheim, R.W. Schafer, and J.R. Buck, *Discrete-Time Signal Processing*, Upper Saddle River, N.J., Prentice Hall, ISBN 0-13-754920-2, 1999.

[27] P. Stoica and R. Moses, *Spectral Analysis of Signals*, Prentice Hall, NJ, 2005.

[28] S.M.G. Kendall, *Time Series*, Second Edition, Charles Griffin & Co., ISBN 0-85264-241-5, 1976.

[29] D.W. Tufts and R. Kumaresan, "Estimation of frequencies of multiple sinusoids: Making linear prediction perform like maximum likelihood," *Proc. IEEE*, vol. 70, pp. 975-989, 1982.

[30] R. de Prony, "Essai Experimentale et Analytique," *J. Ecole Polytechnique (Paris)*, pp. 24-76, 1795.

[31] M.A. Marvasti, A.V. Poghosyan, A.N. Harutyunyan, and N.M. Grigoryan, "An anomaly event correlation engine: Identifying root causes, bottlenecks, and black swans in IT environments", *VMware Technical Journal*, vol. 2, issue 1, pp. 35-45, 2013.

[32] M.A. Marvasti, A.V. Poghosyan, A.N. Harutyunyan, and N.M. Grigoryan, "Method and apparatus for root cause and crytical pattern prediction using virtual directed graphs", US Patent 20130097463, published 2013.

[33] A.N. Harutyunyan, A.V. Poghosyan, N.M. Grigoryan, and M.A. Marvasti, "Abnormality analysis of streamed log data", in *IEEE/IFIP Netwotk Operations and Management Symposium (NOMS)*, 5-9 May, Krakow, Poland, 2014.

[34] M.A. Marvasti, A.V. Poghosyan, A.N. Harutyunyan, and N.M. Grigoryan, "Methods and systems for for abnormality analysis of streamed log data", US Patent 20140053025, published 2014.

[35] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: a survey," *ACM Computing Surveys*, pp. 1-72, Sept. 2009.

[36] C. Wang, S.P. Kavulya, J. Tan, L. Hu, M. Kutare, M. Kasick, K. Schwan, P. Narasimhan, and R. Gandhi, "Performance troubleshooting in data centers: an annotated bibliography," *ACM SIGOPS Operating Systems Review*, vol. 47, issue 3, pp. 50-62, 2013.

User-Centric Heterogeneity-Aware MapReduce Job Provisioning in the Public Cloud

Eric Pettijohn*, Yanfei Guo*, Palden Lama† and Xiaobo Zhou*

*Department of Computer Science, University of Colorado, Colorado Springs, USA
†Department of Computer Science, University of Texas, San Antonio, USA
Emails: {epettij4, yguo, xzhou}@uccs.edu, palden.lama@utsa.edu

Abstract

Cloud datacenters are becoming increasingly heterogeneous with respect to the hardware on which virtual machine (VM) instances are hosted. As a result, ostensibly identical instances in the cloud show significant performance variability depending on the physical machines that host them. In our case study on Amazon's EC2 public cloud, we observe that the average execution time of Hadoop MapReduce jobs vary by up to 30% in spite of using identical VM instances for the Hadoop cluster. In this paper, we propose and develop U-CHAMPION, a user-centric middleware that automates job provisioning and configuration of the Hadoop MapReduce framework in a public cloud to improve job performance and reduce the cost of leasing VM instances. It addresses the unique challenges of hardware heterogeneity-aware job provisioning in the public cloud through a novel selective-instance-reacquisition technique. It applies a collaborative filtering technique based on UV Decomposition for online estimation of ad-hoc job execution time. We have implemented U-CHAMPION on Amazon EC2 and compared it with a representative automated MapReduce job provisioning system. Experimental results with the PUMA benchmarks show that U-CHAMPION improves MapReduce job performance and reduces the cost of leasing VM instances by as much as 21%.

1 Introduction

Today, big data processing frameworks such as Hadoop MapReduce [1] are increasingly deployed in public clouds. However, due to the absence of automation tools, currently end users are forced to make job provisioning decisions manually. Recent studies [16, 28, 29, 31] have focused on improving Hadoop job performance through automated resource allocation and parameter configuration. However, most research has been done on small private clusters, which tend to be homogeneous with respect to the hardware configuration and performance.

One of the foremost challenges of MapReduce job provisioning in a public cloud is imposed by the heterogeneity of the underlying hardware infrastructure [10,21, 27]. Cloud datacenters usually upgrade their hardware infrastructure over time, resulting in multiple generations of hardware with widely varying performance [25]. Such hardware heterogeneity has a significant impact on Hadoop job completion time [21]. However, the VM instances offered by public cloud providers do not indicate the performance implications of the heterogenous hardware that hosts them. Furthermore, there is no guarantee that one VM will always be provisioned on the same type of hardware. Our motivational case study on Amazon EC2 public cloud shows that the average execution time of Hadoop MapReduce jobs varies by up to 30% despite using identical VM instances for the Hadoop cluster. Hence, there is an urgent need for user-centric approaches that can address these challenges without requiring explicit control of the cloud environment.

In this paper, we present U-CHAMPION, a user-centric heterogeneity-aware middleware approach that automates Hadoop job provisioning and configuration in a public cloud to improve job performance and reduce the cost of leasing VM instances. However, there are several challenges in achieving heterogeneity-aware job provisioning in a public cloud.

It is challenging to develop accurate performance models for diverse Hadoop jobs running on a heterogenous cloud environment. Recent studies focused on intensive profiling of routinely executed jobs in the Hadoop environment in order to estimate their performance for various input data sizes [28]. However, such an approach is not feasible for ad-hoc jobs submitted to the system, which have unpredictable execution characteristics. To address this challenge, U-CHAMPION performs two-phase job profiling and performance modeling. In the offline phase, it applies support vector machine (SVM) regression modeling to estimate the completion time of various Hadoop jobs for different input data sizes, re-

source allocations, CPU models, and configuration parameters. In the online phase, it performs a lightweight profiling of ad-hoc jobs submitted to the system by using only a subset of possible configurations. Then, it applies the UV Decomposition technique to quickly estimate the job performance for all possible configurations.

U-CHAMPION's job performance models provide the foundation for making heterogeneity-aware resource allocation and configuration decisions for Hadoop jobs. However, it is significantly challenging to provision Hadoop jobs with the desired resource configurations in a public cloud. This is due to the fact that cloud providers do not allow the end-users to decide where their VM instances should be hosted. In addition, there are important cost/performance trade-offs inherent in cloud systems, which rent VM instances to users by the hour. U-CHAMPION addresses these challenges through a novel selective-instance-reacquisition technique. The main idea is to acquire new VM instances from the cloud whenever there is an expectation that doing so will result in more cost savings.

We have implemented U-CHAMPION on Amazon EC2 and evaluated its impact on Hadoop job performance and cost efficiency by using the PUMA benchmarks [3]. For comparison, we implemented AROMA [16], an automated MapReduce job provisioning system proposed recently. Experimental results demonstrate U-CHAMPION's improved accuracy in predicting ad-hoc Hadoop job performance. This is mainly due to its hardware heterogeneity awareness, and the effectiveness of the UV Decomposition approach. Furthermore, U-CHAMPION improves MapReduce job performance and reduces the cost of leasing VM instances by as much as 21%.

2 The Case Study and Motivations

Modern public clouds, such as Amazon EC2, routinely run large numbers of applications simultaneously on huge datacenters. In settings such as parallel data processing jobs, the MapReduce framework has become invaluable, allowing a relatively easy setup. However, the changing conditions within large datacenters have led to significant difficulties that are most visible in the public cloud.

2.1 Heterogeneity Characterization

Inside of a commercial datacenter, hardware is in constant flux. Servers are upgraded in sections, since fully upgrading a datacenter at once is prohibitively expensive. This has given rise to the current state, where several generations of hardware inhabit a single datacenter.

Table 1: Hardware heterogeneity in Amazon EC2.

CPU Type (Small VMs)	# of VMs	Percent of Total
US West-2 Datacenter		
E5-2650	101	82.79
E5645	21	17.21
US East Datacenter		
E5-2650	10	7.46
E5430	20	14.93
E5645	32	23.88
E5507	72	53.73

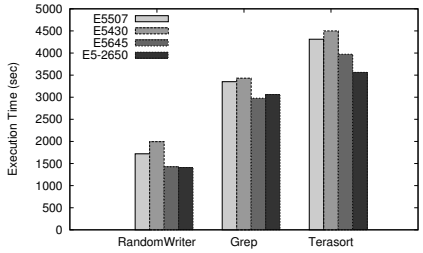


Figure 1: Impact of hardware heterogeneity on Hadoop job execution time.

We conducted a study on Amazon EC2 cloud datacenters in the US West and US East regions to measure the extent of their hardware heterogeneity. The US East datacenter in Virginia was created in 2006, while the US West-2 datacenter in Oregon started serving customers in 2011. We focused on analyzing CPU heterogeneity due to the evidence for it being the most significant source of performance variability [10, 21], and due to the ease and speed of determination (via the `cpuid` command).

The US East region shows a greater degree of hardware heterogeneity. This can be explained through the general observation that datacenters tend to grow more heterogeneous as they grow older, with newer servers being brought in to replace the older systems. Table 1 summarizes the results of our survey of the US West-2 and US East datacenters. The data was obtained by checking the CPU type of several hundred `m1.small` instances created on these datacenters on EC2. We report the number and the percentage of the VM instances running on various CPU types.

2.2 Impact of Hardware Heterogeneity

Next, we analyze the impact of Amazon EC2 hardware heterogeneity on the performance of Hadoop jobs. In this experiment, we ran three Hadoop benchmark programs (`RandomWriter`, `Grep` and `Terasort`) on Hadoop clusters of various CPU types. Each cluster consists of two VMs with the same CPU type. We ran each trial at least five times and reported the average completion times.

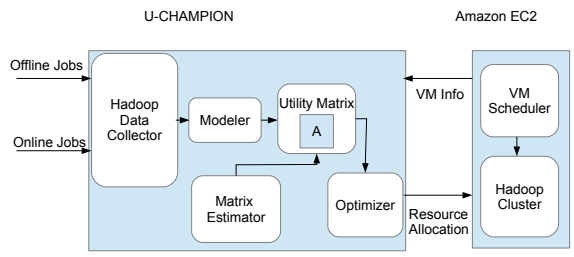


Figure 2: The system architecture of U-CHAMPION.

Figure 1 shows the job execution times of several benchmark applications running on clusters of two `m1.small` VMs. We observe that the average job execution times can vary by as much as 30% between a cluster running E5430 CPUs and one running E5-2650 CPUs. However, cloud users can not determine what CPU types will be associated with their VM instances. This motivates us to propose a user-centric heterogeneity-aware MapReduce job provisioning in the public cloud.

3 U-CHAMPION Design

3.1 Architecture

We aim to create an automated job provisioning system which integrates cluster setup optimization (cost awareness and instance selection) with heterogeneity-aware provisioning (parameter configuration, resource allocation) to improve Hadoop job performance and reduce the cost of leasing Cloud resources. The U-CHAMPION Architecture is shown in Figure 2. End users submit jobs to U-CHAMPION through a command-line interface, and our system provides appropriate configuration parameters, VM types, and underlying hardware types in order to minimize cost. With this information, a cluster is started on Amazon EC2 and the job is submitted to the master node of the cluster along with the results of the parameter optimization.

U-CHAMPION consists of three major components; the modeler, the estimator, and the optimizer.

3.2 Job Performance Modeling

In order to build accurate performance models of Hadoop jobs, we run various Hadoop benchmarks on several clusters on Amazon EC2 off line. We mine the Hadoop logs of executed jobs for execution time, input size, and various Hadoop configuration parameters. We use the *cpuid* package to obtain the node CPU configuration. This enables the creation of a database from which we can create our own performance models.

3.2.1 Support Vector Machine Models

Our system applies a powerful supervised machine learning technique to learn the performance model for each job. It constructs a support vector machine (SVM) regression model to estimate the completion time of jobs for different input data sizes, resource allocations, CPU models, and configuration parameters. SVM methodology is known to be robust for estimating real-valued functions (regression problem) from noisy and sparse training data having many attributes [7, 26]. This property of SVM makes it a suitable technique for performance modeling of complex Hadoop jobs in the Cloud environment.

We conduct stepwise regression on the data sets collected from our test-bed of virtualized Amazon EC2 Instances. For data collection, we measured the execution times of various Hadoop jobs with different input data sizes in the range of 1 GB to 50 GB, using various Hadoop parameter configurations and running on different cluster sizes of Hadoop nodes comprising of `m1.small` instances on Amazon EC2. Due to the inherent cost of running instances on EC2, we limit ourselves to `m1.small` instances in order to stretch our resources further.

U-CHAMPION incorporates hardware heterogeneity by mining data clusters for CPU type during the regression modeling. As shown in our case study, differences in CPU cause large differences in performance between seemingly identical `m1.small` instances in Amazon EC2. U-CHAMPION accounts for these differences, thereby directly increasing estimation accuracy.

3.3 Online Matrix Estimation

For ad-hoc jobs, U-CHAMPION performs a lightweight online profiling on a small portion of the input dataset with various Hadoop configuration parameters. This profiling is performed on two different CPU configurations in parallel to provide a seed of heterogeneity information. U-CHAMPION relies on online matrix estimation to obtain the complete performance model with heterogeneity information.

We apply UV Decomposition [23], a collaborative filtering technique used for matrix estimation for extremely sparse data, and which was used in the Netflix Challenge [2]. We apply it here to a similar problem, where we need to estimate the response of a job to new configuration and cluster conditions in terms of execution time by estimating based on previously collected data. UV Reconstruction has been shown to be effective for matrices where less than 5% of values are known [23].

3.4 Cost Optimization

U-CHAMPION improves the job execution time by searching for the optimal configuration and underlying hardware. It tries to provision the Hadoop cluster on the CPU type that leads to the best performance. The user has no control of VM placement in the public cloud. Thus, the optimization in U-CHAMPION has to consider both the cost of job execution and the cost of acquiring the desired CPU type.

3.4.1 Cost Estimation

The costs associated with Amazon EC2 instances are well-known and published on their website. Users are charged by the hour, and are charged for a full-hour for each partial hour used. Therefore, the cost of running a job in EC2 can be estimated as

$$c_{job} = t_{job} \cdot n_{inst} \cdot c_{inst}$$

where t_{job} is the number of hours a job takes to complete with the current cluster (rounded up to the nearest integer due to the discrete charging intervals on EC2), n_{inst} represents the number of instances, and c_{job} is the total cost of one job execution. Note that neither n_{inst} nor c_{inst} changes as a result of CPU type, so that the estimated execution time of the job is the only variable which can be optimized for a cluster of a specific size and instance type (i.e. small, medium or large standard instances).

The cost of acquiring a VM that is provisioned on a specific CPU type can be estimated by

$$e_{cpu} = \frac{c_{inst}}{p_{cpu}} \quad (1)$$

where e_{cpu} is the expected cost of obtaining a VM with given CPU type, c_{inst} is the cost of the VM instance per hour, and p_{cpu} is the probability of obtaining an instance with that CPU type from Amazon (Here we use the data provided in Table 1). We represent e_{cpu} as the expected number of VMs needed to find a certain CPU type (which is simply $1/p_{cpu}$) multiplied by the cost/hour, since Amazon charges one hour of cost upon requesting a VM.

3.4.2 Cost Saving by VM Reacquisition

Here we provide our logic for an algorithm which provides cluster optimization through selective instance reacquisition. We acquire new instances wherever we have an expectation that doing so will result in more cost savings through predicted execution time improvement than cost overhead involved in requesting additional instances and closing under-performing ones.

This leads to the examination of our tradeoff for each VM, which is

$$t_{job} \cdot c_{inst} \geq \frac{t_{job} \cdot c_{inst}}{\alpha} + e_{cpu}$$

where α is the speedup from changing CPU type. Here we state that if the total estimated cost of a VM is greater than the estimated cost of the VM with a new CPU plus the estimated cost of obtaining that CPU, then it is advantageous for us to look for higher-performing instances. The speedup α is obtained through previous results for jobs run on the various CPU types. By performing this examination on all VM instances, we are able to optimize the cost saving for the Hadoop cluster.

4 Implementation and Evaluation

4.1 Testbed

We build our testbed using Amazon EC2 service. We use the US East datacenter due to the large amount of observed hardware heterogeneity. The datacenter has four different CPU types: Intel Xeon E5-2650, Intel Xeon E5645, Intel Xeon E5507, and Intel Xeon E5430. We provisioned multiple `m1.small` virtual machine instances. Each of them have one vCPU and 1.7 GB memory. The VMs are created using the standard Amazon Machine Image (AMI) provided by *alestic.com* and installed with Ubuntu Linux 10.04.

We build Hadoop clusters using Hadoop version 1.1.2, and provision with sizes ranging from 2 to 10 slave nodes for the experiments. Each slave node is configured with one map slot and one reduce slot.

We use the PUMA benchmark suite [3] to test the performance of U-CHAMPION with representative MapReduce jobs. The PUMA benchmark contains various MapReduce benchmarks and real-world test inputs. In the experiments, we performed offline profiling on `Grep`, `Wordcount`, `Inverted Index`, and `RandomWriter` benchmarks, then performed online profiling and model estimation for the `Terasort` benchmark.

For comparison, we implemented AROMA [16]. AROMA is an automated configuration system for Hadoop parameters using machine learning to profile jobs and clustering of profiles to optimize job execution time and cost. It is hardware heterogeneity agnostic.

The output of our job models is the job execution time for a set of inputs. We used the LIBSVM library [7] to explore appropriate kernel functions and implement the SVM regression technique.

4.2 Execution Time Estimation Accuracy

First, we study the accuracy of job execution time estimation. We create a Hadoop cluster with two slave nodes on Amazon EC2. We use the `Terasort` benchmark with 20 GB input data that is generated by `RandomWriter`. We create job performance models for U-CHAMPION

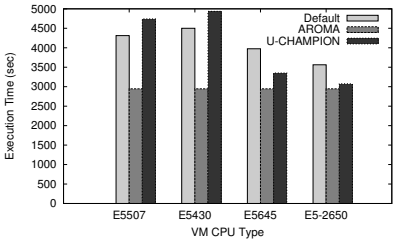


Figure 3: Estimated vs actual execution times.

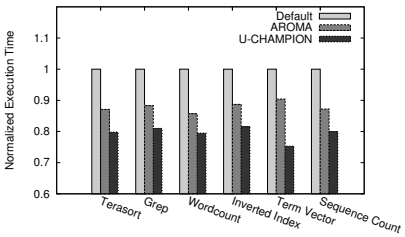


Figure 4: PUMA benchmark job execution time.

and AROMA and use them to estimate the job execution time of `Terasort` on various CPU types. We compare the estimated execution time by U-CHAMPION and AROMA to the actual execution time of the job.

Figure 3 shows the estimated job execution time of the `Terasort` benchmark on different hardware. The estimated job execution time by U-CHAMPION is 9.9%, 9.7%, 15.7%, and 13.7% different from the actual value on the E5507, E5430, E5645, and E5-2650, respectively. U-CHAMPION is able to accurately estimate execution time for different CPU types. AROMA is profiled using E5-2650 CPUs, and it provides the same estimated job execution time for different CPU types. As a result, the worst-case estimation error for U-CHAMPION is less than 16%, whereas AROMA's worst case estimation error is 35%.

4.3 Improving Job Execution Time

In this section, we evaluate U-CHAMPION's ability to reduce the overall job execution time. We build a Hadoop cluster with two slave nodes and run several PUMA benchmarks with 20 GB of input data. We use the job execution time of Hadoop with a default configuration as the baseline and compare the normalized job execution time of U-CHAMPION and AROMA. The default configuration uses heterogeneity-blind resource provisioning, meaning that we simply use whichever instances are assigned to us by Amazon.

Figure 4 shows the normalized job execution time of all benchmarks using these three approaches. The results show that U-CHAMPION outperformed the default configuration, with up to 21% shorter job execution times. U-CHAMPION provides the optimized configuration and cluster for each job, leading to this significant improvement in job execution time. In the public clouds, users are charged for VM instances by the hour. Thus, a reduction in job execution time directly results in cost savings. U-CHAMPION also outperformed AROMA, achieving up to 20% job execution time savings. U-CHAMPION achieves better performance than AROMA due to its ability to exploit the heterogeneity in the underlying hardware. U-CHAMPION not only provides an optimized job configuration, but also provisions better VM instances through selective instance reacquisition.

Table 2: Cost Optimization for `RandomWriter` Benchmark Workload.

Before Optimization	After Optimization
E5-2650 × 1	E5-2650 × 3
E5430 × 2	
E5645 × 2	E5645 × 7
E5507 × 5	
Execution time improvement	14.5%
Execution cost improvement	14.5%
Cost overhead	$1.68

4.4 Cluster Optimization vs Cost

U-CHAMPION estimates the cost of finding better-performing VM instances for a cluster, and compares it with cost-savings achieved via lower execution time due to said instances. The cost of `m1.small` VM instance is $C_{inst} = \$0.06/h$ and approximately 7.46% of our instances will be E5-2650 instances to start (see Table 1). The approximate cost of finding the best-performing CPU type (E5-2650) for `RandomWriter` is $\frac{0.06}{0.0746} = 0.804$ dollars (Eq. 1). As the execution time of our `RandomWriter` task approaches infinity, we can obtain an average of 13% and a maximum of 30% (ie, we started with only the worst-performing VM instances) cost savings by using only the best-performing VM instances.

Table 2 shows example results of our cost algorithm for a Hadoop cluster of 10 slave nodes running the *RandomWriter* with 40 GB input data. The underlying CPUs of the node before and after the algorithm is run are shown. U-CHAMPION creates 28 new VM instances to obtain the VMs with desired CPU type, therefore the cost overhead of this cluster performance enhancement is $\$1.68$ $(28 * C_{inst})$. Keep in mind that this provides execution time and cost improvement as long as the cluster is running. The cost will be amortized across all jobs run on this cluster until it is shut down by the user. For this example, we assume that the instances opened follow

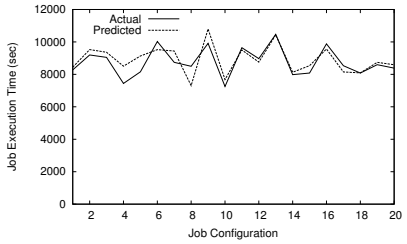


Figure 5: Prediction accuracy for an ad-hoc job for different Hadoop configurations.

the distribution which we observed in Table 1 in order to show expected results.

4.5 Adaptiveness to Ad-Hoc Jobs

Previously we showed that ad-hoc jobs submitted to the U-CHAMPION system were predicted with reasonable accuracy even in the absence of similar jobs. Here we evaluate the performance under more reasonable conditions. And, we show that the model generated for the `Terasort` benchmark by UV Decomposition remains useful under a variety of job configurations.

This experiment assumes that a `Terasort` workload with 20GB of input data (generated by TeraGen) has been submitted to the system. We also assume an 8 VM cluster, and that several other benchmarks (`Wordcount`, `Grep`, `Inverted Index`, etc) have been profiled offline. In this case, we use the methodology described in section 3.3 to create a model for `Terasort`. Figure 5 shows the prediction accuracy for 20 different Hadoop configurations using the new model. We see prediction error here of less than 17%.

5 Related Work

Recent studies have focused on improving the performance of applications in clouds [5, 6, 8, 12, 14, 24] through elastic resource allocation and VM scheduling. Paragon [8] implements a heterogeneity-aware job scheduling system using a Singular Value Decomposition (SVD) technique similar to U-CHAMPION, but considers scheduling only single-node applications and requires full control of the cloud environment, making it very hard to use for a user in the public cloud.

Users of the public cloud only have limited information about the cloud environment and have no control of the hardware their VMs run on. U-CHAMPION, along with other user-centric research [16, 19], uses the limited information that is available in order to improve the decisions made by *one* user.

Hardware heterogeneity is a prevalent issue in public clouds [10, 17, 21]. Recent studies show that it is feasible to leverage the hardware heterogeneity to improve the performance of applications [9–11, 20, 21].

There are also some works focusing on improving Hadoop performance by reducing the delay due to shuffle and straggler tasks [13, 18, 30]. Park *et al.* proposed a novel VM reconfiguration approach that is aware of the data locality of Hadoop [22]. Guo *et al.* proposed and implemented iShuffle [13], a user-transparent shuffle service that pro-actively pushes map output data to nodes via a novel shuffle-onwrite operation and flexibly schedules reduce tasks considering workload balance.

There is a rich set of research focused on the parameters and performance of Hadoop clusters. Jiang *et al.* [15], conducted a comprehensive performance study of Hadoop and summarized the factors that can significantly improve Hadoop performance. Verma *et al.* [28, 29], proposed a cluster resource allocation approach for Hadoop. AROMA [16] provides a novel framework for automated parameter estimation and cluster resource provisioning in order to maximize job performance in a given cluster.

6 Conclusion

U-CHAMPION is proposed and developed to enable a user-centric and heterogeneity-aware MapReduce job provisioning in the public cloud. It addresses the unique challenges imposed by the public cloud environment through a novel selective-instance-reacquisition technique. This technique applies our proposed optimization algorithm to acquire new VM instances if it results in more cost savings. Furthermore, U-CHAMPION is able to make accurate performance prediction of ad-hoc Hadoop jobs through its UV Decomposition technique and by the incorporation of hardware heterogeneity-awareness in job performance modeling. Extensive evaluation of U-CHAMPION on Amazon EC2 Cloud with representative benchmark applications demonstrated its improved performance prediction accuracy as compared to a heterogeneity-unaware approach. Furthermore, the results showed its ability to improve Hadoop job performance and reduce the cost of leasing the Cloud resources by up to 21%.

Our future work will extend U-CHAMPION to a multi-user environment for mitigating performance interference.

Acknowledgement

This research was supported in part by U.S. NSF CAREER award CNS-0844983, research grants CNS-1320122 and CNS-1217979.

References

[1] Apache Hadoop Project. `http://hadoop.apache.org`.

[2] Netflix Prize. `http://www.netflixprize.com`.

[3] PUMA: Purdue mapreduce benchmark suite. `http://web.ics.purdue.edu/~fahmad/benchmarks.htm`.

[4] AHMAD, F., CHAKRADHAR, S. T., RAGHUNATHAN, A., AND VIJAYKUMAR, T. N. Tarazu: Optimizing mapreduce on heterogeneous clusters. In *Proc. of the ACM Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2012).

[5] BEN-YEHUDA, O. A., BEN-YEHUDA, M., SCHUSTER, A., AND TSAFRIR, D. The resource-as-a-service (raas) cloud. In *Proc. of the USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)* (2012).

[6] BU, X., RAO, J., AND XU, C. Interference and locality-aware task scheduling for mapreduce applications in virtual clusters. In *Proc. of the ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC)* (2013).

[7] CHANG, C.-C., AND LIN, C.-J. LIBSVM: A library for support vector machines. *ACM Trans. Intelligent System Technology* (2011).

[8] DELIMITROU, C., AND KOZYRAKIS, C. Paragon: Qos-aware scheduling for heterogeneous datacenters. In *Proc. of the ACM Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2013).

[9] DELIMITROU, C., AND KOZYRAKIS, C. Quasar: Resource-efficient and qos-aware cluster management. In *Proc. of the ACM Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2014).

[10] FARLEY, B., JUELS, A., VARADARAJAN, V., RISTENPART, T., BOWERS, K. D., AND SWIFT, M. M. More for your money: exploiting performance heterogeneity in public clouds. In *Proc. of the ACM Symposium on Cloud Computing (SoCC)* (2012).

[11] GUEVARA, M., LUBIN, B., AND LEE, B. C. Market mechanisms for managing datacenters with heterogeneous microarchitectures. *ACM Trans. on Computer Systems 32*, 1 (Feb. 2014), 3:1–3:31.

[12] GUO, Y., LAMA, P., RAO, J., AND ZHOU, X. V-cache: Towards flexible resource provisioning for clustered applications in iaas clouds. In *Proc. IEEE Int'l Parallel and Distributed Processing Symposium (IPDPS)* (2013).

[13] GUO, Y., RAO, J., AND ZHOU, X. iShuffle: Improving hadoop performance with shuffle-on-write. In *Proc. of the USENIX Int'l Conference on Autonomic Computing (ICAC)* (2013).

[14] JALAPARTI, V., BALLANI, H., COSTA, P., KARAGIANNIS, T., AND ROWSTRON, A. Bridging the tenant-provider gap in cloud services. In *Proc. of the ACM Symposium on Cloud Computing (SoCC)* (2012).

[15] JIANG, D., OOI, B. C., SHI, L., AND WU, S. The performance of MapReduce: an in-depth study. *Proc. VLDB Endowment* (2010).

[16] LAMA, P., AND ZHOU, X. AROMA: automated resource allocation and configuration of mapreduce environment in the cloud. In *Proc. of the ACM Int'l Conference on Autonomic computing (ICAC)* (2012).

[17] LEE, G., CHUN, B., AND RANDY, H. Heterogeneity-aware resource allocation and scheduling in the cloud. In *Proc. of the USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)* (2011).

[18] LI, X., WANG, Y., JIAO, Y., XU, C., AND YU, W. CooMR: Cross-task coordination for efficient data management in mapreduce programs. In *Proc. of the Int'l Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (2013).

[19] LIM, S.-H., HUH, J.-S., KIM, Y., SHIPMAN, G. M., AND DAS, C. R. D-factor: a quantitative model of application slow-down in multi-resource shared systems. In *Proc. ACM SIGMETRICS* (2012).

[20] MARS, J., AND TANG, L. Whare-map: Heterogeneity in "homogeneous" warehouse-scale computers. In *Proc. of the ACM Int'l Symposium on Computer Architecture (ISCA)* (2013).

[21] OU, Z., ZHUANG, H., NURMINEN, J. K., YLÄ-JÄÄSKI, A., AND HUI, P. Exploiting hardware heterogeneity within the same instance type of amazon ec2. In *Proc. of the USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)* (2012).

[22] PARK, J., LEE, D., KIM, B., HUH, J., AND MAENG, S. Locality-aware dynmaic vm reconfiguration on mapreduce clouds. In *Proc. of the ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC)* (2012).

[23] RAJARAMAN, A., AND ULLMAN, J. *Textbook on Mining of Massive Datasets*. 2011.

[24] RAO, J., WANG, K., ZHOU, X., AND XU, C. Optimizing virtual machine performance in NUMA multicore systems. In *Proc. of the IEEE Int'l Symposium on High Performance Computer Architecture (HPCA)* (2013).

[25] REISS, C., TUMANOV, A., GANGER, G. R., KATZ, R. H., AND KOZUCH, M. A. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proc. of the ACM Symposium on Cloud Computing (SoCC)* (2012).

[26] SCHÖLKOPF, B., SMOLA, A. J., WILLIAMSON, R. C., AND BARTLETT, P. L. New support vector algorithms. *Neural Comput.* (2000).

[27] SCHWARZKOPF, M., MURRAY, D. G., AND HAND, S. The seven deadly sins of cloud computing research. In *Proc. of the USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)* (2012).

[28] VERMA, A., CHERKASOVA, L., AND CAMPBELL, R. H. ARIA: automatic resource inference and allocation for mapreduce environments. In *Proc. of the ACM Int'l Conference on Autonomic Computing (ICAC)* (2011).

[29] VERMA, A., CHERKASOVA, L., AND CAMPBELL, R. H. Resource provisioning framework for mapreduce jobs with performance goals. In *Proc. of the ACM/IFIP/USENIX Int'l Conference on Middleware* (2011).

[30] WANG, Y., QUE, X., YU, W., GOLDENBERG, D., AND SEHGAL, D. Hadoop acceleration through network levitated merge. In *Proc. of the Int'l Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (2011).

[31] WOLF, J., RAJAN, D., HILDRUM, K., KHANDEKAR, R., KUMAR, V., PAREKH, S., WU, K., AND BALMIN, A. Flex: a slot allocation scheduling optimizer for mapreduce workloads. In *Proc. of the ACM/IFIP/USENIX Int'l Conference on Middleware* (2010).

[32] ZAHARIA, M., KONWINSKI, A., JOSEPH, A. D., KATZ, R., AND STOICA, I. Improving mapreduce performance in heterogeneous environments. In *Proc. of the USENIX Conference on Operating Systems Design and Implementation (OSDI)* (2008).

Exploiting Temporal Diversity of Water Efficiency to Make Data Center Less "Thirsty"

Mohammad A. Islam, Kishwar Ahmed, Shaolei Ren, Gang Quan
Florida International University

Abstract

Data centers, which include both cyber (e.g., servers) and physical (e.g., cooling units) assets, are notorious for their energy consumption and carbon footprint. Nonetheless, a less-known fact about data centers is that they are extremely "thirsty" (for cooling), consuming millions of gallons water each day and raising serious concerns amid extended droughts. To curtail the surging water footprint, we adopt a holistic cyber-physical approach and incorporate the inherent physical characteristic of data center — time-varying water efficiency — into server provisioning and workload management. Specifically, we propose an online batch job scheduling algorithm, called WACE (minimization of WAter, Carbon and Electricity cost), which dynamically adjusts server provisioning to reduce the water consumption by deferring delay-tolerant batch jobs to water-efficient time periods. We demonstrate the effectiveness of WACE via trace-based simulations, showing that WACE reduces 27% water consumption compared to state-of-the-art scheduling algorithms.

1 Introduction

Ubiquitous Internet services and explosive IT demand have led to a new wave of constructing gigantic data centers, accounting for 1.7-2.2% of the total electricity usage in the United States as of 2010 [13]. Data centers consist of both cyber assets (e.g., servers, networking equipment) and physical assets (e.g., cooling systems, energy storage device). While data centers are notorious for huge energy consumption due to power-hungry servers, cooling systems — data centers' physical assets — are very "thirsty", evaporating millions of gallons of water each day for rejecting server heat. For example, cooling towers in AT&T's large data center facilities consume 1 billion gallons of water in 2012, approximately 30% of the entire company's water consumption [3]. In addition, just as they are accountable for carbon emissions via electricity usage, data centers also consume a vast amount of offsite water *remotely* embedded in electricity generation: e.g., in the U.S., an average of 1.8 liter of water is evaporated, or "lost", into the air for just one kilowatt-hour electricity generation, even excluding the water-consuming hydropower [29, 32]. A typical water-cooled data center is illustrated in Fig. 1.

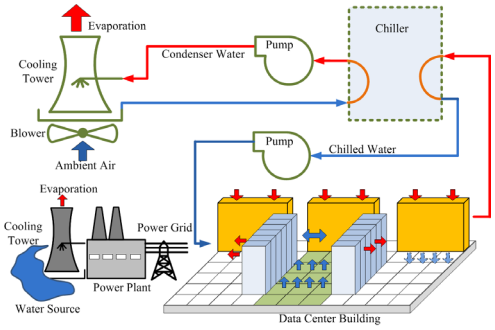


Figure 1: Water consumption in data center.

Why is water critical to data centers? As shown in a recent survey by Uptime Institute [34], cooling towers are widely employed by large data centers (over 40%), even though there exist various types of cooling systems and data centers in low-temperature regions may use cold outside air for cooling. Water conservation has been become essential for green certifications (being sought by a majority of large data centers [34]), tax credits [36], and corporate social responsibility [3]. On the other hand, water is also crucial for electricity generation (e.g., thermoelectricity, nuclear power) [29, 32], which is undeniably essential for data center operation.

Water is not equal to energy. Despite the nexus between water and energy [29, 32], the existing studies for minimizing data center energy consumption cannot minimize water footprint, because when optimizing for energy efficiency, they neglect the physical characteristic (in particular, *time-varying* water efficiency, details provided in Section 3) of data center cooling systems and electricity generation. In fact, for reducing water footprint, it is

not only important to minimize energy, but also crucial to consider *"when"* to consume energy.

Prior studies on minimizing electricity cost [26] and carbon emissions [9] do *not* lead to water minimization solutions either, because water efficiency is not in proportion to electricity cost-/carbon-efficiency (e.g., nuclear power incurs little carbon emission but can consume more than 2L of water per kWh [20, 23]). While using air economizer (i.e., "free" cooling) and recycled/sea water can reduce potable water consumption [10, 21], these techniques, however, focus on improved "engineering" and do not apply to all data centers, because they typically require high upfront costs and/or suitable locations/climate conditions.

Software-based approach to water conservation. We incorporate water footprint as an integral metric into data center operation. For water conservation, the high-level intuition is to exploit the inherent physical characteristic of data center onsite cooling towers and offsite electricity generation (i.e., temporal diversity of water efficiency): we would like to defer batch workloads to time periods with better water efficiency while shutting down some servers during time periods with low water efficiency. To turn this intuition into reality, a notable challenge is that it is difficult to determine which time periods are water-efficient without foreseeing far future information, due to the time-varying nature of water efficiency, job arrivals, carbon emission rate and electricity price. A straw man technique can be setting a threshold on water efficiency: process batch jobs only when data center water efficiency is better than the threshold. Nonetheless, setting a too high threshold may degrade the delay performance too much, whereas setting a too low threshold may unnecessarily waste water.

To address the challenge, we propose a new online delay-tolerant batch workload management algorithm, called WACE (minimization of WAter, Carbon and Electricity cost), to reduce water footprint, while also including electricity cost and carbon footprint as an integral part of the optimization objective. A remarkable feature of WACE is that it can be implemented online based on the currently available information, yet we demonstrate its effectiveness through a traced-based simulation. Our simulation results show that, compared with state-of-the-art scheduling algorithms, WACE can reduce the cost by approximately 20%, while reducing the water consumption by approximately 27%.

2 Model

We consider a discrete-time model by equally dividing the entire time horizon of interest (e.g., one year) into K time slots. The duration of each time slot may range from minutes up to an hour. We focus on facility-level server provisioning and workload management. Next, we provide modeling details, which are consistent with the literature (e.g., [16, 38]).

2.1 Workload

In general, there are two types of workloads in data centers: delay-tolerant batch workloads (e.g., back-end processing, scientific applications) and delay-sensitive interactive workloads (e.g., web services or business transactional applications). We focus on scheduling batch workloads and denote by $a(t) = [0, a_{max}]$ the amount of batch workload arrivals at time t, quantified in terms of machine-time [17,35]. Although this widely-used model cannot capture all the low-level details (e.g., parallelism), it provides a *good* guidance for dynamically "sizing" the data center (i.e., how many servers can be turned off) and hence suffices for our purpose.

2.2 Data center

The data center has $r(t)$ amount of on-site renewable energy, e.g., by solar panels [1]. There are a total of $M(t)$ homogeneous servers that are available for processing batch jobs at time t. Servers may run at different processing speeds and incur different power [18]: we consider an array of finite processing speeds denoted by $\mathscr{S} = \{s_1, \cdots, s_N\}$, from which a speed s is chosen for processing batch workloads. Following [8, 18], we express the average power consumption of a server at time t as $\alpha \cdot s(t)^n + p_0$, where α is a positive factor and relates the processing speed to the power consumption, n is empirically determined (e.g., between $1 \sim 3$), and p_0 represents the power consumption in idle or static state. We model the server energy consumption by interactive workloads as an exogenously-determined value $p_{int}(t)$. We write the energy consumption by batch workloads as $p_{bat}(t) = m(t) \cdot [\alpha \cdot s(t)^n + p_0]$. Hence, the total server energy consumption can be formulated as

$$p(t) = p_{bat}(t) + p_{int}(t). \tag{1}$$

Next, given the available on-site renewable energy $r(t)$, the data center's electricity usage at time t is $[\gamma(t)p(t) - r(t)]^+$, where $[\,\cdot\,]^+ = \max\{\cdot, 0\}$ and $\gamma(t)$ is the factor of Power Usage Effectiveness (PUE) capturing the non-IT energy consumption.

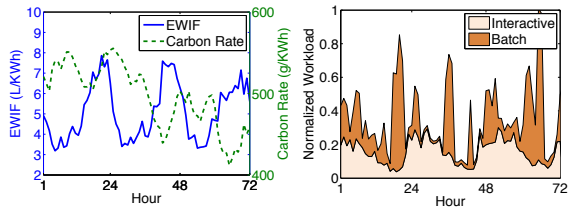
3 Online Batch Job Scheduling: WACE

In this section, we formulate the cost, present problem formulation and develop an online algorithm, WACE, to minimize the total cost via online batch job scheduling.

3.1 Cost

Our work aims to address three "costs": water consumption, electricity cost and carbon emission.

• **Water consumption.** As illustrated in Fig. 1, water is consumed in data center's onsite physical asset (i.e.,



(a) (b)

Figure 2: Water-carbon efficiency and workload trace. **(a)** EWIF and carbon emission rate in California. **(b)** Workload trace [12, 16].

cooling tower) and also offsite power plants. To assess water usage efficiency, an emerging metric, called Water Usage Effectiveness (WUE), was recently developed by The Green Grid [32]. WUE is the ratio of water consumption to IT equipment energy, where water consumption includes both direct and indirect water consumption (i.e., onsite water for data center cooling and offsite water for electricity production).

Direct water: Cooling towers directly consume onsite fresh water, and direct WUE at time t, denoted by $\varepsilon_D(t)$, is affected by various factors, such as non-stationary outside wet bulb temperature [31]. Hence, as an inherent characteristic of cooling tower, direct WUE exhibits a time-varying nature. In practice, direct WUE can be monitored in real-time [7].

Indirect water: Indirect water efficiency is quantified in terms of Energy Water Intensity Factor (EWIF), which measures the amount of water consumption per kWh electricity. Different energy fuel types (e.g., thermal, nuclear, hydro) have different EWIFs [19]. As the energy fuel mixes vary over time due to varying peak/non-peak demand [9], the resulting average EWIF exhibits a temporal diversity. Fig. 2(a) demonstrates the time-varying EWIF for California, calculated based on [19] and California energy fuel mixes [2]. In our study, we calculate the average EWIF as follows:

$$\varepsilon_I(t) = \frac{\sum_k b_k(t) \times \varepsilon_k}{\sum_k b_k(t)} \tag{2}$$

where $b_k(t)$ denotes the amount of electricity generated from fuel type k, and ε_k is the EWIF for fuel type k.

Now, we formulate the water consumption at time t as

$$w(t) = \varepsilon_D(t) \cdot p(t) + \varepsilon_I(t) \cdot [\gamma(t) \cdot p(t) - r(t)]^+, \tag{3}$$

where $p(t)$ is the server power, $\gamma(t)$ is PUE and $r(t)$ is available on-site renewable energy.

• **Electricity cost.** We denote the electricity price at time t by $u(t)$, and hence the electricity cost is $e(t) = u(t) \cdot [\gamma(t)p(t) - r(t)]^+$, where $[\gamma(t)p(t) - r(t)]^+$ is the data center electricity usage. Note that with the emergence of smart grid, large data centers may have the market power to influence real-time electricity price, and if so, the electricity price $u(t)$ can be modeled following [37, 39].

• **Carbon emission.** We calculate the (average) carbon emission rate following [9], and Fig. 2(a) demonstrates the time-varying carbon emission rate for California. An interesting observation is that carbon emission efficiency does not align with EWIF (i.e., indirect water efficiency). The difference between carbon efficiency and water efficiency becomes even greater if we factor in the time-varying direct WUE for cooling towers. The same observation also holds for electricity cost efficiency versus water efficiency. Next, we express the total carbon footprint of the data center at time t as $c(t) = \phi(t) \cdot [\gamma \cdot p(t) - r(t)]^+$, where we neglect the small carbon emission by the on-site renewable energy.

3.2 Problem formulation

In this subsection, we first describe our objective and constraints, and then present the problem formulation.

Objective. We aim to minimize the electricity cost while incorporating carbon emission and water consumption. We construct a parameterized *total cost* function as follows

$$g(t) = e(t) + h_w \cdot w(t) + h_c \cdot c(t), \tag{4}$$

where $h_w \geq 0$ and $h_c \geq 0$ are weighting parameters for water consumption and carbon emission relative to the electricity cost. Such a multi-objective formulation is common in the literature Our optimization objective is to minimize the long-term average cost expressed as $\bar{g} = \frac{1}{K} \sum_{t=0}^{K-1} g(t)$, where K is the total number of time slots in the period of interest.

Constraints. First, the number of servers to process batch jobs needs to satisfy

$$0 \leq m(t) \leq M(t), \tag{5}$$

where $M(t)$ is the maximum available number of servers. The server can only select one of the supported speeds:

$$s(t) \in \mathscr{S} = \{s_0, s_1, \cdots, s_N\}. \tag{6}$$

We also need to guarantee that batch jobs will be processed (without dropping):

$$\bar{a} < \bar{b}, \tag{7}$$

$$b(t) = m(t) \cdot s(t), \tag{8}$$

where $\bar{a} = \sum_{t=0}^{K-1} a(t)$ and $\bar{b} = \sum_{t=1}^{K-1} b(t)$ are the long-term average workload arrival and allocated server capacity, respectively. The constraint (8) states the relation between processed batch jobs and server provisioning.

Algorithm 1 WACE

1: At the beginning of each time t, observe the data center state information $r(t)$, $\varepsilon_D(t)$, $\varepsilon_I(t)$, $\phi(t)$ and $p_B(t)$, for $t = 0, 1, 2, \cdots, K-1$

2: Choose $s(t)$ and $m(t)$ subject to (5)(6)(8) to minimize

$$V \cdot g(t) - q(t) \cdot b(t) \tag{12}$$

3: Update $q(t)$ according to (11).

Problem formulation. We present an offline problem formulation for batch job scheduling as follows

$$\mathbf{P1}: \quad \min_{\mathscr{D}} \bar{g} = \frac{1}{K} \sum_{t=0}^{K-1} g(s(t), m(t)) \tag{9}$$

$$s.t., \quad \text{constraints } (5), (6), (7), (8). \tag{10}$$

Clearly, finding the optimal offline solution to **P1** requires complete offline information (i.e., workload arrivals, direct WUE, EWIF, carbon emission rate, on-site renewables and electricity prices) throughout the entire time period, which is very challenging, if not impossible, to obtain in practice. Therefore, we develop an online algorithm below.

3.3 WACE

To enable an online algorithm, we remove (7) and maintain a batch job queue that stores unfinished batch jobs. Specifically, assuming that $q(0) = 0$, we write the job queue dynamics as

$$q(t+1) = [q(t) - b(t)]^+ + a(t), \tag{11}$$

where $[\cdot]^+ = \max\{\cdot, 0\}$, $a(t)$ quantifies batch job arrivals, and $b(t)$ indicates the amount of processed jobs.

Intuitively, when the queue length becomes large, the data center should increase the number of servers and/or server speed to reduce the queue backlog to avoid too much delay. Hence, we incorporate the queue length into the objective function, as described in Algorithm 1. In (12), the queue length determines how much emphasis the optimization gives on the resource provisioning $b(t)$ for processing batch jobs. WACE is purely online and only requires the currently available information. The parameter $V \geq 0$ in line 2 of Algorithm 1, referred to cost-delay parameter, acts as a tradeoff control knob: the larger V, the smaller impact of the queue length on optimization decisions.

While Algorithm 1 appears simple, it is provably-efficient, even compared to the optimal offline algorithm that has future information. In particular, one can show based on the recently-developed Lyapunov technique [22] that the gap between the average cost achieved by WACE and that by the optimal offline algorithm is bounded, while the batch job queue length is also upper bounded, translating into a finite queueing delay. We omit the proof details due to space limitations.

4 Performance Evaluation

This section presents trace-based simulation studies of a data center to evaluate WACE.

4.1 Data Sets

We consider a large data center consisting of 300,000 homogeneous servers with a peak power of 64MW. Each server has 15 discrete speed levels, uniformly ranging from 1.6GHz to 3GHz. The duration of each time slot is set to 1 hour and the total simulation period is 1 year. The default weighting parameters for water consumption and carbon emission are $h_w = 15$ and $h_c = 0.15$, respectively, and the PUE is set to 1.2.

• **Workloads:** We consider that the data center serves both interactive and batch jobs, which are taken from the literature [12, 16], respectively. The maximum arrival rate for the batch jobs and interactive jobs are scaled to be 80% and 30% of the data center maximum capacity, respectively, while the maximum combined workload arrival rate still satisfies the peak data center capacity. Fig. 2(b) illustrates a snapshot of the traces for 3 days, normalized with respect to the maximum data center capacity.

• **Others:** We use the demand-responsive electricity prices modeled by the fitted function shown in [37]. We collect the temperature data in Mountain View, CA, and fuel mix data of California ISO [2,5] from October, 2012 to September, 2013. We use the EWIF and carbon emission rates for different electricity generation methods presented in [19,30,33] to calculate the EWIF and carbon emission rate for the data center. The first 3-day data for EWIF and carbon rate are shown in Fig. 2(a). Direct WUE is modeled based on empirical measurement [31].

4.2 Simulation Results

We now compare the performance of WACE with three benchmarks.

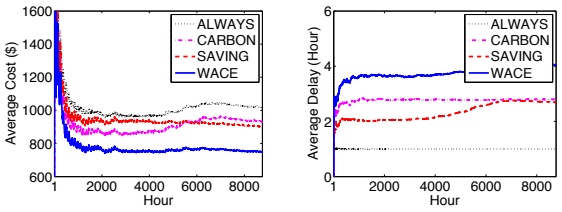
Benchmarks

The three benchmarks are described as follows.

• **SAVING:** SAVING only optimizes the electricity cost of the data center and is water- and carbon-oblivious. It applies WACE with zero weights for water and carbon.

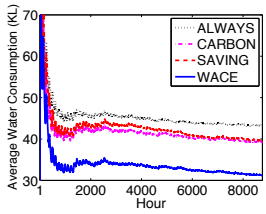
• **CARBON:** CARBON only optimizes the carbon emission of the data center and is electricity- and water-oblivious. Essentially, it applies WACE with an "infinite" weight for carbon.

• **ALWAYS:** ALWAYS does not use any optimization and tries to process jobs as soon as possible. This is the *de-factor* algorithm used in many data centers.

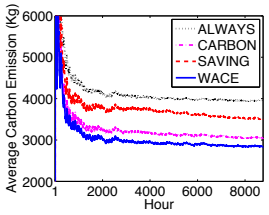


(a) Average total cost

(b) Average delay

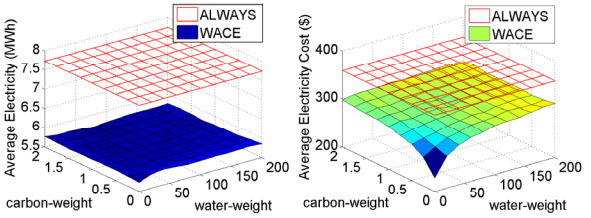


(c) Water consumption



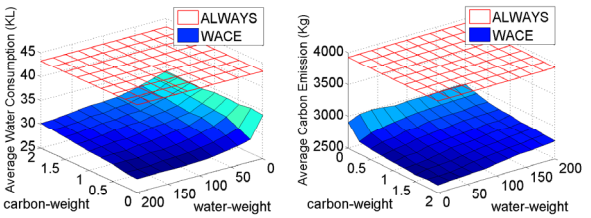
(d) Carbon emission

Figure 3: Comparison between WACE and benchmarks.



(a) Average electricity consumption

(b) Electricty Cost



(c) Water Consumption

(d) Carbon Emission

Figure 4: Impact of water and carbon weights.

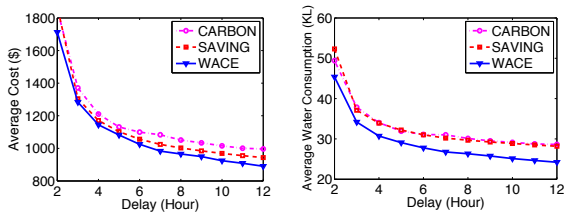
Performance comparison

Here, by fixing the cost-delay parameter V, we compare the performance of WACE with the three benchmark algorithms and show the results in Fig. 3, where the average value at any time slot t represents the cumulative average from 0 to t. In Fig. 3(a), we see that WACE achieves the lowest average total cost among all the algorithms. Compared to WACE, the other three algorithms, i.e., SAVING, CARBON and ALWAYS, incur a 20%, 24% and 36% higher cost, respectively. The delay performance comparison shows that ALWAYS has the lowest delay of 1 (due to its greedy nature), both SAVING and CARBON have an average delay around 3 time slots (i.e., hours), while WACE has a delay close to 4 time slots. The delay figure identifies that WACE is taking more advantage from the delay tolerance of batch jobs and hence achieves a lower average total cost by opportunistically processing batch jobs when the combined cost factor is relatively lower. The water consumption and carbon emission results show that compared to WACE, the benchmark algorithms, i.e., SAVING, CARBON and ALWAYS, incur more water consumption by 27%, 25% and 38.5% and higher carbon emission by 23%, 7.4% and 39%, respectively. This highlights the benefit of WACE in terms of sustainability (while the delay performance is compromised to a small and tolerable extent).

Impact of water and carbon weights

Now, we will study the impact of water weight (h_c) and carbon weight (h_w) on the performance of WACE. For both weighting factors, we start from zero and go up to the value which makes the water/carbon cost equal to 90% of the total cost, while keeping the other weight at its default value (i.e., $h_w = 15$, $h_c = 0.15$). We show a set of 3-dimension figures to capture all the possible combinations of water and carbon weights within the above mentioned range, and compare WACE with ALWAYS (which is the *de factor* reference algorithm, as currently many data centers are still performance-driven). In all cases, V is appropriately chosen for WACE to achieve an average delay equal to 4 hours.

Fig. 4(a) shows that the average electricity consumption remains almost same with varying water and carbon weights. This is because, the actual energy consumption for processing a fixed amount of workloads remains relatively the same, no matter WACE is trying to reduce carbon emission (i.e., high value of carbon weight) or water consumption (i.e., high value of water weight). The small variation in electricity consumption, however, can be attributed to the effect of discrete speed settings, which let servers run with variable dynamic energy consumption (but still fixed static energy as long as a server is turned on). From Fig. 4(b), we see that increase in either water or carbon weight increases the electricity cost. We have already seen that the weighting factors have little effect on the actual energy consumption, and hence the increased electricity cost implies the following fact: with increased water and/or carbon weight, WACE schedules batch jobs to find low water consumption and/or carbon emission due to sustainability considerations, not solely caring about the electricity cost. Fig. 4(c) and Fig. 4(d) show the decreasing trend of water consumption and carbon emission as the corresponding weighting factor is increased. The effect is straightforward, as increased weighting factor means a higher priority in the optimization algorithm.



(a) Average cost (b) Water Consumption

Figure 5: Impact of average delay constraint.

Next, we note that WACE has a lower electricity cost, carbon emission and water consumption compared to ALWAYS for a wide range of water and carbon weights, although no algorithms (including WACE) can possibly outperform SAVING/CARBON in terms of electricity cost/carbon emissions, as they solely minimize their respective metric. Nonetheless, Fig. 4 shows that, by appropriately choosing water and carbon weights, we will get a better performance from WACE than the widely-employed ALWAYS scheduling algorithm in terms of electricity cost, water consumption and carbon emissions (at the expense of increasing delay for batch jobs).

Impact of average delay constraint

Now, we study the relation of total cost and water with average delay performance. For this purpose, we vary the delay constraint from 2 to 12 time slots and show the corresponding cost and water consumption in Fig. 5. We see in Fig. 5(a) that the average total cost decreases for all algorithms with increased delay constraint (i.e., a less stringent delay requirement). In particular, given any delay constraint, WACE has the lowest average total cost. The performance gap between WACE and other two algorithms (i.e., SAVING and CARBON) increases with more relaxed delay constraint. ALWAYS is not shown as its delay is constantly 1 time slot. Fig. 5(b) shows a similar pattern of the change in water consumption with varying delay constraints. We can see that WACE has the lowest water consumption as it incorporates time-varying water efficiency when making scheduling decisions. Fig. 5 provides us with an important guidance for choosing an appropriate set of water and carbon weights so that the data center can run in low cost and/or reduced water footprints without much impact on the average delay performance.

We also conduct sensitive studies to demonstrate the robustness of WACE, and interested readers are referred to the technical report [12].

5 Related Work

In this section, we discuss the related work.

• **Data center optimization:** Several prior studies have focused on identifying methods of cost cutting while ensuring the quality-of-service. For example, finding a balance between energy cost of data center and performance loss through dynamically provisioning server capacity has been the primary focus of many recent studies [11, 16]. Other approaches include, but are not limited to, exploiting the spatio-temporal variation of electricity prices [15, 25, 26]. Cyber-physical approaches to optimizing data center cooling system and server management are also investigated [14, 24]. Electricity cost can be further reduced when the advantage of geographical load balancing is combined with the dynamic capacity provisioning approach [**?**, 26]. Nonetheless, none of these studies address water consumption in data centers.

• **Water reduction in data center:** Most of the existing efforts on water efficiency have been focusing on improved "engineering": for example, installing advanced cooling system [4], and using recycled water [10]. Our study focuses on integrating physical characteristic of time-varying water efficiency with control of data center's cyber asset (i.e., servers and workloads). More recent study [28] aims at optimizing water efficiency for delay-sensitive interactive workloads, but it does not apply to our study, because it neglects carbon footprints and does not exploit the temporal diversity of water efficiency. Another work [27] preliminarily minimizes water footprint via resource management, but it neglects many important factors, such as carbon emissions, demand-responsive electricity prices, discrete server speed selection and interactive jobs. Other research that is remotely related to data center water consumption includes [6], which develops a dashboard to visualize the water efficiency. To our best knowledge, holistically minimizing electricity cost, carbon emission and water footprint by leveraging the delay tolerance of batch jobs and temporal diversity of water efficiency has not been studied by any prior work.

6 Conclusions

In this paper, we studied water consumption in data centers and proposed an efficient online batch job scheduling algorithm, WACE, for minimizing the operational cost (incorporating electricity cost, water consumption and carbon emission) while bounding the average queue length. WACE exploits and integrates the physical characteristic of time-varying water efficiency with data center's cyber asset management (i.e., server provisioning and workload scheduling). We demonstrated the effectiveness of WACE via trace-based simulations, showing that WACE reduces the water consumption by over 27% compared to the state-of-the-art solutions, with a negligible delay increase.

References

[1] Apple and the environment, http://www.apple.com/environment/.

[2] California ISO, http://www.caiso.com/.

[3] AT&T Sustainability, http://www.att.com/gen/landing-pages?pid=24188.

[4] Facebook data center dashboard, http://www.fbpuewue.com.

[5] Historical Weather, http://www.wunderground.com/.

[6] C. Bash, T. Cader, Y. Chen, D. Gmach, R. Kaufman, D. Milojicic, A. Shah, and P. Sharma. Cloud sustainability dashboard, dynamically assessing sustainability of data centers and clouds. *HP Labs Tech. Report (HPL-2011-148)*, Sep. 2011.

[7] Facebook. Open sourcing pue, wue dashboards, https://code.facebook.com/posts/272417392924843/open-sourcing-pue-wue-dashboards.

[8] X. Fan, W.-D. Weber, and L. A. Barroso. Power provisioning for a warehouse-sized computer. In *ISCA*.

[9] P. X. Gao, A. R. Curtis, B. Wong, and S. Keshav. It's not easy being green. *SIGCOMM Comput. Commun. Rev.*, 42(4):211–222, Aug. 2012.

[10] Google's Data Center Efficiency. http://www.google.com/about/datacenters/.

[11] B. Guenter, N. Jain, and C. Williams. Managing cost, performance and reliability tradeoffs for energy-aware server provisioning. In *IEEE Infocom*, 2011.

[12] M. A. Islam, K. Ahmed, S. Ren, and G. Quan. Making data center less "thirsty" via online batch job scheduling. *Tech. Report*, 2014, http://users.cis.fiu.edu/~sren/doc/paper/icac_2014_water_full.pdf.

[13] J. G. Koomey. Growth in data center electricity use 2005 to 2010, 2011.

[14] L. Li, C.-J. M. Liang, J. Liu, S. Nath, A. Terzis, and C. Faloutsos. Thermocast: A cyber-physical forecasting model for datacenters. In *KDD*, 2011.

[15] M. Lin, Z. Liu, A. Wierman, and L. L. H. Andrew. Online algorithms for geographical load balancing. In *IGCC*, 2012.

[16] M. Lin, A. Wierman, L. L. H. Andrew, and E. Thereska. Dynamic right-sizing for power-proportional data centers. In *IEEE Infocom*, 2011.

[17] Z. Liu, Y. Chen, C. Bash, A. Wierman, D. Gmach, Z. Wang, M. Marwah, and C. Hyser. Renewable and cooling aware workload management for sustainable data centers. In *SIGMETRICS*, 2012.

[18] J. R. Lorch and A. J. Smit. Improving dynamic voltage scaling algorithms with pace. In *SIGMETRICS*, 2001.

[19] J. Macknick, R. Newmark, G. Heath, and K. Hallett. A review of operational water consumption and withdrawal factors for electricity generating technologies. *NREL Tech. Report: NREL/TP-6A20-50900*, 2011.

[20] U.S. Dept. of Energy. Energy demands on water resources. Dec. 2006.

[21] Microsoft Global Foundation Services. Microsoft recycles waste to provide clean power for data center R&D, 2012 (http://www.globalfoundationservices.com).

[22] M. J. Neely. *Stochastic Network Optimization with Application to Communication and Queueing Systems*. Morgan & Claypool, 2010.

[23] Nuclear Energy Institute. Environment: Emissions prevented, http://www.nei.org/resourcesandstats/nuclear_statistics/Environment-Emissions-Prevented.

[24] L. Parolini, N. Tolia, B. Sinopoli, and B. H. Krogh. A cyber-physical systems approach to energy management in data centers. In *ICCPS*, 2010.

[25] A. Qureshi, R. Weber, H. Balakrishnan, J. Guttag, and B. Maggs. Cutting the electric bill for internet-scale systems. In *SIGCOMM*, 2009.

[26] L. Rao, X. Liu, L. Xie, and W. Liu. Reducing electricity cost: Optimization of distributed internet data centers in a multi-electricity-market environment. In *IEEE Infocom*, 2010.

[27] S. Ren. Batch job scheduling for reducing water footprints in data center. In *Allerton*, 2013.

[28] S. Ren. Optimizing water efficiency in distributed data centers. In *Cloud and Green Computing*, 2013.

[29] M. J. Rutberg. Modeling water use at thermoelectric power plants, 2012.

[30] J. V. Spadaro, L. Langlois, and B. Hamilton. Greenhouse gas emissions of electricity generation chains: Assessing the difference. *IAEA bulletin*, 42(2):19–28, 2000.

[31] SPX Cooling. Water usage calculator, `http://spxcooling.com/green/leed/water-usage-calculator/`.

[32] The Green Grid. Water usage effectiveness (WUE): A green grid data center sustainability metric. *Whitepaper*, 2011.

[33] P. A. Torcellini, N. Long, and R. Judkoff. Consumptive water use for us power production, 2003.

[34] Uptime Institute. Data center industry survey, 2013, `http://uptimeinstitute.com/2013-survey-results`.

[35] R. Urgaonkar, B. Urgaonkar, M. J. Neely, and A. Sivasubramaniam. Optimal power cost management using stored energy in data centers. In *SIGMETRICS*, 2011.

[36] U.S. Green Building Council. Leadership in energy & environmental design, `http://www.usgbc.org/leed`.

[37] P. Wang, L. Rao, X. Liu, and Y. Qi. D-pro: Dynamic data center operations with demand-responsive electricity prices in smart grid. *IEEE Transactions on Smart Grid*, 3(4):1743–1754, December 2012.

[38] Y. Yao, L. Huang, A. Sharma, L. Golubchik, and M. J. Neely. Data centers power reduction: A two time scale approach for delay tolerant workloads. In *Infocom*, 2012.

[39] Y. Zhang, Y. Wang, and X. Wang. Electricity bill capping for cloud-scale data centers that impact the power markets. In *ICPP*, 2012.

Real-time Edge Analytics for Cyber Physical Systems using Compression Rates

Sokratis Kartakis and Julie A. McCann
Department of Computing, Imperial College London, UK
{*s.kartakis13, j.mccann*}*@imperial.ac.uk*

Abstract

There is a movement in many practical applications of Cyber-Physical Systems to push processing to the edge. This is particularly important were the CPS is carrying out monitoring and control, where the latency between the decision making and control message reception should be minimal. However, CPS are limited by the capabilities of the typically battery powered low resourced devices. In this paper we present a self-adaptive scheme that both reduces the amount of resources required to store high sample rate data at the edge and at the same time carries out initial data analytics. Using out Smart Water datasets, plus a selection from other real world CPS applications, we show that our algorithm reduces computation by 98%; data volumes by 55%; while requiring only 11KB of memory at runtime (including the compression algorithm). In addition we show that our system supports self-tuning and automatic reconfiguration which means that manual tuning is alleviated and the scheme can be both applied to any kind of raw data automatically and is able self-optimize as the nature of the incoming data changes over time.

1 Introduction

The work presented in this paper is part of a Smart Water project that both monitors water distribution networks (WDN) and controls its valves to optimize water network performance and lifetime over varying demands. ICT to support WDN typically consist of remote or online battery-powered telemetry units (data loggers) that record water data such as flow and pressure etc periodically over numbers of minutes and aggregate this data and send to a server periodically; typically via the mobile phone networks or 3G. Contemporary approaches use Wireless Sensor Network (WSN) [1], [2], [3], [4], [5] technologies to monitor the status of the water network and detect leakage or water bursts. The main drawbacks of these approaches are: (a) the analysis of the data takes place off-line, in base stations or servers meaning that optimal real time decision-making for control would be unrealistic and (b) the sensor nodes require a lot of energy, which places upper bounds on the amounts of data that can be sensed and relayed for analysis. There is a move to make WDN more dynamic and intelligent using wireless sensors and actuation effecting a CPS to monitor and optimally control the water network in real time, by pushing analytics to the edge and increasing the decision-making capacities of energy-constrained sensor nodes.

Typically such CPS projects monitor the dynamical conditions of the water distribution network. Traditionally this data is sensed at the edge of the network then sent to off-line servers to identify potential failures. Here further analysis via fusion with other data sets, such as customer data may take place. To do this, high precision pressure and flow data, at rates that can exceed 100 samples a second per sensor which can equate to high-precision data averaging at over 512bytes per sec or 0.45Mbytes per 15 minutes. If the system has to transmit this amount of data in 15-minute intervals then the communication process alone will drain the battery of the sensor node rapidly. Therefore, our aim is to reduce the energy cost related to the communication without sacrificing the precision of the data. To this end we evaluated a number of lossless compression algorithms.

During this evaluation, using real data, we observed a correlation between compression rate and data value fluctuation, and from this derived a scheme that enables the identification of transients or failures in the WDN. This means that instead of compressing raw sensor data and sending it to servers to be decompressed and then analyzed for anomalies, we can use the compression rate to detect anomalies and outliers directly on the sensor node. This is faster, more lightweight and provides early indications of an issue, which can be fed directly into the control function without having to communicate via servers saving time and energy. Furthermore, we have

expanded the system using ideas inspired from active learning to support optimal selection of the algorithms input parameters to enable self-tuning and automatic reconfiguration.

This paper is organized as follows: Section 2 contains our evaluation setup of the compression algorithm and presents our correlation observations. Section 3 describes our anomaly detection algorithm showing that the compression rate can be used in the indirect analysis of raw data. Section 4 presents the cross-evaluation system for the selection of optimal parameters. Section 5 describes the execution of the system using other kind of datasets, and section 6 discusses future work and concludes the paper.

2 Compression Rate and Raw Data Correlation

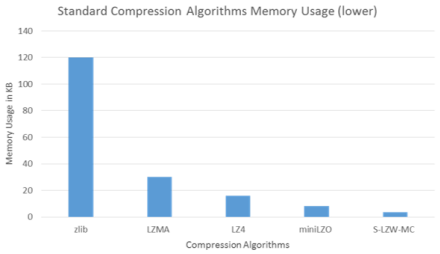


Figure 1: Compression algorithms evaluation.

In order to reduce the energy consumption that would be consumed by high data rate transmissions, while maintaining a high data precision, each sensor node uses lossless compression. Our choice takes memory and energy constraints of our devices into account, so computation and memory intensive algorithms are inappropriate, in spite of their potentially better compression rates. For example, current ultra low power MCUs have 64Kbytes memory[6], therefore we limit compression to 10K.

MiniLZO [7] (coding method is sliding window - LZ77), requires 8.192KB memory at runtime, and S-LZW-MC [8] (coding method is dictionary - LZ78), requires 3.250KB of memory (Figure 1). We evaluated the compression rate of each algorithm which we adapted to use in embedded platforms. Three different real datasets (Datasets A, B, and C - Figure 2 black line), provided by a large UK water company from their loggers were used [1]. Each data set consists of 5.5 million data pairs. In the evaluation the input stream is converted into 512-byte packets with the following structure: (a) timestamp (8-byte double data type), (b) 62 measurements (8-byte double * 62 = 469 bytes), and (c) CRC (8-byte double data type).

[1] We anonymize the company and dataset names for privacy reasons.

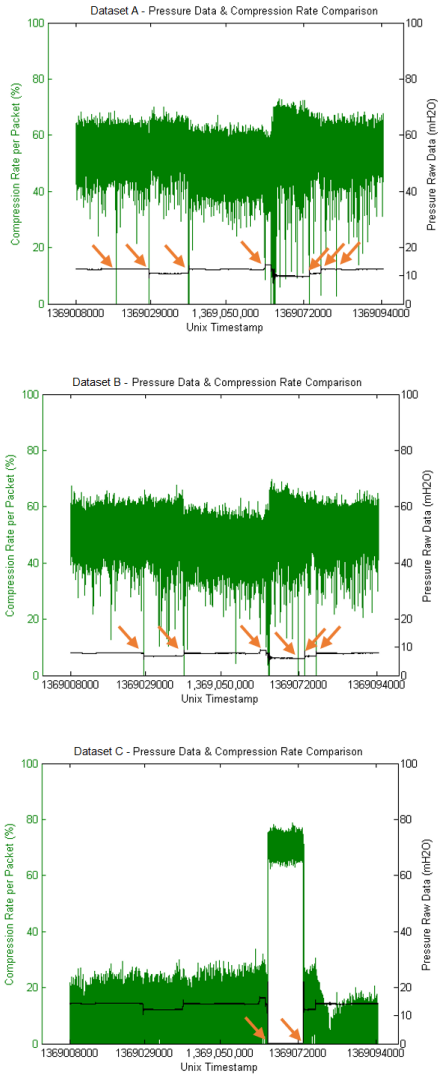


Figure 2: Compression rate & raw data comparison of Dataset A, B, and C.

Using the miniLZO compression algorithm, the results of the compression rate per packet can be seen in the three charts of Figure 2. Note that the original water pressure data is also overlaid on the same graphs in black. It is clear from Figure 2, that these traces highlight data anomalies as indicated by the orange arrows. From this, we formed the hypothesis that we could use the correlation of the compression rate and raw data.

We verified offline that the anomalies indicated on our graphs were true. From water technician logs we observed that they were valve position changes which were used to simulate water bursts, causing significant pressure data fluctuation. At these points the compression algorithm is unable to compress the data so the compression rate falls to 0%. In Figure 2, the drop in compression

rate isolates the areas of raw data where the fluctuation pattern is changeable. The Dataset C is more problematic and the water pressure fluctuation is quite high (see Figure 2c) resulting in the compression rate averaging at 16%. Further, in the same dataset a great drop in water pressure occurs (because the local valve was closed for a small period) which impacts compression rate, which increases to 70%.

Confident that our hypothesis was confirmed and we could use compression rate fluctuation to detect instabilities in high sample rate data, we derived an analytics algorithm that is performed at the end of the compression stage. This has the added advantage that the analytics cost m times less in terms of scale and complexity, where m is the number of measurements per packet. Because each packet contains 62 measurements (m = 62), the produced compression rates are approximately 89,000 (5,518,000 total measurements / 62 measurements per packet). Thus, the analysis is applied to 98% less values.

3 Anomaly Detection Algorithm

We produce a scheme to automatically detect significant changes in compression rate and therefore identify the timestamps of anomalies. To maximize the anomaly detection while minimizing the number of false-positive results, noise is removed from the compression rate stream using a one-dimensional Kalman Filter [9], [10] indicated in Figure 3b with a blue line. The use of Kalman filters is motivated by: (a) its support of streaming analysis using only the current input measurement (and therefore is memory efficient), (b) no matrix calculations are required (therefore it is computationally efficient), (c) ease of the algorithm tuning process, and (d) implementation simplicity.

For every new data value input, the Kalman Filter algorithm uses and updates the Kalman state. The Kalman state consists of the process noise covariance q, the measurement noise covariance r, the actual value x after noise removal, the estimation error covariance p, and the Kalman gain k. During the initialization process the parameters which need tuning are the noise q, the sensor noise r, the initial estimated error p and the initial value of x. The Kalman filter was manually initialized using the following parameters: q = 0.005, r = 25, p = 0, and x = the first compression rate measurement. In every new measurement, the algorithm updates the Kalman state using the following steps:

```
1: x = x
2: p = p + q
3: k = p / (p + r)
4: x = x + k * (measurement  x)
5: p = (1  k) * p
```

After noise removal, the anomalies can be detected accurately because according to Figure 3b (which presents

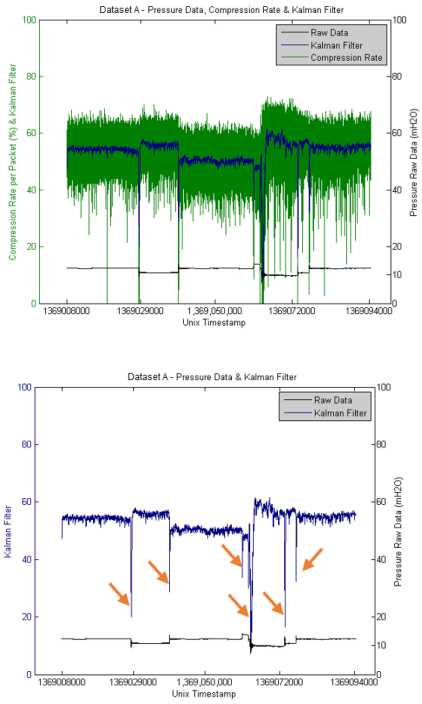


Figure 3: Apply Kalman Filter to Dataset A and drop detection - (w = 128, q = 0.005, r = 25).

the x value of Kalman Filter state and raw data) the anomalies are presented as great drops (orange arrows). The drops are being detected by using the average and the standard deviation of the compression rate moving average for a predefined window size w. We use this because it smoothes the states for easier analysis and reduces threshold computation to window sizes. Specifically, the algorithm computes the moving average of compression rate data with a window size w = 128 Kalman Filter x measurements, with the average *avg* and the standard deviation *std* of the moving average. In every Kalman state update, the algorithm checks if:

```
(Kalman state x) > (avg + std * l) ||
(Kalman state x) < (avg - std * l)
```

Where l represents the elasticity of the outlier detection (smaller values mean that the system is more sensitive - in Figure 4a l=3 and in b l=1.5). As can be observed in Figure 4a (Dataset C), the algorithm suffers from a cold start effect (it identifies the first values to be outliers because the moving average is not calculated). To solve this problem, the algorithm initializes the *avg* and computes *std* by using the current compression rate value. Furthermore, another problem occurs when a significant variation of compression rate data is detected (Figure 4a). In that case, because the standard deviation has a high value, the algorithm needs more intervals for the moving

average calculations to detect the outliers or anomalies. The solution is to reset the values, that is to initialize the *avg* and *std*, every time the distance between the boundaries created by the standard deviation become greater than a specific threshold t (in our system the threshold t = 35).

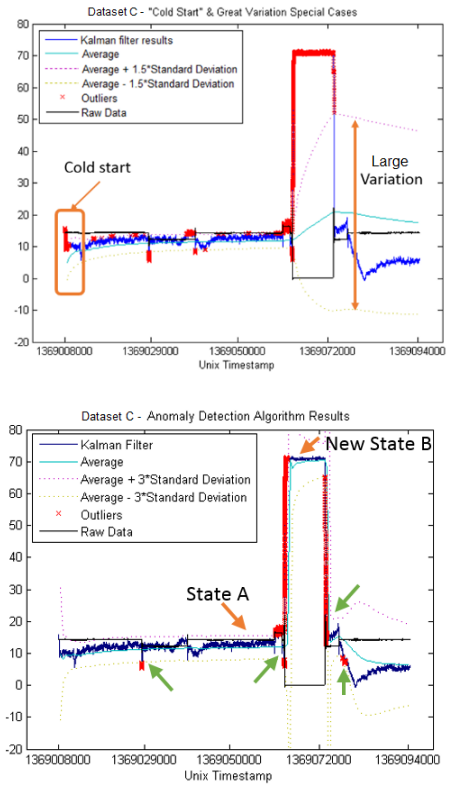


Figure 4: Dataset C - (a) "Cold Start", Large Variation and unelastic outliers detection (l = 1.5), and (b) fixed algorithm results (l=3) - (w = 128, q = 0.005, r = 25).

Figure 4b shows that this solves the cold start and large variation problems and illustrates the anomalies based on normalized data (green arrows). Furthermore, another benefit is that the algorithm can be adapted to changes in the behavior of the data stream. For example, Figure 4b, the algorithm detected the anomaly (red markers x value = timestamp) when the compression rate changes from 10% to 70% as there is no immediate drop (State A to new State B), the algorithm recognizes that the system has a new steady state (B) until the next drop from 70% to 10%. Therefore, this shows that the algorithm adapt extremely fast to new conditions/states.

We applied this approach to Dataset A and B, and Figure 4 presents the results for Dataset A(raw data = black line). The red x markers are the anomalous values detected; the green arrows illustrate the process of matching the timestamps between compressed and raw data.

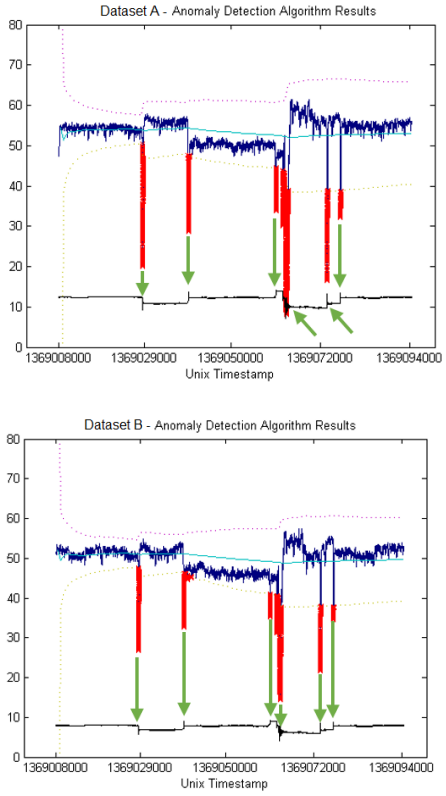


Figure 5: Algorithm results (l=3) for (a) Dataset A and (b) Dataset B - (w = 128, q = 0.005, r = 25).

4 Input Parameters Optimal Tuning

According to the above analysis, by tuning the input parameters, our algorithm can be applied to any case of high sample rate anomaly detection in hardware-constrained sensor nodes. However, to maximise the performance of this, an element of tuning is required as observed in the previous section. Table 1 aggregates all the tuning parameters required by our algorithm.

Table 1: Algorithm input parameters

Process	Parameters
Input stream split	Packet size m
Input stream data precision	Measurement bytes
Kalman Filter initialization	Noise q Sensor noise r Initial estimated error p
Moving average computation	Window size w
Boundaries creation	Elasticity l
Great variation threshold	Threshold t

The initialization of input parameters using a manual approach is inappropriate because it requires permutations of all the different combinations of parameters val-

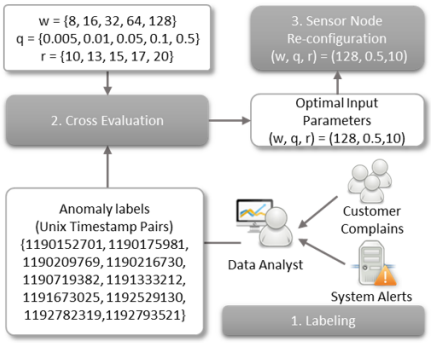


Figure 6: Sensor node re-configuration process.

ues. In order to optimize the algorithms tuning process, we borrow ideas from active learning techniques [11]. This approach requires feedback from real users or an offline system to identify true anomalies, but this is not onerous[2]. Here true anomalies for a single representative training dataset are labeled.

For the results that we present here, we applied the active learning idea by asking water data technicians to manually label anomalies on a subset of our evaluation data. Then, we created an offline cross-evaluation system of Figure 6, which uses our algorithm and calculates the correct, false/positive (FP), and true/negative (TN) anomaly detections based on the initial labeling. This establishes the optimal input parameters as the combination that maximizes the following distance:

```
Distance D = [Correct - (FP + TN)] Detections
```

Using this, one can imagine that a system would update parameters to re-configure the in-node anomaly detection algorithm over time. The data analysis component could recognize that the system requires reconfiguration using customer complains (anomalies are being missed) and system alarms (which increase when the water network is unstable).

Before the creation of the cross-evaluation system, the algorithm's input parameters for datesets were selected manually. In the previous section we show that correct anomalies were identified however here we show that the cross-evaluation system further improves the algorithms accuracy significantly. The reason is that manual observation of high sample rate data is difficult because the high density data. For example, Figure 7 presents the results of the Distance D of each different combination of the following parameter sets for Dataset A:

```
w = {64, 128, 512, 1024}
q = {0.001, 0.005, 0.05}
```

[2]Data anomaly detection can be confirmed off-line automatically by correlating candidate stream data anomalies with other data sets such as customer or water technician records.

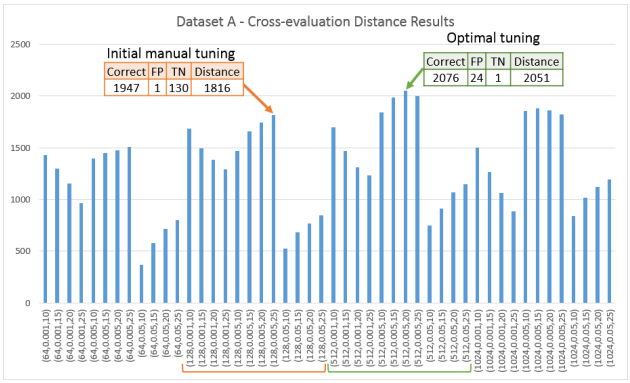


Figure 7: Cross evaluation distance calculation results - Dataset A.

```
r = {10, 15, 20, 25}
```

The orange arrow on Figure 7 are showing input parameters we derived manually, which were $w = 128$, $q = 0.005$, $r = 25$ (Figure 7 orange combinations). The initial selection of parameters was made on a moving average window w = 128 because our intuition was that a smaller window used to calculate the thresholds, would provide greater accuracy. However, the cross-evaluation system shows that the optimal combination would have a window of $w = 512$, $q = 0.005$, $r = 20$ (Figure 7 green arrow) where the distance D indicates that accuracy will be increased by 15%. Because the cross-evaluation system uses an exhaustive approach, it always returns the optimal combination of input parameters reducing the effort and time to find the optimal combination manually.

5 Using Different Datasets

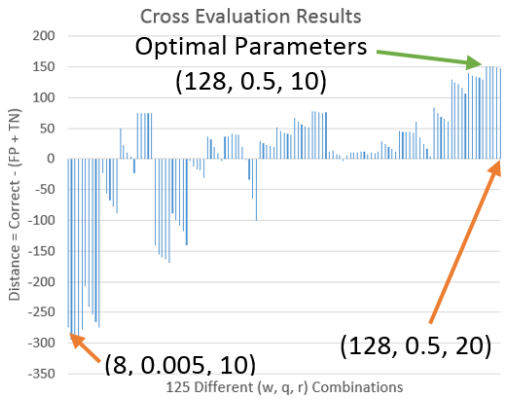


Figure 8: Cross evaluation distance calculation results - Temperature Dataset.

To understand the generality of the work beyond water applications, we applied our cross-evaluation system to datasets from St Bernard Mountain Pass sensor nodes

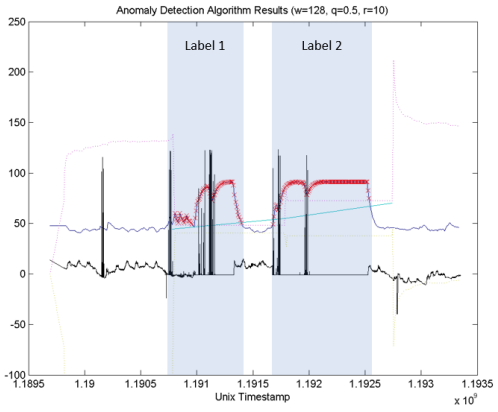


Figure 9: Temperature data anomaly detection results - (w = 128, q = 0.5, r = 10).

in Switzerland [12] reporting temperature, soil moisture and watermark measurements.

Figure 8 and Figure 9 illustrates the results of our cross-evaluation system for the temperature dataset. During the labeling process, we defined two periods as anomalies and we executed our cross evaluation process which was initiated using the following:

```
w = {8, 16, 32, 64, 128}
q = {0.005, 0.01, 0.05, 0.1, 0.5}
r = {10, 13, 15, 17, 20}
```

Figure 8 presents the results of distance D per combination of input parameters, where the optimal combination (w = 128, q = 0.5, r = 10) with 170 correct and 21 error anomaly detection (D = 151). Furthermore, from Figure Figure 9 we can infer that the anomalies are detected precisely and in order to verify our results we manually checked 125 cross evaluation runs. We repeated the same process for soil moisture, and watermark sensor measurements from ten different sensor nodes and we achieved similar results, but due to lack of space we do not include them in this paper.

6 Future Work & Conclusion

This paper presents a scheme that combines lightweight compression and anomaly detection for Cyber-Physical Systems. The work has been developed as part of a Smart Water project and we show that it not only significantly reduces the amount of communications between sensor devices and the cloud, but also that early transient or event (such as water bursts) detection can run on low-resourced sensor nodes meaning that local control functions can occur with minimal latency. The main contribution is the innovative approach to analyzing high sample rate data by using compression rate rather than raw data. The main benefits of our system are: (a) the size of the program at run time, which can be applied in embedded systems, (b) data reduction (and proportional communication and energy costs) by 55%, (c) computation reduction by 98%, (d) the algorithm can be applied independently of the content of the raw data with an appropriate initial tuning, (e) the adaptation of our method to match trend/state changes in the raw data. We extend the system to be able to derive initialization parameters for self-configuration and to adjust said parameters as the nature of the underpinning data changes over time thus showing significant performance improvements over manual tuning by a further 15%.

Future work of our approach is to examine the effect of changes in data precision (e.g. by using float instead of double values), to test our algorithm with other lightweight compression algorithms.

7 Acknowledgments

This work forms part of the Big Data Technology for Smart Water Nets research project funded by NEC Corporation, Japan.

The hydraulic pressure datasets used in this paper have been provided by Dr Ivan Stoianov and Mr Asher Hoskins, Dept Civil Engineering, Imperial College London, UK.

References

[1] Babak Aghaei. Using wireless sensor network in water, electricity and gas industry. In *Electronics Computer Technology (ICECT), 2011 3rd International Conference on*, volume 2, pages 14–17. IEEE, 2011.

[2] Alexandre Santos and Mohamed Younis. A sensor network for non-intrusive and efficient leak detection in long pipelines. In *Wireless Days (WD), 2011 IFIP*, pages 1–6. IEEE, 2011.

[3] WANG Zhu, HAO Xiao-qiang, and WEI De-bao. Remote water quality monitoring system based on wsn and gprs [j]. *Instrument Technique and Sensor*, 1:018, 2010.

[4] Michael Allen, Ami Preis, Mudasser Iqbal, and Andrew J Whittle. Water distribution system monitoring and decision support using a wireless sensor network. In *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), 2013 14th ACIS International Conference on*, pages 641–646. IEEE, 2013.

[5] Ivan Stoianov, Lama Nachman, Sam Madden, Timur Tokmouline, and M Csail. Pipenet: A wireless sensor network for pipeline monitoring. In *Information Processing in Sensor Networks, 2007. IPSN 2007. 6th International Symposium on*, pages 264–273. IEEE, 2007.

[6] Atmel. Atmel AVR 8-bit and 32-bit Microcontrollers. `http://www.atmel.com/products/microcontrollers/avr/default.aspx`, 2014. [Online; accessed 20-March-2014].

[7] Jan Kraus and Viktor Bubla. Optimal methods for data storage in performance measuring and monitoring devices. In *Proceedings of Electronic Power Engineering Conference*, 2008.

[8] Christopher M Sadler and Margaret Martonosi. Data compression algorithms for energy-constrained devices in delay tolerant networks. In *Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 265–278. ACM, 2006.

[9] Interactive Matter Lab. Filtering Sensor Data with a Kalman Filter. `http://interactive-matter.eu/blog/2009/12/18/filtering-sensor-data-with-a-kalman-filter/`, 2009. [Online; accessed 20-March-2014].

[10] Reza Olfati-Saber. Distributed kalman filter with embedded consensus filters. In *Decision and Control, 2005 and 2005 European Control Conference. CDC-ECC'05. 44th IEEE Conference on*, pages 8179–8184. IEEE, 2005.

[11] Burr Settles. Active learning literature survey. *University of Wisconsin, Madison*, 52:55–66, 2010.

[12] Guillermo Barrenetxea, François Ingelrest, Gunnar Schaefer, and Martin Vetterli. Wireless sensor networks for environmental monitoring: the sensorscope experience. In *Communications, 2008 IEEE International Zurich Seminar on*, pages 98–101. IEEE, 2008.

Self-Optimizing Citizen-centric Mobile Urban Sensing Systems

Usman Adeel Shusen Yang Julie A. McCann
Department of Computing, Imperial College London, UK

Abstract

In this paper, we develop a novel networking scheme that supports both real-time and delay-tolerant urban sensing applications. This maintains optimality through self-adapting its communications strategy using either inexpensive short-range opportunistic transmissions or reliable long-range cellular radios. Core to this scheme is the trading of mobile sensor data in a virtual market where we demonstrate that our scheme can incentivize phone users to participate. We show that the scheme can optimise network throughput while minimising total phone costs, in terms of 3G and battery costs.

1 Introduction

The integration of sensing, computing and communication capabilities in mobile devices has turned them to a powerful computing and sensing platform. The ubiquity of these sensor-rich smart-phones is beginning to play an increasingly important role in the evolution of cyber-physical systems (CPS) in urban scenarios.

Sensing is a crucial component for CPS, which process and react to the data gathered from the physical environment autonomously. The sheer numbers of mobile phone users combined with the relatively powerful computing and communication capabilities of modern phones, make mobile sensing a much more flexible and cost-effective paradigm than traditional CPS such as Wireless sensor Networks. Furthermore, the inherent phone owner mobility enables increased sensing coverage both spatially and over time; providing opportunities to collect data at a higher granularity and with broader coverage. Mobile sensing can exploit the social structures of the physical world to improve the performance of cyber world and in doing so provides better services to the users in the physical world by optimising the organization of the available resources in cyber world. This paves the way towards large-scale citizen-centric urban sensing applications for smart cities [12].

Figure 1 illustrates a typical Mobile Urban Sensing System (MUSS). According to the demands of specific sensing applications, mobile phones can produce sensing data such as available parking places, traffic congestion, noise levels, air pollution, and smart meter readings. The sensor data can be sent to the MUSS server through cellular communication or be multi-hopped via short-range radios such as WiFi direct, Bluetooth, and LTE direct.

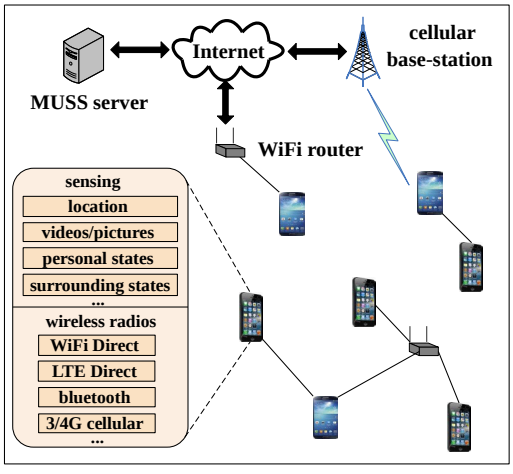


Figure 1: Conceptual illustration of the mobile urban sensing system.

Work carried out in this paper has been funded by the EIT ICTLabs - Cyber Physical Systems Action Line, I3C project and the ICRI-Cities Institute

A MUSS has the following distinguishing characteristics:

1. **Large Sensing Coverage and huge volume of data:** MUSS has the potential to monitor every interesting element in a city, and then relay sensed data to the Internet. As a result, it is predicted to be one of the major sources of big data.
2. **Self-organization/Self-healing:** In MUSS, mobile phone users can join and leave the network very frequently. Similarly disasters and the dynamic nature of urban environments can cause network failure. Therefore, MUSS should adapt autonomously to current network states such as channel conditions and evolving logical network topologies.
3. **Heterogeneous data types:** Due to the diversity of MUSS applications and their potential to require large sensing coverage, sensor data formats will vary considerably; from simple physical readings such as ambient light to video frames. In turn this data would differ in lifetime, monetary value, and privacy level, etc.
4. **Social/Economic Concerns of Self-Interest Phone Users** Mobile phone users may not be willing to fulfil a MUSS task, due to privacy concerns and the potential costs that would be incurred impacting battery usage and the amount of money they pay for communications 3/4G. Therefore, taking account of the social and economic behaviours of phone users is vital to the success of MUSS.

In this paper, we show how to provide autonomic and cost-effective networking services for MUSS considering all the features discussed above. We firstly present a brief review of mobile phone sensing research, and examine the main communication techniques designed to support MUSS. We develop a joint pricing and data routing scheme aiming to support both real-time and delay-tolerant MUSS applications by seamlessly combining cellular communication [9] and opportunistic networking [2]. Off-loading communication to low cost(free), short range communications releases the burden on traditional communications technologies which will reach upper physical bounds if all future city CPS systems use them.

2 Background

In this section, we present a brief review of current mobile sensing research and communication support for mobile phone sensing. For recent comprehensive surveys, we refer the reader to [8, 7, 4].

2.1 Mobile Phone Sensing

According to different sensing scales, mobile phone sensing applications can be categorized into personal sensing, community sensing, and public sensing; HyperFit [6], CenceMe [10],CarTel [5] are examples of personal, community and public sensing deployments respectively. Community and public sensing can play significant role in the development of CPS in smart cities where one wishes to better understand large-scale phenomena through citizens collaboration.

Defined by awareness of phone users, mobile phone sensing can be classified into two different sensing paradigms: *Participatory Sensing* requires active participation from the phone users in terms of collecting and sampling the data. e.g., manual entry of lowest prices or deals for goods or taking a picture.*Opportunistic Sensing* shifts the burden of MUSS tasks from the phone users to the background sensing system, which makes it more suitable for community/public sensing.

Currently, the majority of mobile sensing applications send sensing data directly to the server through single-hop 3/4G *cellular radio communications*. However, due to limitations such as 3/4G costs to the phone users [9] and cellular system's capacity bounds [3], using cellular communication solely would not be a feasible solution for the potential huge volume of urban sensing data.

With the increase in the short-range communication capabilities of smart phones, such as in WiFi Direct for Android OS 4.0+, efficient neighbour discovery [1], and the development of smart Device-to-Device (D2D) communications [3]; it becomes more and more promising to use *opportunistic networking* [2] for delay-tolerant MUSS applications [1, 13, 14]. By leveraging inherent human mobility and low-cost short-range communication, sensor data can be sent to base-stations (e.g. WiFi routers) in a carry-and-forward fashion by relaying the data in short hops via

different mobile phones.

This opportunistic networking can significantly reduce energy and telephony costs for phone users and at the same time mitigate sensor data traffic load over cellular communication channels.

3 A Citizen-centric Networking Scheme for MUSS

In this section, we present our lightweight and fully distributed networking scheme to support both real-time and delay-tolerant MUSS applications in a cost-effective way, through the combination of cellular communication and WiFi direct. Specifically, we consider a MUSS network that consists of three types of nodes: smart phones, static WiFi routers, and a cellular base-station as shown in Figure 2 Each phone can report sensed data to the mobile sensing server through 3G cellular radio directly, or through a WiFi router nearby. In addition, two nearby phones have the opportunity to communicate directly to each other through WiFi Direct during their contact duration, such as phones B and C shown in Figure 2. In our model, each data packet produced by a smart phone has a monetary value (e.g. which can be represented in terms of a national currency or tokens to be traded in other ways such as to purchase mobile phone apps). Further, each packet is has a lifetime *e.g.*, 10 minutes, and its duration is tightly coupled to the worth specific applications attribute to the packet. The MUSS operates in discrete time with a unit time slot t =1, 2, ... Every phone x maintains a data buffer that stores the sensor data packets generated by its own sensors, and the data received from other phones.

3.1 Algorithm Description

At every time slot t =1, 2, ..., our scheme operates as follows:

Sensor Data Sampling

1. According to the requirements of the MUSS application (e.g. the demands of external MUSS users), each phone x generates sensor data packet(s), and then assigns its monetary value and initial Time-To-Live (TTL) value to each packet. Then, x inserts the sensor data packets into its phone data buffer.

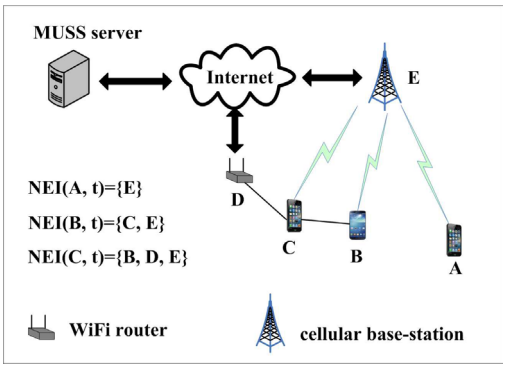


Figure 2: Example of MUSS network to describe the proposed scheme.

Instantaneous Neighbour Discovery

2. Each node x builds a one-hop neighbour table $NEI(x,t)$, consisting of the cellular base-station, and all phones and WiFi routers that can connect to x through WiFi radios at current slot t. Neighbour discovery schemes such as [1] can be used. Take Figure 2 for instance, the instantaneous one-hop neighbour table of phone C, $NEI(C,t)$, consists of three nodes: B, D, and E.

Transmission Quality Estimation

3. Each node x estimates its transmission capacity, $rate_{x,y}(t)$, between itself and each of its instantaneous neighbours y in $NEI(x,t)$, i.e. the maximum number of packets that x can transmit to y, based on the data rates of their wireless radios and WiFi duty-cycle settings of x and y [1].

4. Each node x estimates the monetary costs of sending and receiving a packet, denoted as $scost_x(t)$ and $rcost_x(t)$ respectively, based on its remaining energy, system resource usage, and 3G bills costs. It worth noting that if x is a WiFi router or the cellular base-station, its receiving cost, $rcost_x(t)$, is equal to zero.

Pricing

5. Each phone x sets its current data selling price, $sell_x(t)$, as the total monetary value of the data packets in its data buffer multiplied by a positive system parameter α, set by the server. For instance, if $\alpha = 0.1$ and x's data packets are worth 10 cents,

therefore $sell_x(t) = 1$ cent per packet. Then phone x communicates the selling price $sell_x(t)$ to all nodes in $NEI(x,t)$. Recall, the selling prices of any cellular base-station and WiFi router are set as zero for every slot.

Profits computing

6. Each phone x computes the potential individual profits, $profits_{x,y}(t)$, it could obtain by selling data to each of its neighbours y in $NEI(x,t)$. $profits_{x,y}(t)$ is computed as a function of the cost (that would be incurred in this potential data trading) and selling price differences between x and y

$$\begin{aligned} & profits_{x,y}(t) \\ &= (sell_x(t) - sell_y(t) - scost_x(t) \\ &\quad -rcost_y(t))rate_{x,y}(t) \end{aligned} \tag{1}$$

Data Trading

7. Denote y^* as the neighbour that can currently give phone x maximum profits if it sells on its data. If $profits^*_{x,y}(t) > 0$, then x sells $rate^*_{x,y}$ (t) number of packets to y. Note that the number of packets are a function of the communications rate so as to not overload that link. A data packet with a smaller TTL will be forwarded with a higher priority. Packets which have reached a 0 TTL value will be dropped as they are deemed no longer useful to the application.

8. Upon receiving data packets from the seller x, the buyer y^* pays $(sell_x(t) - sell_y(t) - rcost_y(t))rate_{x,y}(t)$ total amount of money to x, which means that the cost incurred in this trade is paid by the seller x.

In this scheme, devices can seamlessly switch between short range and long-range communication. At a moment in time each seller node selects the neighbour node with the minimal price and minimal transmission cost as the potential buyer (The node to recieve the data). It encourages system to transmit data to other neighbours using short-range communication due to high cost of long-range communication (3G).

The above scheme is very lightweight, as it implements simple arithmetic calculations and does not require any historic information to be maintained. Also it does not require future knowledge of mobile phones and their trajectory to be speculated.

3.2 Throughput Optamality and Self- *

Since the total value of the data carried by each phone in its buffer is proportional to its queue backlog, it can be verified that the proposed scheme implicitly solves a stochastic optimization problem (i.e. we minimize the total transmit and receive costs for all phones) in a fully distributed way, by using the Lyapunov "drift-plus-penalty" method [11]. According to Lyapunov optimization theory, optimal throughput and long-term minimization of global system costs can be achieved, by controlling the weight between queue backlogs and communications costs [11]. In our scheme, this weight is controlled by the price scaling parameter α. Based on the Lyapunov "drift-plus-penalty" method, it is not difficult to verify that as α decreases, the global system costs (total cost of all phone users) also decrease, but the average queue backlogs increase resulting an increase in end-to-end transmission delays. Therefore, by controlling the pricing parameter α it is not difficult to prove that the proposed scheme can not only achieve throughput optimality, which is highly desirable when transmitting large volumes urban sensing data; but it can also minimize the total cost incurred by the phone users [14, 11].

Besides achieving throughput optimality, our scheme exhibits the following autonomic behaviours.

1. *Self-optimization* Since the neighbour table on each phone can include a cellular base-station, and all WiFi routers and other phones nearby, the phones can optimize their profit by automatically switching data transmission between WiFi radio and cellular radio, according to selling prices and transmission costs.

2. *Self-organization* This scheme is fully distributed, because it requires only the local information of each mobile phone and its current one-hop neighbours. This enables MUSS to self-organize based on current network state and topology. Moreover it is flexible enough to cope with partial failure of communication infrastructure *e.g.*, by natural disasters and can scale across urban space.

4 Evaluation

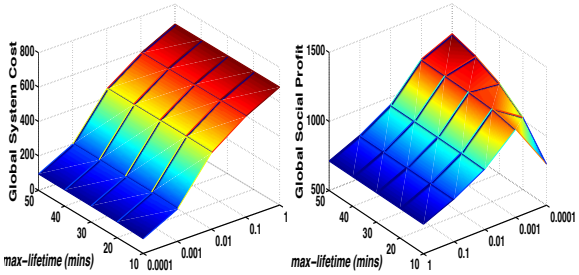
4.1 Simulation Settings

To evaluate the performance of our scheme, we constructed extensive simulations using the realistic simulator Castalia (http://castalia.npc.nicta.com.au/). We randomly deployed a 151-node MUSS in a $800m \times 800m$ geographic area, consisting of 10 WiFi routers, 140 mobile phones, and one cellular base-station. We set the duration of a slot to 1 second and each simulation lasts for 10^6 seconds (around 12 days). The transmission ranges of the WiFi direct radio was set to 50 meters i.e. the typical WiFi direct transmission range in practice (http://www.wi-fi.org). The time-varying transmission capacities of all cellular and WiFi radios were randomly set between 1 and 50 packets per second. We used a realistic human mobility model, Heterogeneous Human Walk (HHW) [15], to simulate the mobility of smart phones. The movement speed of each phone was randomly distributed between 1 and 10m/s (i.e. representing walking speeds and typical urban vehicular speeds).

Each sensor and mobile phone produces sensor packets with a random monetary value of 10 credits at a rate of one packet per second. For every mobile phone, the receiving and transmitting costs of WiFi radios were randomly set between 0.1 and 1 credits per packet, while that of the cellular communications were set between 1 and 10 credits per packet.

4.2 Impact of Packet Lifetime and Pricing Parameter α

In this set of simulations, we study the impact of different packet lifetimes and the pricing parameter α on the global system cost and global social profits. The lifetime (i.e. the initial TTL value) of each generated packet was randomly set between 5 seconds and the $max-lifetime$ minute, this latter parameter is a simulation variable ranging from 10 to 50 minutes. The randomness of the packet lifetime assignment can reflect the heterogeneity of mobile sensing data. The simulation results are shown in Figure 3a and Figure 3b. In all simulations, around 10%-65% of the sensor data traffic is sent through cellular radios, and the rest is sent over WiFi direct radios.

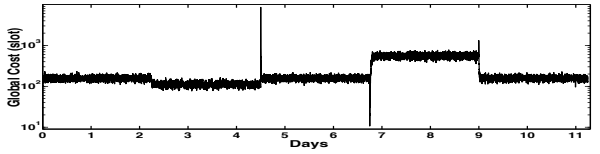


(a) Global Cost (b) Global Profits

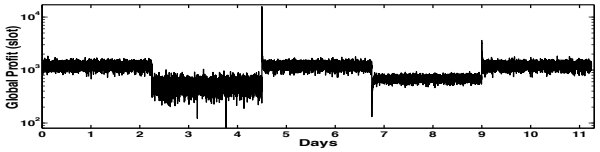
Figure 3: Impact of sensor data lifetime and parameter α on System Cost and Global Profits.

We use time-average global system costs and global social profits (both in credits per second) to measure the performance of our scheme. Here the global system cost is measured as the sum of both the transmission and reception costs of all phones, and global social profits is computed as the total value of all the successfully received packets (by the MUSS server) minus the total system cost.

As illustrated in Figure 3a, the total system cost shows a monotonically decreasing trend as the pricing parameter α decreases; this verifies our optimal throughput discussion in Subsection 3.2. By setting a sufficiently small α, the global system cost can be arbitrarily close to the minimal, according to Lyapunov optimization theory. However, the end-to-end delay becomes large as α decreases, resulting a higher risk of a packet being dropped, taking TTL into account. This is reflected in Figure 3b, where the global social profit shows a concave curve as α decrease when the packet life time is large. This is caused by the joint effects of decreased system cost and increased in dropped packets. When max-lifetime is sufficiently large, global social profits exhibit a monotonically increasing function of α. This is because the impact of packet loss caused by expired TTL on the global social profits can be ignored. It is worth noting that every phone obtained positive profit in all simulations. This means that our scheme manages to incentivize phone users to participate in the MUSS because they receive a fair reward.



(a) Dynamic Global Cost



(b) Dynamic Global Profits

Figure 4: Impact of Dynamic environment on System Cost and Global Profits.

4.3 Self-Configuration in Dynamic Environment

To study the ability of our proposed scheme to adapt to changing scenarios, we constructed experiments by dividing the total simulation time into five periods of equal duration of 2×10^5 secs. In first period, MUSS operates normally with operational cellular and Wifi communication. In the second period, we disabled the cellular communication of all nodes so that data packets can only be transmitted and received directly through Wifi direct radios (simulating cellular failure similar to what has occured in disaster situations). MUSS returns back to normal (mixed) state in the third period. In fourth period we disabled Wifi direct communication between all the nodes in the network, so that nodes can only transmit data packets through cellular communication. Finally, network returns again to normal state in the fifth period.

Here we used global system costs and global social profits in every slot (both in credits) to measure the effect of changing topology over time. Here the global system cost is measured as the sum of both the transmission and reception costs of all phones in a slot. Global social profits is computed as the total value of all the successfully received packets (by the MUSS server) minus the total system cost in a slot.

In Figure 4, we can see that in the second period, the global system cost decreases when we cellular communication is disabled. This is due to all the transmissions being relayed through Wifi direct only, which is cheaper than cellular communication. However, global phone user profits also decrease in spite of the decrease in cost. This is due to the large delay in multi-hop transmission which results in increased number of dropped packets with smaller TTL. In the fourth period, system costs increase significantly when Wifi direct communication is disabled due to the high cost of cellular communication. This is also reflected by the decrease in global profits of the MUSS. We can also see that the network self-adjusts very quickly to the changing conditions of the network. When the network returns to normal operation in third and fifth period, the data buffer of the phones contain large numbers of data packets that are sent instantly after availably of alternate option. This is the reason behind sudden spikes in system cost and global profits at the start of these periods. Once the backlog reduces, the system becomes stable.

5 Conclusion

In this paper, we study how to provide a cost-effective networking service for real-time and delay tolerant applications in Mobile Urban Sensing System (MUSS). We first highlight the challenges of MUSSs and review current mobile sensing research. Then we propose a joint pricing and routing scheme to support both real-time and delay-tolerant MUSS applications through seamless integration of cellular and short-range communications of mobile phones. The proposed scheme is not only lightweight and fully distributed, but can also achieve optimal throughput, which is highly suitable to deliver large amount of mobile sensing data. Through simulations, we demonstrate that our scheme can minimize global system costs, as well as effectively incentivize phone users to participate in the MUSS. We also show that our scheme self adapts to dynamic network conditions. To support future complex MUSSs, many open research challenges remain including faithful sensor data market design for discouraging phone users to subvert the market through misinformation, networking schemes with social privacy awareness, as well as joint sensor data analysis, filtering and networking.

References

[1] Bakht, M., Trower, M., and Kravets, R. H. Searchlight: won't you be my neighbor? In *Proc. ACM MOBICOM* (2012), pp. 185–196.

[2] Conti, M., Giordano, S., May, M., and Passarella, A. From opportunistic networks to opportunistic computing. *IEEE Communications Magazine 48*, 9 (2010), 126–139.

[3] Doppler, K., Rinne, M., Wijting, C., Ribeiro, C. B., and Hugl, K. Device-to-device communication as an underlay to lte-advanced networks. *IEEE Communications Magazine 47*, 12 (2009), 42–49.

[4] Ganti, R. K., Ye, F., and Lei, H. Mobile crowdsensing: current state and future challenges. *IEEE Commun. Mag. 49*, 11 (2011), 32–39.

[5] Hull, B., Bychkovsky, V., Zhang, Y., Chen, K., Goraczko, M., Miu, A., Shih, E., Balakrishnan, H., and Madden, S. Cartel: a distributed mobile sensor computing system. In *Proc. ACM SenSys* (2006), pp. 125–138.

[6] Jarvinen, P., Jarvinen, T., Lahteenmaki, L., and Sodergard, C. Hyperfit: hybrid media in personal nutrition and exercise management. In *IEEE Second International Conference on Pervasive Computing Technologies for Healthcare.* (2008), pp. 222–226.

[7] Khan, W. Z., Xiang, Y., Aalsalem, M. Y., and Arshad, Q. Mobile phone sensing systems: a survey. *IEEE Communications Surveys & Tutorials 15*, 1 (2013), 402–427.

[8] Lane, N. D., Miluzzo, E., Lu, H., Peebles, D., Choudhury, T., and Campbell, A. T. A survey of mobile phone sensing. *IEEE Communications Magazine 48*, 9 (2010), 140–150.

[9] Liu, H., Hu, S., Zheng, W., Xie, Z., Wang, S., Hui, P., and Abdelzaher, T. Efficient 3g budget utilization in mobile participatory sensing applications. In *Proc. IEEE INFOCOM* (2013), pp. 1411–1419.

[10] Miluzzo, E., Lane, N. D., Fodor, K., Peterson, R., Lu, H., Musolesi, M., Eisenman, S. B., Zheng, X., and Campbell, A. T. Sensing meets mobile social networks: the design, implementation and evaluation of the cenceme application. In *Proc. ACM SenSys* (2008), pp. 337–350.

[11] Neely, M. J. Stochastic network optimization with application to communication and queueing systems. *Synthesis Lectures on Communication Networks 3*, 1 (2010), 1–211.

[12] O'Grady, M., and O'Hare, G. How smart is your city? *Science 335*, 6076 (2012), 1581–1582.

[13] Park, U., and Heidemann, J. Data muling with mobile phones for sensornets. In *Proc. ACM SenSys* (2011), pp. 162–175.

[14] Yang, S., Adeel, U., and McCann, J. A. Selfish mules: Social profit maximization in sparse sensornets using rationally-selfish human relays. *IEEE Journal on Selected Areas in Communications 31*, 6 (2013), 1124–1134.

[15] Yang, S., Yang, X., Zhang, C., and Spyrou, E. Using social network theory for modeling human mobility. *IEEE Network 24*, 5 (2010), 6–13.

Gait Recognition Using Encodings With Flexible Similarity Measures

Michael B. Crouse Kevin Chen H. T. Kung
School of Engineering and Applied Sciences
Harvard University

Abstract

Gait signals detectable by sensors on ubiquitous personal devices such as smartphones can reveal characteristics unique to each individual, and thereby offer a new approach to recognizing users. Conventional pattern matching approaches use inner-product based distance measures which are not robust to common variations in time-series analysis (e.g., shifts and stretching). This is unfortunate given that it is well understood that capturing such variations is paramount for model performance. This work shows how machine learning methods which encode gait signals into a feature space based on a dictionary can use convolution and Dynamic Time Warping (DTW) similarity measures to improve classification accuracy in a variety of situations common to gait recognition. We also show that data augmentation is crucial in gait recognition, as diverse training data in practical applications is very limited. We validate the effectiveness of these methods empirically, and demonstrate the identification of user gait patterns where shift and stretch variations in measurements are substantial. We present a new gait dataset that contains a complete representation of the variations that can be expected in real-world recognition scenarios. We compare our techniques against the current state of the art gait period detection and normalization schemes on our dataset and show improved classification accuracy under all experimental scenarios.

1 Introduction

The Internet of Things (IoT) has created an explosion of sensor data due to the increased number of devices with embedded sensors, ranging from smart watches and phones to healthcare wearables and head-mounted devices. These devices, combined with new environments such as connected consumer smart homes, have opened a new set of applications and scenarios directly enabled by sensor data. Novel techniques that utilize and extract meaningful information from this vast stream of data are the key to success in this new area.

Much of this new sensor data is measured over time. The analysis of time-series data is a well studied subject explored in the signal processing community for applications such as networking, computer vision and speech recognition. More recently, there has been substantial interest in individual motion tracking and personalized gesture recognition for improved human and machine interaction with applications in user interface design and gaming. These efforts leverage the recent growth in the number and type of devices containing sensors for motion, audio, and video, yet there remains much work to be done in how to make best use of such a diverse set of sensors.

Gait recognition is one form of motion tracking that has been studied for a variety of reasons including fall detection, locomotion for robotics, and health/fitness monitoring. Human gait is the result of the cyclic motion of a set of human limbs [2]. Gait has also been used to uniquely identify a human using visual sensors (video cameras) as well as motion sensors. Using biometrics such as gait or fingerprints for identifying users is an important approach that will become increasingly useful in IoT. Gait is an ideal target as it can be obtained passively, monitored using a variety of sensors, and provides a fairly robust indicator of identity. Gait is also promising as a user identification mechanism that could improve the usability and security provided by current authentication schemes.

Machine learning techniques have achieved great successes in the fields of computer vision and speech recognition. In this paper, we adapt the same framework and propose a representation encoding method tailored to gait recognition, and report improved performance over the best published results.

2 Related Work

User identification using gait patterns with motion-based sensors has been the subject of much study over the past decade. Most of the approaches regarding gait recognition utilize accelerometers attached to subject for gathering data. In many cases, the accuracy of these systems are fairly high, but require multiple commercial sensors with several fixed points and extensive configuration [5, 10]. Recently, due to the integration of accelerometers and gyroscopes into smart phones, several new approaches have been proposed to reduce the number of sensors and relax the constrains on sensor placement [7, 4, 8, 9]. The most common technique for dealing with time-series data is through gait period detection, which involves locating the strike points of a subjects' gait signal. A strike point corresponds to a subject's heel striking the ground.

The work by Juefei-Xu et. al. is on recognizing a user's gait pattern recorded from off the shelf Android devices with users walking down a hallway [7]. Their technique relies heavily on normalization around the strike points. Their two normalization methods are: 1) centering measurements around a strike-point and 2) measurements between consecutive strike-points, interpolated to get uniform segment lengths. This approach attempts to address shift and stretch variance that naturally occurs in a human's gait between steps. Frank et. al. proposes the use of nonlinear dynamic systems to form a geometric time delay embedding per subject [4]. After training, they classify each new sample by selecting the nearest-neighbor using Euclidean distance. Their model has high accuracy but requires large segments of data to be able to perform classification accurately.

Our approach uses a general machine learning model coupled with traditional signal processing similarity measures. Specifically, we utilize a dictionary/encoding framework and use convolution and Dynamic Time Warping (DTW) measures for encoding to provide more robust representations for classification in a scenario with large intra-class variation. We contribute a unique dataset that was captured by a smart phone under a wide variety of conditions including different paces, phone orientations, and over different days. We report the highest accuracy compared with the traditional gait period detection and normalization methods.

3 Problem Definition

Utilizing accelerometer and gyroscope measurements of users' gait for recognition is fundamentally a time-series analysis problem. As previously mentioned, time-series analysis is a well-studied area explored mostly by the signal processing community, one of the key applications being voice recognition and speech translation. While gait recognition involves similar problems, it has several unique properties and challenges that need to be addressed. Unlike audio data captured from speech, gait pattern data captured by motion sensors is periodic in nature. Periodicity in the data provides both benefits in terms of recognition and challenges in determining the appropriate unit to perform recognition (a single step or an ensemble of steps). Additionally, measured gait patterns for a single user can have large variations between days due to factors such as phone placement on a user's body and terrain differences. Another source of variation stems from different paces, as humans rarely maintain a single pace while walking and between walks due to an assortment of variables including mood, environment and destination.

Past research has shown the feasibility of gait pattern recognition using data captured from accelerometers and gyroscopes. However, it remains a challenging problem due to several reasons: (1) lack of public datasets, (2) the recognition scheme needs to work with a very small amount of training data, (3) high degree of signal variations (pace/phone placement, etc).

We have identified several imperative criteria for any system that attempts gait recognition:

1. robust under different phone placement
2. robust under different walking speed
3. robust between different days
4. requiring only a small amount of training data

Our approach utilizes conventional machine learning techniques coupled with traditional time-series similarity measures to address each of these necessary conditions.

4 Methodologies

With sufficient samples, we can separate classes well using our feature representations, as demonstrated by the 99% accuracy for the complete training case in Table 2. We have observed the same phenomena in another public dataset [3], of which more than 98% of the samples are linearly separable if trained on the entire dataset.[1] In the following section we describe our approach of addressing the situation where only limited amount of samples are available. Specifically, we need to handle large variations in gait signals of the same user over different days and at different paces using the very few samples available during the training phase for better handling of variations such as shifts and stretches.

[1] This is generally not true in most other applications.

We propose a pipeline that includes (1) preprocessing, (2) feature encoding, (3) linear Support Vector Machine (SVM) classification, and (4) data augmentation. Preprocessing is a fixed process that makes our method insensitive to sensor orientation. Feature encoding deals with more variations, and requires (unlabeled) data for training. We use a linear Support Vector Machine (SVM) to classify samples in their feature representations. Finally, data augmentation is used to increase sample diversity.

4.1 Preprocessing

Gyroscopes and accelerometers both report readings for 3 directions (or axes). Let a segment x_{raw} be a $3 \times T$ matrix, where T is the length of the signal. We compute the 3×1 principal eigenvector v from x_{raw}, and then compute $x = v^T x_{raw}$. This makes x insensitive to sensor orientation.

4.2 Feature Encoding

We use a dictionary/coding framework for representation encoding. In particular, we consider the simplest setting: random patch dictionary and distance encoding. We construct the random patch dictionary D by selecting random samples from the training set. In distance encoding, a feature vector f for a given sample x is computed by $f_i = \mathsf{dist}(x, d_i)$ with components f_i where $\{d_i\}$ are the entries in the dictionary, and dist is some distance measure. By encoding we mean the process of computing the feature vector f for a given sample x. As we will see, samples corresponding to gait signals of different users are readily separable with linear SVM when samples are expressed in their feature representations.

- **Encoding with Convolutional Distance Measure**
  The most obvious variation in gait signal is perhaps the shift. A shift is a change translation in time of a gait signal. When a gait signal is captured from a user, the alignment for comparison with other signals is unknown and must be accounted for. Therefore, we want the feature encoding to be shift invariant, so that shifted versions of the same pattern would be transformed to the same feature vector. To this end, we use the distance measure defined as follows:

  $$\mathsf{dist}_{\mathsf{conv}}(a,b) = \max(|\mathsf{conv}(a_r, b)|)$$

  where a_r is a in the reverse order. This finds the offset that gives maximum correlation. We will use $\mathsf{Enc}_{\mathsf{conv}}$ to denote encodings with $\mathsf{dist}_{\mathsf{conv}}$.

- **Encoding with DTW Distance Measure**
  Another classical measure for evaluating similarity between time series is dynamic time warping (DTW) [11]. DTW finds an optimal "path" that can morph one signal to another. For gait signal, this means DTW will likely consider x and x' similar if they are compressing/stretching variations of one another. Figure 2 shows a cartoon that demonstrates the flexibility of this measure. The DTW distance measure is defined as follows:

  $$\mathsf{dist}_{\mathsf{dtw}}(a,b) = \mathsf{DTW}_{\mathsf{cost}}(a,b)$$

  We will use $\mathsf{Enc}_{\mathsf{dtw}}$ to denote encodings with $\mathsf{dist}_{\mathsf{dtw}}$. Note that in the DTW algorithm, one may specify the largest matching range, which in our case can be conveniently set to half of the largest step size.

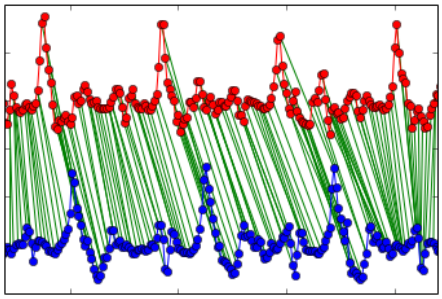


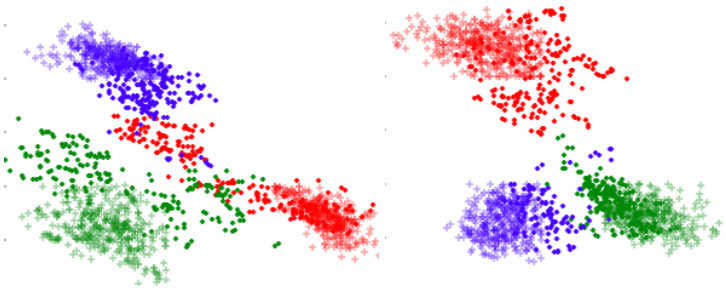
Figure 2: DTW alignment between two segments of the same subject. Blue is sampled from a normal pace session, and red is from a fast pace session. Note that $\mathsf{dist}_{\mathsf{dtw}}$ for this pair of signals would be small because DTW found an optimal way to align them.

In Figure 1(a), we illustrate how our encoding methods can make data more separable. This visualization is made by projecting encoded samples from 3 randomly selected classes onto 2D. One can see that the training data is very separable and form tight clusters for $\mathsf{Enc}_{\mathsf{conv}}$. However, the test examples do not necessarily fall into the correct clusters and may be on the wrong side of the decision boundary. We alleviate this problem with data augmentation.

4.3 Data Augmentation

Besides designing variation-tolerant encodings, data augmentation is another way to deal with variations in samples. For example, applying a shift-invariant encoding corresponds to augmenting the data with shifted-variations of observed data in terms of improving the match between the training and testing distributions.

We identify three major natural variations in gait patterns: shifts, stretching, and compression. Stretching and compression corresponds to scaling the signal in the



(a) $\mathsf{Enc}_{\mathsf{conv}}$ (b) $\mathsf{Enc}_{\mathsf{conv}}$ with data augmentation

Figure 1: Samples projected onto decision planes trained on encoded representations in the feature space with and without augmentation. Transparent plus signs (+) represent samples from the training set, and opaque dots are samples from testing set. Note that data augmentation reduces the amount of overlap between classes.

time axis. We can acquire these variations in the training set by sampling a session in the following way: (1) select a random starting point, (2) select random window size and re-sample signal within the window into a fixed length. Samples generated this way have all three types of variations. In Figure 1(b), we show the advantage of data augmentation using convolution as the similarity measure.

5 Dataset Generation

In this section, we will describe how our dataset is generated and collected. We also provide a brief overview of accelerometers and gyroscopes in terms of what they measure.

5.1 Devices

IoT devices, specifically wearables, are becoming prevalent and most include some form of motion sensing chip. For example, wearables such as the Pebble Watch and Nike+ FuelBand contain 3-axis accelerometers [1]. These sensors are prevalent in smart phones including Apple's iPhone and most Android devices. Our experimental device is an Android HTC Droid DNA placed in the subject's front left pocket. The Android smart phone captures both accelerometer and gyroscope data and records it locally before it is uploaded as a batch for processing. We were able to sample at 50 Hz for both the accelerometer and gyroscope, which should be sufficient for capturing the necessary characteristics unique to a subject's gait.

5.2 Accelerometer

An accelerometer measures the change of position of a test mass. Accelerometers react to a large number of external forces including linear motion, gravity, centripetal force, and other motions [6]. The measurement taken from an accelerometer is the sum of all these forces in terms of acceleration. A 3-axis accelerometer provides measurements along 3 orthogonal axes, x, y, z.

day 1	day 2
P=1, O=1, Pace=1	P=1, O=1, Pace=1
P=2, O=1, Pace=1	P=1, O=1, Pace=2
-	P=1, O=2, Pace=1

Table 1: Experimental configurations recorded for each subject. Each cell is a single session corresponding to 50 seconds sensor data. For day 1 we collect 2 sessions for each setting. P=path (1,2), O=orientation (1,2), Pace (1=normal, 2=fast)

5.3 Gyroscope

Gyroscopes are sensors that measure the angular velocity of an object. They measure the rate of rotation around a single axis. Smart phones and wearables often contain 3-axis gyroscopes capable of extremely accurate, low-latency measurements. These sensors provide a good balance to the accelerometer as the measurements are not biased by gravity or magnetic forces and are less noisy [6].

5.4 Data Collection and Description

Gait measurements were collected under a variety of device orientations, paces, over different paths and days. We define a session of gait measurements as a single walk around our pre-defined course by a subject. Each session is between 40 and 50 seconds in duration where a smart device is placed in the subjects front left pocket

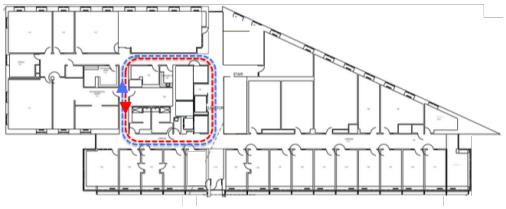


Figure 3: Subjects walk on two paths: counterclockwise and clockwise around the hallway Each subject walks each path twice for a total of 4 recorded walks.

and the watch placed on their left wrist. The orientation of the device is always vertical, with the device's z-axis pointed upward, and we alter whether the device screen faces outward, either away from or facing the subject. We have two different paces, one normal corresponding to 3 to 5 feet per second and one brisk at 4 to 6 feet per second. The two paths can be seen in Figure 3. The paths correspond to a loop around an office hallway, one clockwise and the other counter-clockwise. We recorded all of the subjects on two separate days. The break-down of session configurations is listed in Table 1. We recorded 31 different subjects from our office building, an assortment of students and staff. We have full data for 31 subjects under the basic walk experimental configuration. Our dataset is robust in the sense that we have 9 subjects with multiple days and variations including phone orientation and pace.

6 Experiments

We describe 5 different experiments using our gait dataset. The complete training set serves as a baseline for evaluating the capability of our model and confirming our intuition described in Section 4. The other 4 experiments are designed to observe increasing levels of variance between training and testing data. We consider the following settings:

1. **Complete Training Set**: We take all sessions from the dataset, break them down into segments, and split the set of segments into training and testing sets. The training set is complete because it contains samples from every session. Note that the testing set is still using disjoint set of samples.

2. **Different Sessions**: We take sessions from the same day under the same orientation and pace setting, and split them into training and testing sessions.

3. **Different Orientation**: Sessions are taken from the same day under same pace, and then split by orientation into training and testing sessions.

4. **Different Days**: Sessions are taken under same pace and same orientation, and then split by days into training and testing sessions.

5. **Different pace**: Sessions are taken from the same day under same orientation, and then split by pace into training and testing sessions.

Each training and testing segment contains 300 samples, which roughly correspond to 6 to 7 seconds.

6.1 Evaluation of Data Augmentation and Feature Encoding

We explore the intra-class variations under different settings, and show how different schemes performs under these variations. The reported performance is the multi-class classification accuracy in each of the described settings. Classification accuracy is the number of correctly labeled predictions over the number of total examples in the test set. We discuss the impact of data augmentation and the empirical results of two similarity measures, convolution and DTW.

6.1.1 Data Augmentation

We extract segments of randomized length from training sessions, and then re-sample them with bi-cubic interpolation into segments of a fixed length. In these experiments, the window size is sampled from $\mathsf{Gaussian}(300, 30)$, where the standard deviation (30) is selected empirically to give the best result. The resulting segments are of length 300.

The performance of the convolution encoding with and without data augmentation is shown in the first 2 columns of Table 2. For the complete training set, classification is near perfect because there is little variation within the same session (as shown earlier in Section 4). The convolution measure captures most of the variations between different sessions with the larger number of subjects. It is also able to capture most of the variations under changes in orientations with 88% accuracy. It is important to note that this is with a lower number of subjects.

For sessions over different days we see a larger accuracy drop, implying a more significant change in gait pattern over days. The lower accuracy for different pace is expected, because the convolve-transform is not intended to be scale invariant. With data augmentation, the accuracy is comparable in the first three setting because the variation introduced by our synthetic data is not necessarily important for the testing set. On the other hand, we get significantly better results by having synthetic data for predicting sessions of different pace. This suggests

setting (# subjects)	$\mathsf{Enc}_{\mathsf{conv}}$	$\mathsf{Enc}_{\mathsf{conv}}+\mathsf{aug}$	$\mathsf{Enc}_{\mathsf{dtw}}$	α	β
complete training set (31)	.99	.99	.99	.94	.90
different sessions (31)	.88	.88	.99	.90	.88
different orientation (9)	.88	.92	.73	n/a	n/a
different days (9)	.69	.77	.76	.48	.53
different pace (9)	.48	.70	.79	.75	.76

Table 2: Comparison across different methods. It shows a significant improvement of $\mathsf{Enc}_{\mathsf{conv}}$ by data augmentation. Data augmentation does not change the outcome of $\mathsf{Enc}_{\mathsf{dtw}}$ much and is therefore not listed on the table. α and β are normalization-based methods proposed by Juefei-Xu et. al. Note the first two experiments consist of 31 subjects while the others are only 9.

that our data augmentation captures some variation between different pace and over days.

6.1.2 Encoding Gains with $\mathsf{Enc}_{\mathsf{conv}}$ and $\mathsf{Enc}_{\mathsf{dtw}}$

As described in Section 4, it is important for the transformation to capture legitimate variances related to pace. In Section 4 we hypothesize that the purpose for encoding in our case is to generalize training data. We demonstrate the performance of $\mathsf{Enc}_{\mathsf{dtw}}$ and $\mathsf{Enc}_{\mathsf{conv}}$ in Table 2 (column 2 and 4) in order to empirically validate these claims.

Our method outperforms the state-of-art for general, individual sessions, and inter-day sessions. Many existing approaches rely on normalization techniques to reduce intra-class variations. While they can achieve reasonable accuracy with normalized data, the normalization process has two drawbacks. The process requires a a long sequence to capture the periodicity necessary for determining gait cycles. Secondly, the normalization process is based on heuristics that require parametric tuning per sensor.

Juefei-Xu et. al. proposes two normalization schemes, α and β, that consist of gait period detection via using peak finding heuristics [7]. The normalization method α places the strike point in the center of each segment. The normalization method β uses sampled data between two strike points, using interpolation and re-sampling to yield equal length sequences. Their β normalization provides stretch invariance due to re-sampling the measurements between pairs of steps to yield segments of equal length.

We implemented both of their normalization methods and performed parameter screening to yield the best possible performance on our dataset using their approach (their dataset was not publicly available at the time of this writing).

Table 2 provides a comparison of our encoding schemes versus their gait period detection and normalization methods. We are able to provide improved accuracy on our dataset in all settings for $\mathsf{Enc}_{\mathsf{dtw}}$ and all but pace under $\mathsf{Enc}_{\mathsf{conv}}$ with data augmentation. The pace scenario shows that the normalization methods α and β do provide some stretch invariance whereas $\mathsf{Enc}_{\mathsf{conv}}$ does not. However, we have shown in the previous section that the gap can be bridged by data augmentation. We also show that $\mathsf{Enc}_{\mathsf{dtw}}$ outperforms the normalization-based technique by being more tolerant to variation.[2]

We have observed that insufficient training data is a challenge for gait recognition — segments within a session are practically identical, so most gait datasets have effectively less than 5 samples from each class. However, if given enough samples, gait signals seem linearly separable without involving complex non-linear transformations typical of many machine learning techniques (see first row in Table 2). Unlike most machine learning problems, the task in gait signal recognition is not about finding a non-linear transformation such that classes become linearly separable, but should instead be focused on generalizing the available training data.

7 Conclusion

We have shown that gait signals are readily separable using our encoding which requires almost no data preprocessing. This is observed in our dataset as well as other public gait datasets. The implication is that unlike many other classical classification problems (e.g., computer vision), there is no need to learn a complicated non-linear transform to make the data easily separable.

We identify that the main challenge in gait pattern classification is the disparity (e.g., different pace) between training and testing data, due to a small number of effective samples. We discuss the characteristics of gait signals, and show how feature encoding and data augmentation alleviates the this problem. Our encoding based method outperforms the best published result in terms of short-segment gait signal classification.

[2] We attempted to introduce our orientation transformations to their normalization schemes but it significantly degraded their performance in all scenarios.

8 Acknowledgements

This material is based on research sponsored in part by the Intel Corporation. The authors would like to thank Marcus Comiter for useful discussion and suggestions. We would also like to acknowledge all the participants that contributed walks to our dataset.

References

[1] Pebble teardown. http://www.ifixit.com/. Accessed: 2014-03-04.

[2] BOYD, J. E., AND LITTLE, J. J. Biometric gait recognition. In *Advanced Studies in Biometrics*. Springer, 2005, pp. 19–42.

[3] FRANK, J., MANNOR, S., AND PRECUP, D. Data sets: Mobile phone gait recognition data, 2010.

[4] FRANK, J., MANNOR, S., AND PRECUP, D. A novel similarity measure for time series data with applications to gait and activity recognition. In *Proceedings of the 12th ACM international conference adjunct papers on Ubiquitous computing-Adjunct* (2010), ACM, pp. 407–408.

[5] GAFUROV, D., SNEKKENES, E., AND BOURS, P. Gait authentication and identification using wearable accelerometer sensor. In *Automatic Identification Advanced Technologies, 2007 IEEE Workshop on* (2007), IEEE, pp. 220–225.

[6] GOEHL, D., AND SACHS, D. Motion sensors gaining inertia with popular consumer electronics. *White Paper, IvenSense Inc* (2007).

[7] JUEFEI-XU, F., BHAGAVATULA, C., JAECH, A., PRASAD, U., AND SAVVIDES, M. Gait-id on the move: pace independent human identification using cell phone accelerometer dynamics. In *Biometrics: Theory, Applications and Systems (BTAS), 2012 IEEE Fifth International Conference on* (2012), IEEE, pp. 8–15.

[8] KOBAYASHI, T., HASIDA, K., AND OTSU, N. Rotation invariant feature extraction from 3-d acceleration signals. In *Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on* (2011), IEEE, pp. 3684–3687.

[9] NGO, T. T., MAKIHARA, Y., NAGAHARA, H., MUKAIGAWA, Y., AND YAGI, Y. The largest inertial sensor-based gait database and performance evaluation of gait-based personal authentication. *Pattern Recognition 47*, 1 (2014), 228–237.

[10] RONG, L., JIANZHONG, Z., MING, L., AND XIANGFENG, H. A wearable acceleration sensor system for gait recognition. In *Industrial Electronics and Applications, 2007. ICIEA 2007. 2nd IEEE Conference on* (2007), IEEE, pp. 2654–2659.

[11] SAKOE, H., AND CHIBA, S. A dynamic programming approach to continuous speech recognition. In *Proceedings of the seventh international congress on acoustics* (1971), vol. 3, pp. 65–69.

On-demand, Spot, or Both: Dynamic Resource Allocation for Executing Batch Jobs in the Cloud

Ishai Menache
Microsoft Research

Ohad Shamir
Weizmann Institute

Navendu Jain
Microsoft Research

Abstract

Cloud computing provides an attractive computing paradigm in which computational resources are rented on-demand to users with zero capital and maintenance costs. Cloud providers offer different pricing options to meet computing requirements of a wide variety of applications. An attractive option for batch computing is *spot-instances*, which allows users to place bids for spare computing instances and rent them at a (often) substantially lower price compared to the fixed *on-demand* price. However, this raises three main challenges for users: how many instances to rent at any time? what type (on-demand, spot, or both)? and what bid value to use for spot instances? In particular, renting on-demand risks high costs while renting spot instances risks job interruption and delayed completion when the spot market price exceeds the bid. This paper introduces an online learning algorithm for resource allocation to address this fundamental tradeoff between computation cost and performance. Our algorithm dynamically adapts resource allocation by learning from its performance on prior job executions while incorporating history of spot prices and workload characteristics. We provide theoretical bounds on its performance and prove that the average *regret* of our approach (compared to the best policy in hindsight) vanishes to zero with time. Evaluation on traces from a large datacenter cluster shows that our algorithm outperforms greedy allocation heuristics and quickly converges to a small set of best performing policies.

1 Introduction

This paper presents an online learning approach that allocates resources for executing batch jobs on cloud platforms by adaptively managing the tradeoff between the cost of renting compute instances and the user-centric utility of finishing jobs by their specified due dates. Cloud computing is revolutionizing computing as a service due to its cost-efficiency and flexibility. By allowing multiplexing of large resources pools among users, the cloud enables *agility*—the ability to dynamically scale-out and scale-in application instances across hosting servers. Major cloud computing providers include Amazon EC2, Microsoft's Windows Azure, Google AppEngine, and IBM's Smart Business cloud offerings.

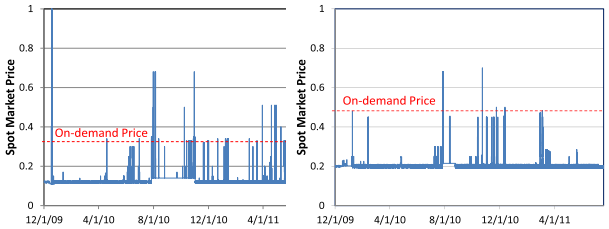


Figure 1: The variation in Amazon EC2 spot market prices for 'large' computing instances in the US East-coast region: Linux (left) and Windows (right). The fixed on-demand price for Linux and Windows instances is 0.34 and 0.48, respectively.

The common cloud pricing schemes are (i) *reserved*, (ii) *on-demand*, and (iii) *spot*. Reserved instances offer users to make a one-time payment for reserving instances over 1-3 years and then receive discounted hourly pricing on usage. On-demand instances allow users to pay for instances by the hour without any long-term commitment. Spot instances, offered by Amazon EC2, allow users to bid for spare instances and to run them as long as their bid price is above the spot market price. For *batch applications* with flexibility on when they can run (e.g., Monte Carlo simulations, software testing, image processing, web crawling), renting spot instances can significantly reduce the execution costs. Indeed, several enterprises claim to save 50%-66% in computing costs by using spot instances over on-demand instances, or their combination [3].

Reserved instances are most beneficial for hosting long running services (e.g., web applications), and may

also be used for batch jobs, especially if future load can be predicted [19]. The focus of this work, however, is on managing the choice between on-demand and spot instances, which are suitable for batch jobs that perform computation for a bounded period. Customers face a fundamental challenge of how to combine on-demand and spot instances to execute their jobs. On one hand, always renting on-demand incurs high costs. On the other hand, spot instances with a low bid price risks high delay before the job gets started (till the bid is accepted), or frequent interruption during its execution (when the spot market price exceeds the bid). Figure 1 shows the variation in Amazon EC2 spot prices for their US east coast region for Linux and Windows instances of type 'large'. We observe that spot market prices exhibit a significant fluctuation, and at times exceed even the on-demand price. For batch jobs requiring strict completion deadlines, this fluctuation can directly impact the result quality. For example, web search requires frequent crawling and update of search index as the freshness of this data affects the end-user experience, product purchases, and advertisement revenues [2].

Unfortunately, most customers resort to simple heuristics to address these issues while renting computing instances; we exemplify this observation by analyzing several case studies, reported on the Amazon EC2 website [3]. Litmus [16] offers testing tools to marketing professionals for their web site designs and email campaigns. Its heuristic for resource allocation is to first launch spot instances and then on-demand instances if spot instances do not get allocated within 20 minutes. Their bid price is set to be above the on-demand price to improve the probability of their bid getting accepted. Similarly, BrowserMob [7], a startup that provides website load testing and monitoring services, attempts to launch spot instances first at a low bid price. If instances do not launch within 7 minutes, it switches to on-demand. Other companies manually assign delay sensitive jobs to on-demand instances, and delay-tolerant ones to spot instances. In general, these schemes do not provide any payoff guarantees or how far do they operate from the optimal cost vs. performance point. Further, as expected, these approaches are limited in terms of explored policies, which account for only a small portion of the state space. Note that a strawman of simply waiting for the spot instances at the lowest price and purchasing in bulk risks delayed job completion, insufficient resources (due to limit on spot instances and job parallelism constraints), or both. Therefore, given fluctuating and unpredictable spot prices (Fig. 1), users do not have an effective way of reinforcing the better performing policies.

In this paper, we propose an online learning approach for automated resource allocation for batch applications, which balances the fundamental tradeoff between cloud computing costs and job due dates. Intuitively, given a set of jobs and resource allocation policies, our algorithm continuously adjusts per-policy weights based on their performance on job executions, in order to reinforce best performing policies. In addition, the learning method takes into account prior history of spot prices and characteristics of input jobs to adapt policy weights. Finally, to prevent overfitting to only a small set of policies, our approach allows defining a broad range of parameterized policy combinations (based on discussion with users and cloud operators) such as (a) rent on-demand, spot instances, or both; (b) vary spot bid prices in a predefined range; and (c) choose bid value based on past spot market prices. Note that these policy combinations are illustrative, not comprehensive, in the sense that additional parameterized families of policies can be defined and integrated into our framework. Likewise, our learning approach can incorporate other resource allocation parameters being provided by cloud platforms e.g., Virtual Machine (VM) instance type, datacenter/region.

Our proposed algorithm is based on machine learning approaches (e.g., [8]), which aim to learn good performing policies given a set of candidate policies. While these schemes provide performance guarantees with respect to the optimal policy in hindsight, they are not applicable *as-is* to our problem. In particular, they require a payoff value per execution step to measure how well a policy is performing and to tune the learning process. However, in batch computing, the performance of a policy can only be calculated after the job has completed. Thus, these schemes do not explicitly address the issue of *delay* in getting feedback on how well a particular policy performed in executing jobs. Our online learning algorithm handles bounded delay and provides formal guarantees on its performance which scales with the amount of delay and the total number of jobs to be processed.

We evaluate our algorithms via simulations on a job trace from a datacenter cluster and Amazon EC2 spot market prices. We show that our approach outperforms greedy resource allocation heuristics in terms of total payoff – in particular, the average regret of our approach (compared to the best policy in hindsight) vanishes to zero with time. Further, it provides fast convergence while only using a small amount of training data. Finally, our algorithm enables interpreting the allocation strategy of the output policies, allowing users to apply them directly in practice.

2 Background and System Model

In this section we first provide a background on the online learning framework and then describe the problem setup and the parameterized set of policies for resource

allocation.

Regret-minimizing online learning. Our online learning framework is based on the substantial body of work on learning algorithms that make repeated decisions while aiming to minimize *regret*. The regret of an algorithm is defined as the difference between the cumulative performance of the sequence of its decisions and the cumulative performance of the best fixed decision in hindsight. We present only a brief overview of these algorithms due to space constraints.

In general, an online decision problem can be formulated as a repeated game between a learner (or decision maker) and the environment. The game proceeds in rounds. In each round j, the environment (possibly controlled by an adversary) assigns a reward $f_j(a)$ to each possible action a, which is not revealed beforehand to the learner. The learner then chooses one of the actions a_j, possibly in a randomized manner. The average payoff of an action a is the average of rewards $\frac{1}{J}\sum_{j=1}^{J} f_j(a)$ over the time horizon J, and the learner's average payoff is the average received reward $\frac{1}{J}\sum_{j=1}^{J} f_j(a_j)$ over the time horizon. The average regret of the learner is defined as $\max_a \frac{1}{J}\sum_{j=1}^{J} f_j(a) - \frac{1}{J}\sum_{i=1}^{J} f_j(a_j)$, namely the difference between the average payoff of the best action and the learner's sequence of actions. The goal of the learner is to minimize the average regret, and approach the average gain of the best action. Several learning algorithms have been proposed that approach zero average regret as the time horizon J approaches infinity, even against a fully adaptive adversary [8].

Our problem of allocating between on-demand and spot instances can be cast as a problem of repeated decision making in which the resource allocation algorithm must decide in a repeated fashion over which policies to use for meeting job due dates while minimizing job execution costs. However, our problem also differs from standard online learning, in that the payoff of each policy is not revealed immediately after it is chosen, but only after some delay (due to the time it takes to process a job). This requires us to develop a modified online algorithm and analysis.

Problem Setup. Our problem setup focuses on a single enterprise whose batch jobs arrive over time. Jobs may arrive at any point in time, however job arrival is monitored every fixed time interval of L minutes e.g., $L = 5$. For simplicity, we assume that each hour is evenly divided into a fixed number of such time intervals (namely, $60/L$). We refer to this fixed time interval as a *time slot* (or *slot*); the time slots are indexed by $t = 1, 2, \ldots$.

Jobs. Each job j is characterized by five parameters: (i) Arrival slot A_j: If job j arrives at time $\in [L(t'-1), Lt')$, then $A_j = t'$. (ii) Due date $d_j \in \mathbb{N}$ (measured in hours): If the job is not completed after d_j time units since its arrival A_j, it becomes invalid and further execution yields zero value. (iii) Job size z_j (measured in CPU instance hours to be executed): Note that for many batch jobs such as parameter sweep applications and software testing, z_j is known in advance. Otherwise, a small bounded over-estimate of z_j suffices. (iv) Parallelism constraint c_j: The maximal degree of parallelism i.e., the upper bound on number of instances that can be simultaneously assigned to the job. (v) Value function: $V_j : \mathbb{N} \to \mathbb{R}_+$, which is a monotonically non-increasing function with $V_j(\tau) = 0\ \forall \tau > d_j$.

Thus, job j is described by the tuple $\{A_j, d_j, z_j, c_j, V_j\}$. The job j is said to be *active* at time slot τ if less than d_j hours have passed since its arrival A_j, and the total instance hours assigned so far are less than z_j.

Allocation updates. Each job j is allocated computing instances during its execution. Given the existing cloud pricing model of charging based on hourly boundaries, the instance allocation of each active job is updated every hour. The i-th allocation update for job j is formally defined as a triplet of the form (o_j^i, s_j^i, b_j^i). o_j^i denotes the number of assigned on-demand instances; s_j^i denotes the number of assigned spot instances and b_j^i denotes their bid values. The parallelism constraint translates to $o_j^i + s_j^i \leq c_j$. Note that a NOP decision i.e., allocating zero resources to a job, is handled by setting o_j^i and s_j^i to zero.

Spot instances. The spot instances assigned to a job operate until the spot market price exceeds the bid price. However, as Figure 1 shows, the spot prices may change unpredictably implying that spot instances can get terminated at any time. Formally, consider some job j; let us normalize the hour interval to the closed interval $[0,1]$. Let $y_j^i \in [0,1]$ be the point in time in which the spot price exceeded the i-th bid for job j; formally, $y_j^i = \inf_{y\in[0,1]}\{p_s(y) > b_j^i\}$, where $p_s(\cdot)$ is the spot price, and $y_j^i \equiv 1$ if the spot price does not exceed the bid. Then the cost of utilizing spot instances for job j for its i-th allocation is given by $s_j^i * \hat{p}_j^i$, where $\hat{p}_j^i = \int_0^{y_j^i} p_j(y)dy$, and the total amount of work carried out for this job by spot instances is $s_j^i * y_j^i$ (with the exception of the time slot in which the job is completed, for which the total amount of work is smaller). Note that under spot pricing, the instance is charged for the full hour even if the job finishes earlier. However, if the instance is terminated due to market price exceeding the bid, the user is not charged for the last partial hour of execution. Further, we assume that the cloud platform provides advance notification of the instance revocation in this scenario.[1] Finally, as in

[1] [23] studies dynamic checkpointing strategies for scenarios where customers might incur substantial overheads due to out-of-bid situation. For simplicity, we do not model such scenarios in this paper. However, we note that the techniques developed in [23] are complementary, and can be applied in conjunction to our online learning

Amazon EC2, our model allows spot instances to be persistent, in the sense that the user's bid will keep being submitted after each instance termination, until the job gets completed or the user cancels it .

On-Demand instances. The price for an on-demand instance is fixed and is denoted by p (per-unit per time-interval). As above, the instance hour is paid entirely, even if the job finishes before the end of the hourly interval.

Utility. The utility for a user is defined as the difference between the overall value obtained from executing all its jobs and the total costs paid for their execution. Formally, let T_j be the number of hours for which job j is executed (actual duration is rounded up to the next hour). Note that if the job did not complete by its lifetime d_j, we set $T_j = d_j + 1$ and allocation $a_j^{T_j} = (0,0,0)$.

The utility for job j is given by:

$$U_j(a_j^1,\ldots,a_j^{T_j}) = V_j(T_j) - \sum_{i=1}^{T_j} \{\hat{p}_j^i s_j^i + p \cdot o_j^i\} \qquad (1)$$

The overall user utility is then simply the sum of job utilities: $U(\mathbf{a}) = \sum_j U_j(a_j^1,\ldots,a_j^{T_j})$. The objective of our online learning algorithm is to maximize the total user utility.

For simplicity, we restrict attention to *deadline value functions*, which are value functions of the form $V_j(i) = v_j$, for all $i \in [1,\ldots,d_j]$ and $V_j(i) = 0$ otherwise, i.e., completing job j by its due date has a fixed positive value [12]. Note that our learning approach can be easily extended to handle general value functions.

Remark. We make an implicit assumption that a user immediately gets the amount of instances it requests if the "price is right" (i.e., if it pays the required price for on-demand instances, or if its bid is higher than market price for spot instances. In practice, however, a user might exhibit delays in getting all the required instances, especially if it requires a large amount of simultaneous instances. While we could seamlessly incorporate such delays into our model and solution framework, we ignore this aspect here in order to keep the exposition simple.

Resource Allocation Policies. Our algorithmic framework allows defining a broad range of policies for allocating resources to jobs and the objective of our online learning algorithm is to approach the performance of the best policy in hindsight. We describe the parameterized set of policies in this section, and present the learning algorithm to adapt these policies, in detail in Section 3.

For each active job, a policy takes as input the job specification and (possibly) history of spot prices, and outputs an allocation. Formally, a policy π is a mapping of the form $\pi : \mathscr{J} \times \mathbb{R}_+ \times \mathbb{R}_+ \times \mathbb{R}_+^n \to \mathscr{A}$, which for every active job j at time τ takes as input:

framework.

(i) the job specification of j: $\{A_j, d_j, z_j, c_j, V_j\}$

(ii) the remaining work for the job z_j^τ

(iii) the total execution cost C_j incurred for j up to time τ (namely, $C_j^\tau \triangleq \sum_{t'=A_j}^{\tau-1} s_j^{t'} \hat{p}_j^{t'} + p \cdot o_j^{t'}$, and

(iv) a history sequence $p_s(\cdot)$ of past spot prices.

In return, the policy outputs an allocation.

As expected, the set of possible policies define an explosively large state space. In particular, we must carefully handle all possible instance types (spot, on-demand, both, or NOP), different spot bid prices, and their exponential number of combinations in all possible job execution states. Of course, no approach can do an exhaustive search of the policy state space in an efficient manner. Therefore, our framework follows a best-effort approach to tackle this problem by exploring as many policies as possible in the *practical operating range* e.g., a spot bid price close to zero has very low probability of being accepted; similarly, bidding is futile when the spot market price is above the on-demand price. We address this issue in detail in Section 3.

An elegant way to generate this practical set of policies is to describe them by a small number of *control parameters* so that any particular choice of parameters defines a single policy. We consider two basic families of parameterized policies, which represent different ways to incorporate the tradeoff between on-demand instances and spot-instances: (1) *Deadline-Centric*. This family of policies is parameterized by a deadline threshold M. If the job's deadline is more than M time units away, the job attempts allocating only spot-instances. Otherwise (i.e., deadline is getting closer), it uses only on-demand instances. Further, it rejects jobs if they become non-profitable (i.e., cost incurred exceeds utility value) or if it cannot finish on time (since deadline value function V_j will become zero). (2) *Rate-Centric*. This family of policies is parameterized by a fixed rate σ of allocating on-demand instances per round. In each round, the policy attempts to assign c_j instances to job j as follows: it requests $\sigma * c_j$ instances on-demand (for simplicity, we ignore rounding issues) at price p. It also requests $(1-\sigma) * c_j$ spot instances, using a bid price strategy which will be described shortly. The policy monitors the amount of job processed so far, and if there is a risk of not completing the job by its due date, it switches to on-demand only. As above, it rejects jobs if they become non-profitable or if it cannot finish on time. A pseudo-code implementing this intuition is presented in Algorithm 1. The pseudo-code for the deadline-centric family is similar and thus omitted for brevity.

We next describe two different methods to set the bids for the spot instances. Each of the policies above can

use each of the methods described below: (i) *Fixed bid*. A fixed bid value b is used throughout. (ii) *Variable bid*. The bid price is chosen adaptively based on past spot market prices (which makes sense as long as the prices are not too fluctuating and unpredictable). The variable bid method is parameterized by a weight γ and a safety parameter ε to handle small price variations. At each round, the bid price for spot instances is set as the weighted average of past spot prices (where the effective horizon is determined by the weight γ) plus ε. For brevity, we shall often use the terms *fixed-bid policies* or *variable-bid policies*, to indicate that a policy (either deadline-centric or rate-centric) uses the fixed-bid method or the variable-bid method, respectively. Observe that variable bid policies represent one simple alternative for exploiting the knowledge about past spot prices. The design of more "sophisticated" policies that utilize price history, such as policies that incorporate potential seasonality variation, is left as an interesting direction for future work.

ALGORITHM 1: Ratio-centric Policy

Parameters (with Fixed-Bid method): On-demand rate $\sigma \in [0,1]$; bid $b \in \mathbb{R}_+$
Parameters (with Variable-Bid method): On-demand rate $\sigma \in [0,1]$; weight $\gamma \in [0,1]$; safety parameter $\varepsilon \in \mathbb{R}_+$
Input: Job parameters $\{d_j, z_j, c_j, v_j\}$
If $c_j * d_j < z_j$ or $p * \sigma * z_j > v_j$, drop job //Job too large or expensive to handle profitably
for Time slot t in which the job is active **do**
  If job is done, return
  Let m be the number of remaining time slots till job deadline (including the current one)
  Let r be the remaining job size
  Let q be the cost incurred so far in treating the job
  // Check if more on-demand instances needed to ensure timely job completion
  if $(\sigma + m - 1)\min\{r, c_j\} < r$ **then**
    // Check if running job just with on-demand is still worthwhile
    if $p * r + q < v_j$ **then**
      Request $\min\{r, c_j\}$ on-demand instances
    else
      Drop job
    end if
  else
    Request $\sigma * \min\{r, c_j\}$ on-demand instances
    Request $(1-\sigma) * \min\{r, c_j\}$ spot instances at price:
    • **Fixed-Bid method:** Bid Price b
    • **Variable-Bid method:** $\frac{1}{Z}\int_y p_s(y)\gamma^{\tau - y}dy + \varepsilon$, where $Z = \int_y \gamma^{\tau - y}dy$ is normalization constant
  end if
end for

Note that these policy sets include, as special cases, some simple heuristics that are used in practice [3]; for example, heuristics that place a fixed bid or choose a bid at random according to some distribution (both with the option of switching to on-demand instances at some point). These heuristics (and similar others) can be implemented by fixing the weights given to the different policies (e.g., to implement a policy which selects the bid uniformly at random, set equal weights for policies that use the fixed-bid method and zero weights for the policies that use the variable-bid method). The learning approach which we describe below is naturally more flexible and powerful, as it *adapts* the weights of the different policies based on performance. More generally, we emphasize that our framework can certainly include additional families of parameterized policies, while our focus on the above two families is for simplicity and proof of concept. In addition, our learning approach can incorporate other parameters for resource allocation that are provided by cloud platforms e.g., VM instance type, datacenter/region. At the same time, some of these parameters may be set a priori based on user constraints e.g., an 'extra-large' instance may be fixed to accommodate large working sets of an application in memory, and a datacenter may be fixed due to application data stored in that location.

3 The Online Learning Algorithm

In this section we first give an overview of the algorithm, and then describe how the algorithm is derived and provide theoretical guarantees on its performance.

Algorithm Overview. The learning algorithm pseudocode is presented as Algorithm 2. The algorithm works by maintaining a distribution over the set of allocation policies (described in Section 2). When a job arrives, it picks a policy at random according to that distribution, and uses that policy to handle the job. After the job finishes execution, the performance of each policy on that job is evaluated, and its probability weight is modified in accordance with its performance. The update is such that high-performing policies (as measured by $f_j(\pi)$) are assigned a relatively higher weight than low-performing policies. The multiplicative form of the update ensures strong theoretical guarantees (as shown later) and practical performance. The rate of modification is controlled by a step-size parameter η_j, which slowly decays throughout the algorithm's run. Our algorithm also uses a parameter d defined as an upper bound on the number of jobs that arrive during any single job's execution. Intuitively, d is a measure of the delay incurred between choosing which policy to treat a given job, till we can evaluate its performance on that job. Thus, d is closely related to job lifetimes d_j defined in Section 2. Note that while d_j is measured in time units (e.g., hours), d measures the number of new jobs arriv-

ing during a given job's execution. We again emphasize that this delay is what sets our setting apart from standard online learning, where the feedback on each policy's performance is immediate, and necessitates a modified algorithm and analysis. The running time of the algorithm scales linearly with the number of policies and thus our framework can deal with (polynomially) large sets of policies. It should be mentioned that there exist online learning techniques which can efficiently handle exponentially large policy sets by taking the set structure into account (e.g. [8], Chapter 5). Incorporating these techniques here remains an interesting direction for future work.

We assume, without loss of generality, that the payoff for each job is bounded in the range $[0,1]$. If this does not hold, then one can simply feed the algorithm with normalized values of the payoffs $f_i(j)$. In practice, it is enough for the payoffs to be on the order of ± 1 on average for the algorithm to work well, as shown in our experiments in Section 4.

ALGORITHM 2: Online Learning Algorithm

Input: Set of n policies π parameterized by $\{1,\ldots,n\}$, upper bound d on jobs' lifetime
Initialize $\mathbf{w}_1 = (1/n, 1/n, \ldots, 1/n)$
for $j = 1,\ldots,J$ **do**
  Receive job j
  Pick policy π with probability $w_{j,\pi}$, and apply to job j
  if $j \leq d$ **then**
    $\mathbf{w}_{j+1} := \mathbf{w}_j$
  else
    $\eta_j := \sqrt{2\log(n)/d(j-d)}$
    for $\pi = 1,\ldots,n$ **do**
      Compute $f_j(\pi)$ to be the utility for job $j-d$, assuming we used policy π
      $w_{j+1,\pi} := w_{j,\pi} \exp\left(\eta_j f_j(\pi)\right)$
    end for
    for $\pi = 1,\ldots,n$ **do**
      $w_{j+1,\pi} := w_{j+1,\pi} / \sum_{r=1}^{n} w_{j+1,r}$
    end for
  end if
end for

Derivation of the Algorithm. Next we provide a formal derivation of the algorithm as well as theoretical guarantees. The setting of our learning framework can be abstracted as follows: we divide time into rounds such that round j starts when job j arrives. At each such round, we make some choice on how to deal with the arriving job. The choice is made by picking a policy π_j from a fixed set of n policies, which will be parameterized by $\{1,\ldots,n\}$. However, initially, we do not know the utility of our policy choice as future spot prices are unknown. We can eventually compute this utility in retrospect, but only after $\leq d$ rounds have elapsed and the relevant spot prices are revealed.

Let $f_j(\pi_{j-d})$ denote the utility function of the policy choice π_{j-d} made in round $j-d$. Note that according to our model, this function can be evaluated given the spot prices till round j. Thus, $\sum_{j=1+d}^{J+d} f_j(\pi_{j-d})$ is our total payoff from all the jobs we handled. We measure the algorithm's performance in terms of *average regret* with respect to any fixed choice in hindsight, i.e.,

$$\max_{\pi} \frac{1}{J} \sum_{j=1+d}^{J+d} f_j(\pi) - \frac{1}{J} \sum_{j=1+d}^{J+d} f_j(\pi_{j-d}).$$

Generally speaking, online learning algorithms attempt to minimize this regret, and ensure that as J increases the average regret converges to 0, hence the algorithm's performance converges to that of the single best policy in hindsight. A crucial advantage of online learning is that this can be attained without *any* statistical assumptions on the job characteristics or the price fluctuations.

When $d = 0$, this problem reduces to the standard setting of online learning, where we immediately obtain feedback on the chosen policy's performance. However, as discussed in Section 1, this setting does not apply here because the function f_j does not depend on the learner's current policy choice π_j, but rather on its choice at an earlier round, π_{j-d}. Hence, there is a delay between the algorithm's decision and feedback on the decision's outcome.

Our algorithm is based on the following randomized approach. The learner first picks an n-dimensional distribution vector $\mathbf{w}_1 = (1/n,\ldots,1/n)$, whose entries are indexed by the policies π. At every round j, the learner chooses a policy $\pi_j \in \{1,\ldots,n\}$ with probability w_{j,π_j}. If $j \leq d$, the learner lets $\mathbf{w}_{j+1} = \mathbf{w}_j$. Otherwise it updates the distribution according to

$$w_{j+1,\pi} = \frac{w_{j,\pi} \exp\left(\eta_j f_j(\pi)\right)}{\sum_{\pi=1}^{n} w_{j,i} \exp\left(\eta_j f_j(i)\right)},$$

where η_j is a step-size parameter. Again, this form of update puts more weight to higher-performing policies, as measured by $f_j(\pi)$.

Theoretical Guarantees. The following result quantifies the regret of the algorithm, as well as the (theoretically optimal) choice of the step-size parameter η_j. This theorem shows that the average regret of the algorithm scales with the jobs' lifetime bound d, and decays to zero with the number of jobs J. Specifically, as J increases, the performance of our algorithm converges to that of the best-performing policy in hindsight. This behavior is to be expected from a learning algorithm, and crucially, occurs without any statistical assumptions on the jobs characteristics or the price fluctuations. The performance also depends - but very weakly - on the size

n of our set of policies. From a machine learning perspective, the result shows that the multiplicative-update mechanism that we build upon can indeed be adapted to a delayed feedback setting, by adapting the step-size to the delay bound, thus retaining its simplicity and scalability.

Theorem 1 *Suppose (without loss of generality) that f_j for all $j = 1,\ldots,J$ is bounded in $[0,1]$. For the algorithm described above, suppose we pick $\eta_j = \sqrt{1\log(n)/2d(j-d)}$. Then for any $\delta \in (0,1)$, it holds with probability at least $1-\delta$ over the algorithm's randomness that*

$$\max_{\pi} \frac{1}{J}\sum_{j=1}^{J} f_j(\pi) - \frac{1}{J}\sum_{j=1}^{J} f_j(\pi_{j-d}) \;\leq\; 9\sqrt{\frac{2d\log(n/\delta)}{J}}.$$

The proof of the theorem is omitted here due to space constraints, and can be found in [18].

4 Evaluation

In this section we evaluate the performance of our learning algorithm via simulations on synthetic job data as well as a real dataset from a large batch computing cluster. The benefits of using synthetic datasets is that it allows the flexibility to evaluate our approach under a wide range of workloads. Before continuing, we would like to emphasize that the contribution of our paper is beyond the design of particular sets of policies - there are many other policies which can potentially be designed for our task. What we provide is a meta-algorithm which can work on any possible policy set, and in our experiments we intend to exemplify this on plausible policy sets which can be easily understood and interpreted.

Throughout this section, the parameters of the different policies are set such that the entire range of plausible policies is covered (with limitation of discretization). For example, the spot-price time series in Section 4.2 ranges between 0.12 and 0.68 (see Fig. 6(a)). Accordingly, we allow the fixed bids b to range between 0.15 and 0.7 with 5 cents resolution. Higher than 0.7 bids perform exactly as the 0.7 bid, hence can be excluded; bids of 0.1 or lower will always be rejected, hence can be excluded as well.

4.1 Simulations on Synthetic Data

Setup: For all the experiments on synthetic data, we use the following setup. Job arrivals are generated according to a Poisson distribution with mean 10 minutes; job size z_j (in instance-hours) is chosen uniformly and independently at random up to a maximum size of 100, and the parallelism constraint c_j was fixed at 20 instance-hours. Job values scale with the job size and the instance prices. More precisely, we generate the value as $x*p*z_j$, where x is a uniform random variable in $[0.5,2]$, and p is the on-demand price. Similarly, job deadlines also scale with size and are chosen to be $x*z_j/c_j$, where x is uniformly random on $[1,2]$. As discussed in Section 3, the on-demand and spot prices are normalized (divided by 10) to ensure that the average payoff per job is on the order of ± 1. The on-demand price is 0.25 per hour, while spot prices are updated every 5 minutes (the way we generate spot prices varies across experiments).

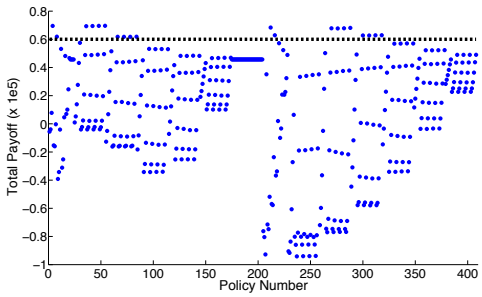


Figure 2: Total payoff for processing 20k jobs across each of the 408 resource allocation policies (while algorithm's payoff is shown as a dashed black line). The first 204 policies are rate-centric, and the last 204 policies are deadline-centric.

Resource allocation policies. We generate a parameterized set of policies. Specifically, we use 204 deadline-centric policies, and a same number of rate-centric policies. These policy set uses six values for M ($M \in \{0,\ldots,5\}$) and σ ($\sigma \in \{0,0.2,0.4,0.6,0.8,1\}$), respectively.

For either policy set, we have policies that use the fixed-bid method ($b \in \{0.1,0.15,0.2,0.25\}$), and policies that use the variable-bid method (weight $\gamma \in \{0,0.2,0.4,0.6,0.8\}$, and safety parameter $\varepsilon \in \{0,0.02,0.04,0.06,0.08,0.1\}$).

Simulation results: *Experiment 1*. In the first experiment, we compare the total payoff across 10k jobs of all the 408 policies to our algorithm. Spot prices are chosen independently and randomly as $0.15+0.05x$, where x is a standard Gaussian random variable (negative values were clipped to 0). The results presented below pertain to a single run of the algorithm, as they were virtually identical across independent runs. Figure 2 shows the total payoff for the 408 policies for this dataset. The first 204 policies are rate-centric policies, while the remaining 204 are deadline-centric policies. The performance of our algorithm is marked using dashed line. As can be seen, our algorithm performs close to the best policies in hindsight. Further, it is interesting to note that we have both deadline-centric and rate-centric policies among the best policies, indicating that one needs to consider both sets as candidate policies.

We perform three additional experiment with similar

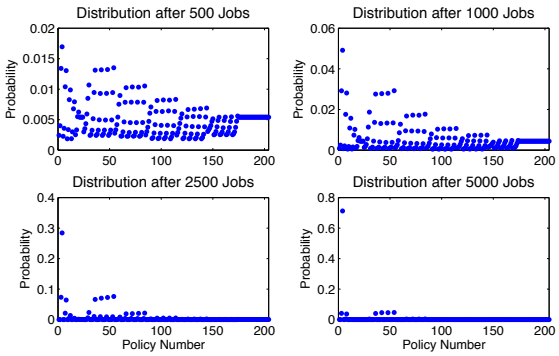


Figure 3: Evaluation under stationary spot-price distribution (mean spot price of 0.1): Probability assigned per policy after executing 500, 1000, 2500 and 5000 jobs.

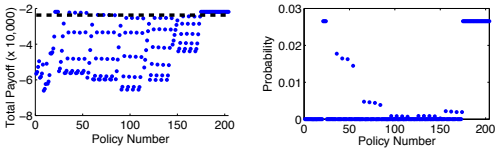


Figure 4: Evaluation under non-stationary distribution (mean spot price of 0.2): (a) Total payoff for executing 10k jobs across each of the 204 policies (while algorithm's payoff is shown as a dashed black line) and (b) the final probability assigned per policy by our learning algorithm.

setup to the above, in order to obtain insights on the properties and inner-working of the algorithm. To be able to dive deeper into the analysis, we use only the 204 rate-centric policies. The only element that we modify across experiments is the statistical properties of the spot-prices sequence.

Experiment 2. Spot prices are generated as above, except that we use 0.1 as their mean (opposed to 0.2 above). After executing 1000 jobs, our algorithm performs close to that of the best policy as it assigns probability close to 1 for that policy, while outperforming 199 out of total 204 policies. Further, its average regret is only 1.3 as opposed to 7.5 on average across all policies. Note that the upper bound on the delay in this experiment is $d = 66$, i.e., up to 66 jobs are being processed while a single job finishes execution. This shows that our approach can handle significant delay in getting feedback, while still performing close to the best policy.

In this experiment, the best policy in hindsight uses a fixed-bid of 0.25. This can be explained by considering the parameters of our simulation: since the on-demand price is 0.25 and the spot price is always relatively lower, a bid of 0.25 always yields allocation of spot instances for the entire hour. This result also highlights the easy interpretation of the resource allocation strategy of the best policy. Figure 3 shows the probability assignment for each policy over time by our algorithm after executing 500, 1000, 2500 and 5000 jobs. We observe that as the number of processed jobs increase, our algorithm provides performance close to the best policy in hindsight.

Experiment 3. In the next experiment, the spot prices is set as above for the first 10% of the jobs, and then the mean is increased to 0.2 (rather than 0.1) during the execution of the last 90% jobs. This setup corresponds to a non-stationary distribution: a learning algorithm which simply attempts to find the best policy at the beginning and stick to it, will be severely penalized when the dynamics of spot prices change. Figure 4 shows the evaluation results. We observe that our online algorithm is able to adapt to changing dynamics and converges to a probability weight distribution different from the previous setting; Overall, our algorithm attains an average regret of only 0.5, as opposed to 4.8 on average across 204 baseline policies.

Note that in this setting, the best policies are those which rely purely on on-demand instances instead of spot instances. This is expected because the spot prices tend to be only slightly lower than the on-demand price, and their dynamic volatility make them unattractive in comparison. This result demonstrates that there are indeed scenarios where the dilemma between choosing on-demand vs. spot instances is important and can significantly impact performance, and that no single instance type is always suitable.

Experiment 4. This time we set the spot price to alternate between 0.3 for one hour and then zero in the next. This variation is favorable for variable-bid policies with small γ, which use a small history of spot prices to determine their next bid. Such policies quickly adapt when the spot price drops. In contrast, fixed-bid policies and variable-bid policies with large γ suffer, as their bid price is not sufficiently adaptive. Figure 5 shows the results. We find that the group of highest-payoff policies are those for which $\gamma = 0$ i.e., they use the last spot price to choose a bid for the current round, and thus quickly adapt to changing spot prices. Further, our algorithm quickly detects and adapts to the best policies in this setting. The average regret obtained by our algorithm is 0.8 compared to 4.5 on average for our baseline policies. Moreover, the algorithm's overall performance is better than 192 out of 204 policies.

4.2 Evaluation on Real Datasets

Setup: *Workload data*. We use job traces from a large batch computing cluster for two days consisting of about 600 MapReduce jobs. Each MapReduce job comprises multiple phases of execution where the next phase can

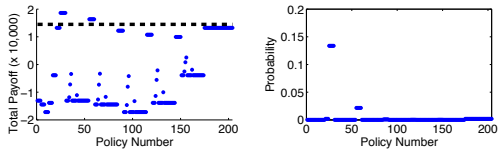


Figure 5: Evaluation under highly dynamic distribution (hourly spot prices alternate between 0.3 and zero): (a) Total payoff for processing 10k jobs across each of the 204 policies (algorithm's payoff is shown as a dashed black line), and (b) the final probability assigned per policy by our learning algorithm.

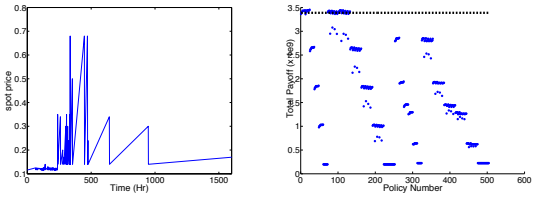


Figure 6: Evaluation on real dataset: (a) Amazon EC2 spot pricing data (subset of data from Figure 1) for Linux instances of type 'large'. The fixed on-demand price is 0.34; (b) Total payoff for processing 20k jobs across each of the 504 resource allocation policies (while algorithm's payoff is shown as a dashed black line)

start only after all tasks in the previous phase have completed. The trace includes the runtime of the job in server CPU hours ($totCPUHours$), the total number of servers allocated to it ($totServers$) and the maximum number of servers allocated to a job per phase ($maxServersPerPhase$). Since our job model differs from the MapReduce model in terms of phase dependency, we construct the parallelism constraint from the trace as follows: since the average running time of a server is $\frac{totCPUHours}{totServers}$, we set the parallelism bound c_j for each job to be $c_j = maxServersPerPhase * \frac{totCPUHours}{totServers}$. Note that this bound is in terms of CPU hours as required. Since the deadline values per job are not specified, we use the job completion time as its deadline. For assigning values per job, we generate them using the same approach as for synthetic datasets. Specifically, we assign a random value for each job j equal to its total size (in CPU hours) times the on-demand price times $B = (\alpha + N_j)$ where $\alpha = 5$ and $N_j \in [0,1]$ is drawn uniformly at random. The job trace is replicated to generate 20k jobs.

Spot Prices. We use a subset of the historical spot price from Amazon EC2 as shown in Figure 1 for 'large' Linux instances. Figure 6(a) shows the selected sample of spot price history showing significant price variation over time. Intuitively, we expect that overall that policies that use a large ratio of spot instances will perform better since on average, the spot price is about half of the on-demand price.

Resource Allocation Prices. We generated a total of 504 policies, half rate-centric and half deadline-centric. In each half, the first 72 are fixed-bid policies (i.e. policies that use the fixed-bid method) in increasing order of (on-demand rate, bid price). The remaining 180 variable-bid policies are in increasing order of (on-demand rate, weight, safety parameter). The possible values for the different parameters are as described for the synthetic data experiments, with the exception that we allow more options for the fixed bid price, $b \in \{0.15,0.2,0.25,0.3,0.35,0.4,0.45,0.5,0.55,0.6,0.65,0.7\}$.

Evaluating our online algorithm on the real trace poses several new challenges compared to the synthetic datasets in Section 4.1. First, jobs sizes and hence their values are highly variable, to the effect that the difference in size between *small* and *large* jobs can be of six orders of magnitude. Second, spot prices can exhibit high variability, or alternatively be almost stable towards the end as exemplified in Figure 6(a).

Simulation results: Figure 6(b) shows the results for a typical run of this experiment. Notably, the payoff of our algorithm outperforms the performance of most of individual policies, and obtains comparable performance to the best individual policies (which are a subset of the rate-centric policies). We repeated the experiment 20 times, and obtained the following results: The average regret per job for our learning algorithm is 2071 ± 1143, while the average regret across policies is 70654 ± 12473. Note that the average regret of our algorithm is around 34 times better (on average) than the average regret across policies.

Figure 7 shows the evolution of policy weights over time for a typical run, until converging to final policy weights (after handling the entire 20000 jobs). We observe that our algorithm evolves from preferring a relatively large subset of both deadline-centric and rate-centric policies (at around 150 jobs) to preferring only rate-centric policies, both fixed-bid and variable-bid (at around 2000 jobs). Eventually, the algorithm converges to a single rate-centric policy with fixed bid. This behavior can be explained based on spot pricing data in Figure 6(a): Due to initially high variability in spot prices, our algorithm "alternates" between fixed-bid policies and variable-bid policies, which try to learn from past prices. However, since the prices show little variability for the remaining two thirds of the data, the algorithm progressively adapts its weight for the fixed-bid policy, which is commensurate with the almost stable pricing curve.

5 Related literature

While there exist other potential approaches to our problem, we considered an online learning approach due to its

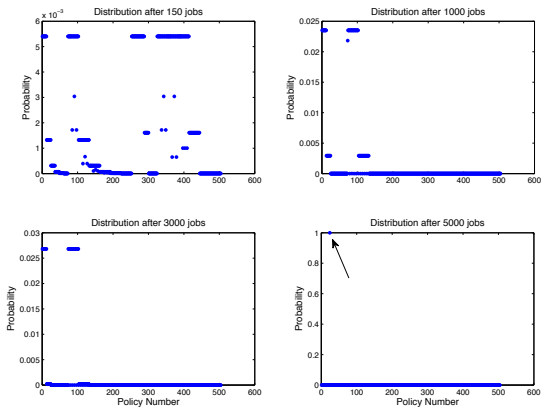


Figure 7: Evaluation on real dataset: The probability assigned per policy by our learning algorithm after processing 150, 1000, 3000 and 5000 jobs. The algorithm converges to a single policy (fixed-bid rate-centric policy) marked by an arrow.

lack of any stochastic assumptions, its online (rather than offline) nature, its capability to work on arbitrary policy sets, and its ability to adapt to delayed feedback. The idea of applying online learning algorithms for sequential decision-making tasks is well known ([9]), and there are quite a few papers which study various engineering applications (e.g., [10, 5, 11, 15]). However, these efforts do not deal with the problem of delayed feedback as it violates the standard framework of online learning. The issue of delay has been previously considered (see [14] and references therein), but are either not in the context of the online techniques we are using, or propose less-practical solutions such as running many multiple copies of the algorithm in parallel. In any case, we are not aware of any prior study of delay-tolerant online learning procedures for our application domain.

The launch of commercial cloud computing offerings has motivated the systems research community to investigate how to exploit this market for efficient resource allocation and cost reductions. Some solution concepts are borrowed from earlier works on executing jobs in multiple grids (e.g., [20] and references therein). However, new techniques are required in the cloud computing context, which directly incorporate cost considerations and a variety of instance renting options. The have been numerous works in this context dealing with different provider and customers scenarios. One branch of papers consider the auto-scaling problem, where an application owner has to decide on the right number and type of VMs to be purchased, and dynamically adapt resources as a function of changing workload conditions (see, e.g., [17, 6] and references therein).

We focus the reminder of our literature survey on cloud resource management papers that include spot instances as one of the allocation options. Some papers focus on building statistical models for spot prices which can be then used to decide when to purchase EC2 spot instances (see, e.g., [13, 1]). Similarly, [24] examines the statistical properties of customer workload with the objective of helping the cloud determine how much resources to allocate for spot instances.

In the context of large-scale batch applications, [4] proposes a probabilistic model for bidding in spot prices while taking into account job termination probabilities. However, [4] focuses on pre-computation of a fixed (non-adaptive) bid, which is determined greedily based on existing market conditions; moreover, the suggested framework does not support an automatic selection between on-demand and spot instances. [22] uses a genetic algorithm to quickly approximate the pareto-set of makespan and cost for a bag of tasks; each underlying resource configuration consists of a different mix of on-demand and spot instances. The setting in [22] is fundamentally different than ours, since [22] optimizes a global makespan objective, while we assume that jobs have individual deadlines. Finally, [21] proposes near-optimal bidding strategies for cloud service brokers that utilize the spot instance market to reduce the computational cost while maximizing the profit. Our work differs from [21] in two main aspects. First, unlike [21], our online learning framework does not require any distributional assumptions on the spot price evolution (or the job model). Second, our model may associate a different *value* and *deadline* for each job, whereas in [21] the value is only a function of job size, and deadlines are not explicitly treated.

6 Conclusion

In this paper we design and evaluate an online learning algorithm for automated and adaptive resource allocation for executing batch jobs over cloud computing platforms. Our basic model can be extended to solve other resource allocation problems in cloud domains such as renting small vs. medium vs. large instances, choosing computing regions, and different bundling options in terms of CPU, memory, network and storage. We expect that the learning framework developed here would be useful in addressing these extensions. An interesting direction for future research is incorporating reserved instances, for long-term handling of multiple jobs. This makes the algorithm stateful, in the sense that its actions affect the payoffs of policies chosen in the future. This does not accord with our current theoretical framework, but may be handled using different tools from competitive analysis.

Acknowledgements. We thank our shepherd Alexandru Iosup and the ICAC reviewers for the useful feedback.

References

[1] O. Agmon Ben-Yehuda, M. Ben-Yehuda, A. Schuster, and D. Tsafrir. Deconstructing Amazon EC2 spot instance pricing. *ACM Transactions on Economics and Computation*, 1(3):16, 2013.

[2] M. Alizadeh, A. Greenberg, D. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center TCP (DCTCP). In *ACM SIGCOMM Computer Communication Review*, volume 40, pages 63–74. ACM, 2010.

[3] AWS Case Studies. `http://aws.amazon.com/solutions/case-studies/`.

[4] A. Andrzejak, D. Kondo, and S. Yi. Decision model for cloud computing under sla constraints. In *MASCOTS*, 2010.

[5] I. Ari, A. Amer, R. Gramacy, E. Miller, S. Brandt, and D. Long. Acme: Adaptive caching using multiple experts. In *WDAS*, 2002.

[6] Y. Azar, N. Ben-Aroya, N. R. Devanur, and N. Jain. Cloud scheduling with setup cost. In *Proceedings of the 25th ACM symposium on Parallelism in algorithms and architectures*, pages 298–304. ACM, 2013.

[7] Browsermob. `https://aws.amazon.com/solutions/case-studies/Browsermob`.

[8] N. Cesa-Bianchi and G. Lugosi. *Prediction, learning, and games*. Cambridge University Press, 2006.

[9] Y. Freund and R. E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *J. Comput. Syst. Sci.*, 55(1):119–139, 1997.

[10] R. Gramacy, M. Warmuth, S. Brandt, and I. Ari. Adaptive caching by refetching. In *NIPS*, 2002.

[11] D. Helmbold, D. Long, T. Sconyers, and B. Sherrod. Adaptive disk spin-down for mobile computers. *MONET*, 5(4):285–297, 2000.

[12] N. Jain, I. Menache, J. Naor, and J. Yaniv. Near-optimal scheduling mechanisms for deadline-sensitive jobs in large computing clusters. In *SPAA*, pages 255–266, 2012.

[13] B. Javadi, R. Thulasiram, and R. Buyya. Statistical modeling of spot instance prices in public cloud environments. In *Fourth IEEE International conference on Utility and Cloud Computing*, 2011.

[14] P. Joulani, A. György, and C. Szepesvári. Online learning under delayed feedback. In *ICML*, 2013.

[15] B. Kveton, J. Y. Yu, G. Theocharous, and S. Mannor. Online learning with expert advice and finite-horizon constraints. In *AAAI*, 2008.

[16] Litmus. `https://aws.amazon.com/solutions/case-studies/Litmus`.

[17] M. Mao and M. Humphrey. Auto-scaling to minimize cost and meet application deadlines in cloud workflows. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 49. ACM, 2011.

[18] I. Menache, O. Shamir, and N. Jain. On-demand, spot, or both: Dynamic resource allocation for executing batch jobs in the cloud. Technical report, Microsoft Research, May 2014. Available from `http://research.microsoft.com/apps/pubs/default.aspx?id=217154`.

[19] S. Shen, K. Deng, A. Iosup, and D. Epema. Scheduling jobs in the cloud using on-demand and reserved instances. In *Euro-Par 2013 Parallel Processing*, pages 242–254. Springer, 2013.

[20] M. Silberstein, A. Sharov, D. Geiger, and A. Schuster. Gridbot: execution of bags of tasks in multiple grids. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, page 11. ACM, 2009.

[21] Y. Song, M. Zafer, and K.-W. Lee. Optimal bidding in spot instance market. In *INFOCOM*, pages 190–198, 2012.

[22] A. Vintila, A.-M. Oprescu, and T. Kielmann. Fast (re-)configuration of mixed on-demand and spot instance pools for high-throughput computing. In *Proceedings of the first ACM workshop on Optimization techniques for resources management in clouds*, pages 25–32. ACM, 2013.

[23] S. Yi, A. Andrzejak, and D. Kondo. Monetary cost-aware checkpointing and migration on Amazon cloud spot instances. *IEEE Transactions on Services Computing*, 5(4):512–524, 2012.

[24] Q. Zhang, E. Gürses, R. Boutaba, and J. Xiao. Dynamic resource allocation for spot markets in clouds. In *Hot-ICE*, 2011.

Real-time Scheduling of Skewed MapReduce Jobs in Heterogeneous Environments

Nikos Zacheilas and Vana Kalogeraki
Athens University of Economics and Business
Athens, Greece
zaheilas@aueb.gr, vana@aueb.gr

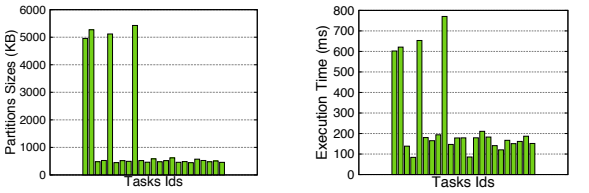
Abstract

Supporting real-time jobs on MapReduce systems is particularly challenging due to the heterogeneity of the environment, the load imbalance caused by skewed data blocks, as well as real-time response demands imposed by the applications. In this paper we describe our approach for scheduling real-time, skewed MapReduce jobs in heterogeneous systems. Our approach comprises the following components: (i) a distributed scheduling algorithm for scheduling real-time MapReduce jobs end-to-end, and (ii) techniques for handling the data skewness that frequently arises in MapReduce environments and can lead to significant load imbalances. Our detailed experimental results using real datasets on a truly heterogeneous environment, Planetlab, illustrate that our approach is practical, exhibits good performance and consistently outperforms its competitors.

1 Introduction

Today, we are experiencing increased demand for processing large amounts of data-intensive tasks. Systems such as IBM's InfoSphere BigInsights [20], Amazon's DynamoDB [2] and Google's MapReduce [10], have rapidly become de facto big data processing frameworks. These systems need to be fast, scalable and highly available. In particular, Google's MapReduce [10] framework has been proposed as a powerful and cost-effective approach for massive-scale processing. It has been utilized by some of the major computing companies, including Amazon, eBay, Facebook, IBM, LinkedIn, Twitter and Yahoo!, via its open-source implementation Hadoop [17] in a wide variety of application domains including real-time analysis of sensor data streams, real-time stock market data analysis and financial trading applications.

The MapReduce model breaks intense processing jobs into smaller tasks that run in parallel on multiple machines. Jobs are split into two stages of processing, *map* and *reduce*: input data are processed by tasks comprising the *map* phase, generating intermediate *(key, value)* pairs. Different *values* corresponding to the same *key* are then aggregated, and each *key* along with its associated *values* are transmitted to the *reduce* phase for further data processing. The partitioning of the intermediate $(key, value)$ pairs to the *reduce* tasks is based on a partitioning function, which in most cases is a simple hash function. However, *partitions* (*i.e.,* the set of intermediate $(key, value)$ pairs that will be processed by the same *reduce* task) can be of varying size, leading to significant data skewness challenges and critical delays on the execution times of the corresponding *reduce* tasks.



(a) Partition Sizes Processed Per Reduce Task

(b) Execution Time Per Reduce Task

Figure 1: Impact of skewed partitions on reduce tasks' execution times

We use the following example to illustrate the load imbalances that can occur due to skewness (*i.e., partitions* of varying sizes) as well as the effect on the execution times of the tasks, when processing skewed data-intensive MapReduce tasks. Figure 1 illustrates the distribution of the *partitions*' sizes and the corresponding execution times of the *reduce* tasks for a MapReduce job that processes a Youtube social graph (detailed information about the experiment can be found in the experimental evaluation section). The experiment run on Planetlab, using 82 processing cores and applying the hash function $(hashcode(key) mod R$, where R is the number of reduce tasks) proposed in the original MapReduce framework [10]. The figure clearly depicts that the exhibited skewness of the *partition* sizes affects the execution time of *reduce* tasks due to the uneven distribution of the data.

The problem is further exacerbated by the fact that jobs often have real-time response requirements, in the form of *deadlines*. As was pointed out by a recent study on Facebook's and Yahoo!'s production workload traces, 95% percentage of their production jobs are short running with an average real-time response requirement of 30 seconds [6], [30]. Timely execution of the tasks is a challenging problem due to the large heterogeneity and resource sharing in the nodes; in a shared processing environment, the execution times of the jobs are greatly affected by multiple tasks invoked concurrently and asynchronously by many jobs that are executed on the same computing resources. Conditions such as slow or misbehaving tasks (*i.e.,* due to hardware failures or misconfiguration), can severely affect the performance of the entire jobs [10]. Nevertheless, to make the deployment of these systems practical, the jobs must be able to operate autonomously in highly dynamic environments, and meet real-time demands, even under load spikes and unpredictable initiation of new tasks.

Current scheduling techniques in the Hadoop MapReduce framework (the most widely used MapReduce implementation) are not adequate, as they either adopt Fair scheduling [19] or Capacity scheduling [18]. These strive to balance the load across the resources rather than meeting real-time demands of the jobs. Recent approaches for scheduling MapReduce jobs include the LATE [42] and EDF [39] schedulers, however both approaches focus on scheduling and do not examine the impact of skewed data on the execution time of jobs. With respect to the skewed *partitions* problem, work has been mainly done by [14] and [27]. The first aims at distributing similar sized *partitions* among the available *reduce* tasks, but in heterogeneous environments this approach can lead to the assignment of large-size *partitions* to slow nodes. Skewtune [27] on the other side, proposes the repartitioning of heavily skewed *partitions*, however the overhead of such schemes can degrade the performance of *short* running jobs. It is clear that none of the existing works offers a unified solution for the two problems but rather examines them separately without taking into account their interaction.

In this work, we present DynamicShare, our system for supporting the execution of real-time, skewed MapReduce jobs in heterogeneous environments. Our goal is to address the joint problem of: (a) scheduling MapReduce jobs dynamically to maximize the probability of meeting their real-time response requirements, and (b) effectively handle the issue of data skewness. We focus on the execution of short running jobs, similar to Facebook Corona [37], (typical queries are: *identify common friends in Facebook*), which execute in the order of seconds. To our knowledge this is the first proposal towards a unified solution to the stated problem. Our approach makes the following contributions:

1. We present a distributed scheduling algorithm for scheduling jobs end-to-end. DynamicShare uses measurements of *laxity* values of the tasks, projected latencies and measurements of resource loads to adjust their scheduling order to compensate for queuing delays, estimates the execution times of the tasks using a non-parametric regression technique and identifies overloaded nodes early on through a Local Outlier Factor algorithm.

2. To handle the data skewness that arises in the *reduce* phase, we design two algorithms, a simple but efficient *Simple Partitions*' assignment algorithm that considers the sizes of the *partitions* and the variable processing capabilities of the nodes to make an appropriate placement, and a *Count-Min sketches* algorithm that enables an even better distribution of the *partitions* but at the expense of additional execution time for the *partitions*' assignment procedure.

3. We have implemented and evaluated DynamicShare on Planetlab, a truly heterogeneous environment, using 82 processing cores in total. Our experimental results utilizing two different datasets, from Youtube and Twitter networks, illustrate that our approach is practical, meets jobs' real-time response demands, effectively addresses the issue of highly skewed data, and outperforms its competitors.

2 System Model and Architecture

2.1 System Model

A MapReduce job is modelled as a sequence of invocations of *M map* and *R reduce* tasks (shown in Figure 2). Tasks are modelled as follows: $map(k_1, v_1) \Rightarrow [k_2, v_2]$ and $reduce(k_2, [v_2]) \Rightarrow [k_3, v_3]$. *Map* tasks take as input (k_1, v_1) pairs and return a list of *(key,value)* pairs of possibly different types, k_2 and v_2. The values associated with the same key k_2 are grouped together into a list and passed as input to the appropriate *reduce* task, which emits arbitrary *(key,value)* pairs of a final type, k_3 and v_3. All $(k_2, [v_2])$ pairs processed by the same *reduce* task on a cluster's node, are considered a *partition*. Recent works, such as [14], suggest the usage of a larger number of *partitions* compared to the number of *reduce* tasks to minimize the skewness of the intermediate data. Our framework follows this approach.

We consider soft real-time MapReduce jobs that are aperiodic and, thus, their arrival times are not known a priori. We focus on applications with intensive reduce phase and limited network transfers. Each job j is associated with a number of parameters: $Deadline_j$ is the

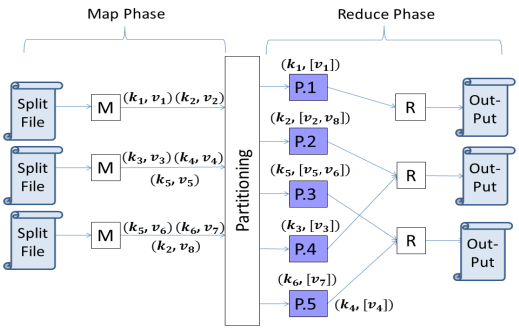


Figure 2: MapReduce Computation Model

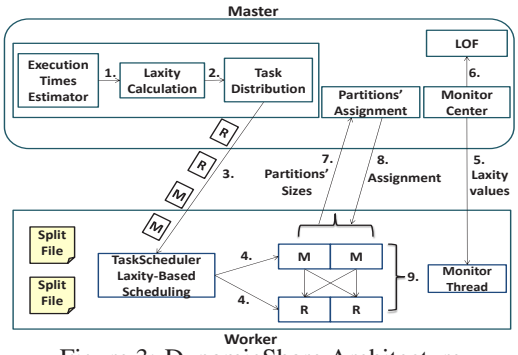


Figure 3: DynamicShare Architecture

time interval, starting at job initiation, within which job j must complete (deadline values are typically assigned by a system administrator based on the real-time demands of the jobs, determining the appropriate deadline for the job is outside the scope of the paper). $Proj\_exec\_time_j$ is the estimated amount of time required for the job to complete, this includes communication and queueing times at the system resources. Every job j should execute within its $Deadline_j$, that is, the sum of the computation times and the corresponding communication times of all tasks invoked by the job (denoted as projected end-to-end execution time) should be smaller than the $Deadline_j$. MapReduce jobs are data-intensive jobs, thus, the end-to-end execution time is mainly attributed to the execution times of the tasks and the communication times are negligible. $Laxity_j$ is defined as the difference between the $Deadline_j$ and $Proj\_exec\_time_j$, and is considered as a measure of urgency for the j job. The *laxity* value for each job is updated dynamically during execution; and determines the order with which the job's tasks will be scheduled at the system resources. Finally, $split\_size_j$ is the user-defined size of a split input file for job j.

Each task t of job j is described with the following metrics: $cpu_{i,t}$ and $memory_{i,t}$ represent the average percentage of CPU and memory required for task t to execute on Worker i. $m_{i,t}$ is the estimated mean execution time of a *map* task on Worker i. This includes the required time to read the total amount of input data, execute the *map* method and transmit the intermediate pairs to the *reduce* tasks. Similarly, $r_{i,t}$ depicts the estimated execution time of a *reduce* task, this corresponds to the time required for grouping $(key, value)$ pairs with the same *key* into a single (*key*,*list_of_values*) pair, plus the required time for executing the *reduce* method. Finally $partitions\_size_{i,t}$ holds the total size of the *partitions* that are assigned on Worker i for a *reduce* task t.

2.2 DynamicShare Architecture

The DynamicShare architecture (shown in Figure 3) comprises a single Master node and multiple Worker nodes. The Master receives MapReduce jobs, along with their corresponding deadline requirements. It is responsible for the assignment of the *map* and *reduce* tasks to Workers and the monitoring of the currently executing jobs. For each submitted job, the Master estimates the execution times of the *map* and *reduce* tasks in order to compute the execution time of the whole job and its corresponding *laxity* value. This value will be used by the TaskScheduler component at each Worker when scheduling the job's tasks (Section III). The Master is responsible to keep track of current resource usage statistics (*i.e.,* CPU, memory) along with their respective *laxity* values, as reported by the Monitor components at the individual Worker nodes during task execution. This information will also be used by the anomaly detection algorithm to identify overloaded nodes. The Master invokes the *partitions*' assignment algorithm to decide how to distribute the generated *partitions* to the available *reduce* tasks (Section IV). Finally, the input data file for each job is uploaded to the cluster and distributed as equal sized split files to the Workers for processing by the *map* tasks; the size of the split file is typically user-defined. Techniques such as the one proposed in [22] can further enhance our framework to distribute the split files based on the processing capabilities of the nodes in the cluster.

3 Dynamic Real-Time Scheduling of MapReduce Jobs

We develop a distributed scheduling approach that dynamically adjusts the execution of jobs on the system resources and measures the impact of overloaded nodes on meeting their end-to-end real-time demands. It consists of the following components: (a) A model for estimating the execution times of tasks based on a commonly used *non-parametric* regression technique, *k-Nearest Neighbor (k-NN) smoothing*. (b) A distributed least laxity first scheduling algorithm for scheduling jobs end-to-end, that uses measurements of the *laxity* values of the tasks to adjust their scheduling order to compensate for queuing delays at Workers. (c) Early detection of overloaded nodes via a Local Outlier Factor algorithm.

3.1 Estimating Execution Times of Tasks

To estimate the execution times of the entire jobs, the fundamental idea is to compute an approximation of the execution times of the *map* and *reduce* tasks and use this approximation in the computation of the execution time of the entire job. Techniques for estimating the execution time of MapReduce jobs, such as building job profiles based on previous execution times [38] or using debug runs before the actual execution [29] have been proposed for homogeneous environments, but without examining the implications of the problem in a heterogeneous setting, where the execution times of the tasks may vary.

We propose an estimation model that considers both the resource requirements of the newly submitted tasks and the previous task runs. There are two main approaches to address this: with *parametric* or *non-parametric* techniques [40]. For each *map* task we maintain a vector $\overrightarrow{x}$ of the task parameters as follows: $\overrightarrow{x} = (split\_size_j, cpu_{i,t}, memory_{i,t})$ Similarly, for *reduce* tasks we have: $\overrightarrow{x} = (partitions\_size_{i,t}, cpu_{i,t}, memory_{i,t})$
The $cpu_{i,t}, memory_{i,t}$ requirements of a newly submitted task are estimated via a histogram based approach similar to [25]. This approach is based on past runs. It distributes the processing requirements of previous tasks into histogram bins and utilizes the mean of the most populated bin for estimating the requirements of the newly issued task. If we model the execution time using *parametric* regression, the functional form of the $m(\overrightarrow{x})$ is assumed known and a technique like Least Squares can be applied for the calculation of the polynomial coefficients. However, in the case of a highly dynamic environment, like our setting, computing the execution time of tasks via a polynomial function is not efficient [21]. Thus, in such environments *non-parametric* techniques are more appropriate.

In *non-parametric* regression no assumption can be made about the functional form of $m(\overrightarrow{x})$, therefore the estimation is regarded as *data driven* because it depends only on previous task runs. All *non-parametric* regression techniques are modelled by the following equation:

$$\hat{m}(\overrightarrow{x}) = \frac{1}{n}\sum_{i=1}^{n} W_i(\overrightarrow{x}) y_i \tag{1}$$

where $W_i(\overrightarrow{x})$ is a weighting sequence and y_i the execution time of a previously issued task. Essentially, the $\hat{m}(\overrightarrow{x})$ can be considered as the weighted average of n previous task runs. Special care must be given to the number of previous runs that will be used. Too many past runs can lead to overly biased results, on the other hand few examples make the curve too "noisy". In our previous works [4], [23] we have shown that the number of runs to use depends on the job's characteristics.

The *non-parametric* regression technique we decided to implement for our estimation problem was *k-Nearest Neighbor (k-NN) smoothing*. In *k-NN smoothing*, the estimation of $\hat{m}(\overrightarrow{x})$ is based on the k past runs that are closest to the given vector $\overrightarrow{x}$. Utilizing a subset of the past runs, instead of all n past runs, is important because only those that have similar resource requirements with the currently examining task are considered in the estimation, and thus a better prediction is possible. We use the Euclidean distance of vectors to identify the closest past runs. In order to achieve better estimations, the impact of the previous runs on the estimation is weighted based on their distance from the examining vector, with the ones being closer receiving higher weights. A weighting function with several optimality properties is the Epanechnikov kernel $K(d)$, where: $K(d) = \frac{3}{4}(1-d^2)$, with $|d| < 1$. The d parameter, in our case is the calculated Euclidean distance of the vectors. The choice of the kernel function is not significant for the results of the approximation [33]. The Epanechnikov kernel function gives more weight to previous runs that are closer to the examining task's vector parameters. In order to be utilized by the estimator, the function must be scaled and normalized. We used the following weighting function:

$$W_i(\overrightarrow{x}) = \begin{cases} \frac{K_R(|\overrightarrow{x}-\overrightarrow{x_i}|)}{\hat{f}(\overrightarrow{x})} & \text{if } i \in N_{\overrightarrow{x}} \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

where $N_{\overrightarrow{x}} = \{i\text{: } \overrightarrow{x_i}$ is one of the k nearest neighbors of $\overrightarrow{x}\}$, $K_R(d)$ is the scaled Epanechnikov kernel function and is given by the following equation: $K_R(d) = \frac{1}{R}K(\frac{d}{R})$. The kernel is scaled by the factor R which is defined as: $R = max(|\overrightarrow{x} - \overrightarrow{x_i}|)$, $i \in N_{\overrightarrow{x}}$. Finally the $\hat{f}(\overrightarrow{x})$ factor in equation (2) is a normalized factor, and is given by the following formula: $\hat{f}(\overrightarrow{x}) = \frac{1}{k}\sum_{\overrightarrow{x_i} \in N_{\overrightarrow{x}}} K_R(|\overrightarrow{x} - \overrightarrow{x_i}|)$.

One important aspect of the algorithm is the choice of the value for k that will determine the number of past runs to be taken into account during the estimation. Too many examples can cause an increase on the bias $E\{\hat{m}(\overrightarrow{x}) - m(\overrightarrow{x})\}$, while few previous runs may lead to large variance $E\{\hat{m}^2(\overrightarrow{x})\}$. The value of k depends on the number n, so it must be adjusted accordingly. It has been proven [40] that by increasing k in proportion to $n^{\frac{4}{5}}$, the *k-NN* technique achieves a constant balance between the variance and the bias.

3.2 Least Laxity First Scheduling

We develop a dynamic, distributed least laxity first scheduling scheme that determines the order of execution of the tasks based on their urgencies and timing constraints. The Least Laxity First Scheduling (LLF) algorithm has been successfully employed in distributed and mobile real-time systems such as in [12], [24]. In LLF each job is associated with a *laxity* value, that represents a measure of urgency for the job and is used to order the execution of the tasks on the Workers. Given the dead-

line $Deadline_j$ and $Proj\_exec\_time_j$ for job j, we compute $Laxity_j$ as:

$$Laxity_j = Deadline_j - Proj\_exec\_time_j, \quad (3)$$

The projected execution time of j job is computed based on its estimates of the execution times of the *map* and *reduce* tasks, as follows: $Proj\_exec\_time_j = max\{m_{i,t}, ..., m_{k,t}\} + max\{r_{z,t}, ..., r_{l,t}\}$, where we consider the maximum execution times of the *map* and *reduce* tasks in the above computation since all tasks of the same phase run in parallel. The *laxity* value for each job is computed initially by the Master node and is used when scheduling the tasks of the job at each node. The closer the $Laxity_j$ value to zero, the more probable is for the job to miss its deadline; a negative value indicates that the deadline will be missed.

Tasks are ordered in the TaskScheduler at each node, based on the *laxity* values of the jobs that invoke them. The advantage of LLF compared to other scheduling approaches such as Earliest Deadline First (EDF) [39] and Hadoop's FIFO and FAIR scheduler, is, that, LLF is a dynamic scheduling algorithm that allows for compensating for queueing delays often experienced in distributed settings or that were mis-calculated at previous nodes. In LLF the task with the smallest laxity value has the higher priority. The *laxity* value of a job is adjusted as the tasks invoked by the job execute on the system resources. To avoid constantly updating the *laxity* values for a job, they are adjusted only when new tasks are inserted into the TaskScheduler's queue or when tasks finish execution or miss their deadlines. As the *laxity* values of the jobs are updated, the task belonging to the job with the smallest *laxity* value will execute next; if a job has negative *laxity* value this job has been estimated to miss its deadline and thus its tasks will be processed only when the TaskScheduler has pending tasks that all have missed their deadlines. That means a task with negative *laxity* will never preempt tasks with positive *laxity* values.

3.3 Identifying Overloaded Nodes

Because nodes' resource capacities are not directly correlated to the amount of data they are assigned for processing, it is possible that the data blocks are distributed unproportionally to the nodes. This may result in Workers becoming overloaded and not capable of completing their assigned tasks within the jobs' deadlines.

To identify overloaded Workers early on, we use the Local Outlier Factor (LOF) algorithm [5] on the *laxity* values of tasks of the same job that run on different Workers. LOF is a metric of anomaly detection that can be applied to a set and identify possible outliers. We consider as outliers, *laxity* values that significantly differ from the rest. Our goal is to proactively identify overloaded Workers before the *laxity* value of the corresponding job becomes negative.

The main idea of the LOF algorithm is to compare the local density of a point's neighbourhood with respect to the local density of its neighbours, seeking one or more points with significant difference from the rest. Thus, we compare the *laxity* values for the same job on the different Workers that execute the tasks of the job. More formally: Let $l\_distance(lax_A)$ be the distance of a lax_A to the l nearest neighbour, lax_A will be the *laxity* value of a job j that runs on a Worker that can possibly be overloaded. We denote the l nearest *laxity* values as $N_l(lax_A)$. This distance is used to define the *reachability distance* metric: $reach\_d_l(lax_A, lax_B) := max\{l\_distance(lax_B), d(lax_A, lax_B)\}$, where lax_B will be the *laxity* value of the same j job on a different Worker. The *reachability distance* of a value lax_A from lax_B depicts the true distance of the two values ($d(lax_A, lax_B)$), but also at least the $l\_distance$ of lax_B. The usage of this distance achieves more stable results as shown on [5]. The *local reachability density* of lax_A is defined by:

$$lrd_l(lax_A) := \frac{|N_l(lax_A)|}{\sum_{lax_B \in N_l(lax_A)} reach\_d_l(lax_A, lax_B)} \quad (4)$$

and intuitively is the inverse of the mean *reachability distance* of lax_A from its neighbouring *laxity* values. The local *reachability density* is then compared with those of the neighbours using the following equation:

$$LOF_l(lax_A) := \frac{\sum_{lax_B \in N_l(lax_A)} lrd_l(lax_B)}{|N_l(lax_A)| * lrd_l(lax_A)} \quad (5)$$

which is the average *reachability density* of the neighbours divided by lax_A's own local density. A *LOF* value below 1 indicates a denser region, which would dictate an inlier, while values significantly greater than 1 indicate outliers.

The Master utilizes formula (5) to compare the *laxity* values of the jobs running in the system, based on the *laxity* values reported by the Workers. If a value greater than 1 is detected, the corresponding node is marked as overloaded, giving the option to execute some of its tasks on a different Worker.

4 Skewed data

As described earlier, the MapReduce framework is susceptible to severe load imbalances caused due to the skewness exhibited in the *partitions*. Recall, that in the original MapReduce framework [10], a *partition* consists of all the intermediate pairs that give the same result when the partitioning function is applied to them. Each *partition* is then assigned to a different *reduce* task. The most commonly used partitioning function, utilized also

in Hadoop, is $hashcode(key)$ mod R. In our system two types of skewness frequently occur:

Skewed Key Frequencies. This occurs when some *keys* appear more frequently in the intermediate pairs, thus the *partition* they are part of becomes extremely large. This issue can be solved by putting more work on the *map* tasks, by aggregating the *values* of the same *key*, and creating intermediate pairs in the form of (*key, list_of_values*).

Skewed Tuple Sizes. This applies to *(key, value)* pairs with complex processing structures on the *value* field. For example in the Twitter social network there is a large assymetry in the types of users and their friendship lists, as a result, pairs have varying sizes, depending on the number of objects that occupy their lists. So this case applies to *partitions* for which the (*key, list_of_values*) pairs contain lists with large amount of data.

We focus on the *Skewed Tuple Sizes* problem, as it has the most significant impact on the execution time of *reduce* tasks. Recent works [14], [15], [26] have shown that solving this problem is not trivial; these primarily aim at partitioning the data in such a way so that all *reduce* tasks finish their processing in similar times.

We propose an approach that uses more *partitions* than the number of *reduce* tasks and takes into account the *partitions*' sizes and our estimates on the task execution times on Worker nodes, assigning the *partitions* in such a way that all *reduce* tasks contribute to the data processing, according to their processing capabilities. We propose an approach that puts more work on powerful nodes, assigning multiple *partitions* on them, while exploiting the slower nodes by assigning them light-sized *partitions*.

4.1 Partition Size Calculation

We exploit two approaches to calculate the size of the *partitions*, the first utilizes the amount of values corresponding to the keys of a *partition* (similar to [14]), while the second uses the Count-Min Sketches [9] data-structure which enables the usage of more hash functions in the calculations.

Simple Partitions. Assume we have p *partitions*, with $p \geq R$. Let $s_m(k)$ be the number of *values* in the *list_of_values* that corresponds to key k, on a *map* task m, $m \in \{0,...,M\}$. We define as $P_m(i), i \in \{0,...,p\}$ the set that contains the *keys* of the i-th *partition* on the m-th *map* task. Then the total size for this *partition* on *map* task m will be: $S_m(i) = \sum_{k \in P(i)} s_m(k)$.

Each *map* task calculates the sizes of all generated *partitions* and sends them to the Master node, who is responsible to aggregate these values for each *partition* in order to calculate its total size. So the Master computes for each *partition* the following value: $S(i) = \sum_{m \in \{1,...,M\}} S_m(i), i = \{0,...,p\}$. These values will be the *partitions*' sizes and will be utilized in the dynamic partitioning assignment algorithm (discussed in the next section).

Count-Min Sketches. Our second technique for estimating the *partitions*' sizes is based on the use of sketches. A sketch is a synopsis data-structure utilized extensively in query-optimization [11]. It provides the capability of capturing the basic features of a dataset by monitoring a significant subset. We use a special type of sketch, called Count-Min Sketch [9], which is mainly used for frequency counting in data streams [8].

Each *map* task creates a local sketch which can be seen as a two-dimensional array, $sketch_m[i,j]$, that stores information regarding the generated *key-value* pairs. Each row of the array corresponds to a different hash function that can be used for the distribution of the intermediate *key-value* pairs to the *partitions*, and each column corresponds to a different *partition*. So, suppose that we have d rows, $H = \{h_i,\ i = 1,...d\}$ be the set of the chosen hash functions, and p columns, the same number as the *partitions* that will be used. It is recommended in [9] that the chosen hash functions need to be pairwise-independent, so we generate d hash functions in the form of $f(x) = (a*x+b)\ mod\ pr$, where a,b are random integers and pr a prime number.

When a new *key-value* pair has been generated, then each of the d hash functions are applied to it, and for the j corresponding position in the array, a counter increases by one, because one more *value* will be added to this *partition*. Initially: $sketch_m[i,j] = 0, \forall i \in \{1,...,d\}, j \in \{1,...,p\}$. So when all the intermediate pairs have been generated, we have:

$$sketch_m[i,j] = \sum_{\forall k: h_i(k)=j} s_m(k), \forall i \in \{1,...,d\}, j \in \{1,...,p\} \quad (6)$$

When all *map* tasks have finished, the generated sketches are emitted to the Master for the creation of the global sketch array. The global sketch will be also a $d \times p$ array, and will be populated using the following equation:

$$sketch[i,j] = \sum_{m=1}^{M} sketch_m[i,j], \forall i \in \{1,...,d\}, j \in \{1,...,p\} \quad (7)$$

This global sketch holds all the information about the *partitions*' sizes, and will be utilized from the *partitions*' assignment algorithm.

4.2 Dynamic Partitioning Algorithm

Once the *partitions* sizes have been estimated, the goal of the dynamic partitioning algorithm is to decide the placement of the *partitions* to the corresponding *reduce* tasks in a way that minimizes the execution time of the *re-*

duce phase, thus increasing the possibility of jobs meeting their deadlines.

Simple Partitions. In the Simple Partitions scheme, we estimate the execution time of each *partition* assigned on a specific *reduce* task, via the *k-NN* estimator (discussed in Section 3.1). So, we sort the *partitions* with respect to their sizes in descending order and try to find the *reduce* task with the smaller execution time. As a *reduce* task might have some *partitions* already assigned to it, we estimate whether the new assignment will still allow the end-to-end execution times of the currently scheduled tasks to be within their deadline constraints, even if we added the new *partition*. The sorting of the *partitions* ensures that the most heavy-sized will be assigned to *reduce* tasks which run on Workers that exhibit the best performance. Finally the Master returns the corresponding assignment to the *map* tasks, in order to know where to emit the generated *partitions*.

Sketch-based approach. In the Sketch-based approach, the same procedure is applied only this time for each row of the global sketch array. The idea is to generate a *partitions*' assignment for each of the possible hash functions and then choose the function that achieves the least execution time. The function's assignment plan will be utilized for the actual distribution of the *partitions*. This approach is applicable because each row of the global sketch table can be seen as a Simple *Partitions* sizes model. The sketch-based approach adds an additional cost in the *partitions*' assignment procedure because the *partitions*' assignment algorithm must be applied for each row of the sketch array. On the other hand, it increases the possibility of achieving a better *partitions*' assignment, with respect to the execution time, because more assignment plans are considered.

5 Evaluation

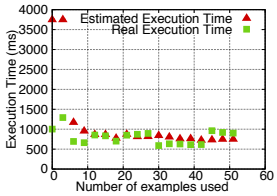
We have performed an extensive experimental study of our approach on Planetlab, a fully distributed heterogeneous environment, using 82 processing cores in total; one dedicated node was utilized as Master and the others were Worker nodes executing *map* and *reduce* tasks. We assume that the Master is failure-free.

Two different jobs were used for the evaluation of our DynamicShare framework. The first job was a Twitter friendship request query on 2GB of available data during the period of Jan 1, 2013 to April 30, 2013, extracted using the Streaming API 2 of Twitter [36], where the goal of each MapReduce job was to identify the unique friends of each user, by examining the tagged and mentioned parts of a tweet. We used a total of 5,900,000 tweets, distributed to fifty-nine available processing cores, each holding about 100,000 tweets. Thus the job consisted of 59 *map* and 23 *reduce* tasks. The job was issued with two different deadlines, the first with $15,000$ ms (strict deadline) and the second with $20,000$ ms (relaxed deadline). The *map* tasks read the available tweets and for each tweet a $(user\_id, list\_of\_friends)$ pair is emitted to the appropriate *reduce* tasks. The latter receive this input and create for each *user_id* the set that contains his unique friends.

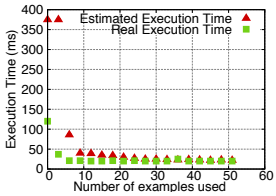
The second job was a friends counting application for a 39MB Youtube [41] social graph. The goal of the job was to calculate the number of unique friends per user, that is, the degree of each node in the social graph. The dataset contained 2,987,628 edges, which were distributed to the available *map* slots leading to approximately 45,000 edges per *map* task. We used the same number of *map* and *reduce* tasks as in the Twitter job, in order to have a fair comparison between the two applications. Youtube jobs were also issued with two different deadlines, to take into account two different type of urgencies strict and relaxed, specifically as strict deadline we used 2000 ms, while as relaxed 4000 ms. These jobs represent commonly issued jobs on Facebook and Yahoo! [6], [30] and thus demonstrate the applicability of our framework in a production workload.

Accuracy of Estimation Model. In the first set of experiments we evaluated the accuracy of our estimation model. In Figure 4 we illustrate the accuracy of our model by comparing the estimated and the actual execution time of a task running on a Worker, as a function of different numbers of previous runs used for the estimation. The results are from the execution runs of one Worker, but similar results were observed for all Workers. As the figure shows, initially when we do not have any previous run of the job's task, the estimated execution time is larger than the actual execution time of the task. However, as tasks execute and more observations become available, the estimates from the *k-NN* estimator are close to the real one.

Laxity Based Scheduling. In the second set of experiments we evaluate the benefit of the least laxity first scheduling (LLF) approach by measuring its ability to meet the deadlines of the jobs. We compared our approach with the following algorithms: (i) Earliest Deadline First (EDF) scheduling as was proposed in [39], where the scheduling criteria is the deadline value, tasks with smaller deadlines will run first. (ii) First In First Out (FIFO) scheduling, where the tasks are ordered based on their arrival order (this is the default scheduling approach used of Hadoop), and (iii) Fair Scheduling (FAIR) scheduling where all tasks are scheduled round-robin so that they get equal time on the available slots. Similarly to [30] we utilized Poisson job arrivals for simulating the assignment of jobs to the framework. A fixed 70% percentage of the assigned jobs had strict deadlines, while the rest had relaxed. This workload mix was used for the

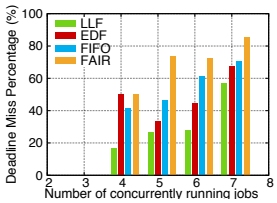


(a) Twitter dataset

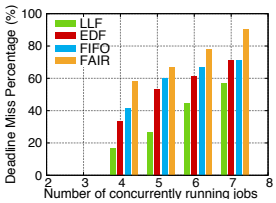


(b) Youtube dataset

Figure 4: Comparison of real and estimated execution time



(a) Twitter dataset



(b) Youtube dataset

Figure 5: Comparison of percentages of deadline misses

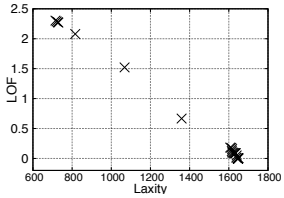


Figure 6: LOF Outlier Detection

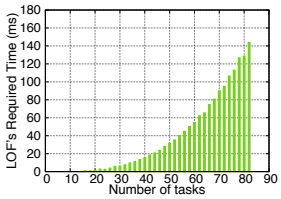
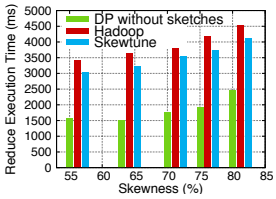
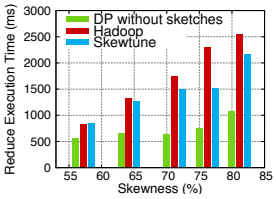


Figure 7: LOF Overhead



(a) Twitter dataset



(b) Youtube dataset

Figure 8: Comparison between Hadoop, DP and Skewtune under varying skewness

evaluation of all four algorithms.

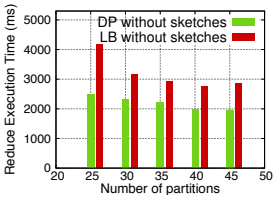
We evaluated the ability of the scheduling algorithms to meet the job deadlines, for varying number of concurrently running jobs. As shown in Figure 5, for a small number of concurrent jobs, all algorithms are able to meet their deadlines. The experiment shows that at all times LLF maintains the smaller percentage of deadline misses. LLF achieves good results even for higher number of jobs, for example when 6 jobs were issued, there was a significant increase on the deadline misses on all the other algorithms due to the increase of the required execution times on some Workers, however, LLF took into account this situation and scheduled the tasks appropriately, thus maintaining few deadline misses.

LOF Evaluation. We now illustrate the working of our anomaly detection algorithm. Due to lack of space we present only the results for the Youtube jobs (as was expected, Twitter jobs had similar results). In Figure 6 we display a snapshot of the LOF's execution. The algorithm identifies overloaded nodes by examining each task's estimated *laxity* value, under normal operation tasks of the same job in different Worker nodes would have similar *laxity* values. In Figure 6 you see that the majority of the tasks have *laxity* values close to 1650 ms. However, five tasks have significant lower *laxity* values resulting to an increase of their lof values and the characterization of the corresponding Workers as overloaded.

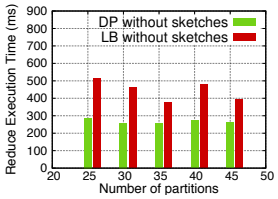
We also examine the overhead of the LOF algorithm in terms of its execution time. The execution time is mainly affected by the number of tasks that report their *laxity* values and need to be examined. So we run multiple *Youtube* jobs with varying number of tasks and report LOF's required execution times (shown in Figure 7). As expected, the execution time increases with the number of tasks. However, the figure shows that the algorithm takes only a few milliseconds to execute, even when it examines 84 tasks.

Dynamic Partitioning. In the last set of experiments we point out the benefits of our proposed *partitions*' assignment algorithm, *Dynamic Partitioning* (*DP*), with respect to handling skewed intermediate data on the *Twitter* and *Youtube* jobs. To avoid the key frequencies skewness, *map* tasks merge for each *key* all the values into a single $(key, list\_of\_values)$ intermediate pair. We calculate the skewness between the generated *partitions* with the following equation: $skewness = 100 - 100 * \frac{\sum_{i=1}^{p} S(i)}{p * max\{S(i)\}}$ The closer the ratio is to 1 the less skewed are considered the generated intermediate pairs. Let P_r be the set that contains the *partitions* processed by *reduce* task r, then the total size of data processed by r will be: $S(P_r) = \sum_{i \in P_r} S(i)$. The achieved balance in regards to the data processed by the *reduce* tasks is given by the following formula: $Balance = 100 * \frac{\sum_{r=1}^{R} S(P_r)}{R * max\{S(P_r)\}}$ The larger the ratio, the more balanced is considered the distribution of the *partitions*, because all *reduce* tasks will process approximately the same amount of data.

We compare the *DP* algorithm utilized by our DynamicShare framework to two schemes: (i) the default *Hadoop* approach where the number of *partitions* is fixed, and equals the number of *reduce* tasks, and (ii) *Skewtune* [27] (*Hadoop*'s most popular enhancement for the skewness issue) that uses the same setting as *Hadoop* but also monitors the execution of *reduce* tasks. When it

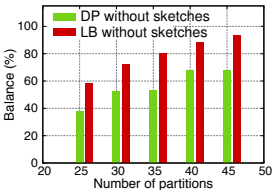


(a) Twitter dataset

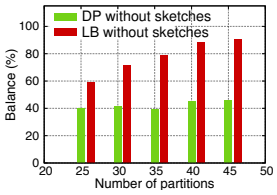


(b) Youtube dataset

Figure 9: Comparison between LB and DP in regards to execution time

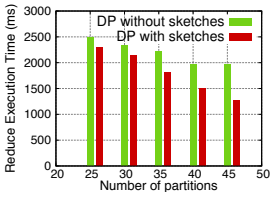


(a) Twitter dataset

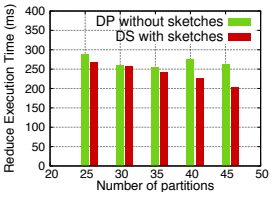


(b) Youtube dataset

Figure 10: Comparison between LB and DP in terms of balance

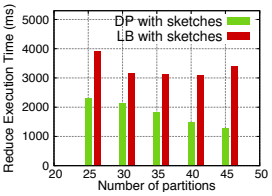


(a) Twitter dataset

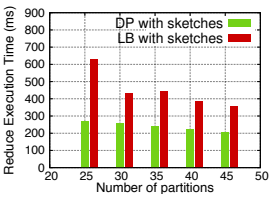


(b) Youtube dataset

Figure 11: Comparison of DP with and without sketches in regards to execution time



(a) Twitter dataset



(b) Youtube dataset

Figure 12: Comparison between LB and DP in regards to execution time when sketches are enabled

detects a task which significantly lacks behind the other normally running, it orders the slow task to repartition its assigned data to the faster *reduce* tasks.

Figure 8 shows the comparison between our algorithm (Simple Partitions schema) with the *Hadoop* MapReduce approach and *Skewtune*. *DP* maintains the execution time of the *reduce* phase low even when the *skewness* reaches 80%. *Skewtune* fails to meet the performance of our algorithm due to the extra overhead of the repartitioning. Although it detects the tasks that suffer from skewed data it requires the repartitioning of their assigned data to the other normally running tasks, a procedure that is beneficial in case of long running jobs as was pointed out in [27], but for short running jobs, such the ones we examine in our work, leads to performance degradation. We do not consider *Skewtune* in case of variable number of *partitions* because it works at *reduce* task level. Regardless of the chosen number of *partitions* to use, some of them will be assigned in a task running on a slow node. These *partitions* will have to be repartitioned thus the overhead will be similar with the presented case.

In the previous experiment we evaluated *DP* with fixed number of *partitions* equal to the *reduce* tasks, in order to have a fair comparison with *Skewtune* and *Hadoop*. To examine the impact of extra *partitions*, we compared our algorithm with the *Load Balance (LB)* algorithm proposed for MapReduce in [14] which also proposes the usage of more *partitions*. The *LB* algorithm strives to maximize the *Balance* metric via a fair distribution of the generated *partitions*. For a fair comparison with our *DP* algorithm when sketches are utilized, we enhanced *LB* with sketches for the estimation of the *partitions*' sizes. In the *partitions*' assignment, each hash function of the global sketch is examined and the one that achieves the best *Balance* is chosen.

The displayed results concern 80% *skewness* between the *partitions*' sizes. Having the same number of *partitions* as *Hadoop* on *LB* does not offer any gain because the *partitions*' assignment will be same as *Hadoop*'s, so we only consider larger values for the number of *partitions*. In Figure 9 you can see the execution time of the *reduce* phase when we use different number of *partitions*. *LB* achieves better results than the default *Hadoop* approach but the *reduce* phase requires higher execution time than our approach. This is mainly due to the fact that our approach is more opportunistic and less "fair" in the work that will be executed from the different tasks. Tasks on nodes with high processing capabilities will process more *heavy-sized partitions* than those that run on slower nodes.

This "unfairness" between the *partitions*' sizes that are processed per *reduce* task, is illustrated in Figure 10. For 40 *partitions*, *LB* achieves approximately 90% *Balance* in the *partitions*' sizes that are processed by the *reduce* tasks for the Twitter job, while our proposal reaches 64%. The difference in the execution times is significant though, as our approach requires approximately 1900 ms while *LB* 2800 ms. The results indicate that in heterogeneous environments, trying to achieve balance between the work assigned on the nodes of the cluster, may not be the right approach. An opportunistic algorithm, such as *DP*, achieves better results as it considers the heterogeneity during the *partitions*' assignment.

Finally we examined the benefit of using sketches in the *partitions*' assignment. Figure 11 shows that sketches mitigate the required execution time of the *reduce* phase as the number of *partitions* increases. This is the expected behaviour since the *key-value* pairs have

more available *partitions* to be spread. The extra hash functions used in the sketches can generate more balance sized *partitions*, thus better decisions are possible for the *DP* algorithm to minimize the *reduce* phase execution time. We examined the requirements of the two approaches in regards to the time they require for deciding the *partitions'* assignment plan. For 45 *partitions*, the no-sketches approach required approximately 80 ms, while when we applied sketches, the assignment required 200 *ms*, so for cases such as the Twitter jobs where the benefits of the sketches approach are larger than 500 ms the extra overhead is negligible. However for very short jobs such as Youtube, a no sketches approach is preferable because the benefits of the sketch-based assignment are overlapped by the extra overhead of the assignment algorithm. In Figure 12, we display results concerning the comparison between *DP* and *LB* when sketches are enabled for both algorithms, although *LB* reduces the required execution time, *DP* still achieves better results.

6 Related Work

Zaharia *et al* [42] were the first to study the problem of scheduling MapReduce jobs in heterogeneous environments. They proposed techniques for prioritizing and scheduling backup copies of slow tasks. Contrary to their work, our approach focuses on meeting real-time response requirements for the tasks and identifies straggling tasks via their *laxity* values. In [1], the authors propose task-stealing solutions in the *map* phase. In an heterogeneous environment like Planetlab, task-stealing could degrade the performance due to the communication overhead between the nodes, and thus could augment the problem. A utility-driven task placement strategy was proposed in [31] using extra processing slots per node when possible. [39] propose the usage of EDF scheduling of user submitted jobs. Our approach differs from them because it does not only consider the real-time demands of the jobs, but also effectively handles the issue of data skewness.

Much work has been done with respect to estimating the execution times of MapReduce tasks such as [29], which utilizes debug runs to calculate the processing speeds of the assigned job. [21] applied *non-parametric* regression for tasks executing in heterogeneous environments, but not on a MapReduce framework, and also using only the input data size as the vectors' value. We were inspired by recent works in automatic anomaly detection ([3], [13], [34], [35]) and adopted the usage of machine learning techniques for the estimation of the tasks' execution times.

Focus on the skewed data impact on MapReduce was mainly expressed by [14], [15], [27] and [28]. We compared our approach with these proposals and displayed results that indicate their inapplicability in our setting. Techniques like [14] and [15] aim to equally distribute the *partitions* to the available *reduce* tasks, however as we pointed out in the experiments such decision is not beneficial in a heterogeneous environment. [27] adds the overhead of repartitioning the assigned *partitions*, an overhead which deteriorates the performance of short jobs and can lead to deadline misses. [32] requires a pre-processing step for estimating the *partitions'* sizes, adding overhead in the calculations thus making the execution of short jobs impossible.

Authors in [16] propose a new abstraction on top of Hadoop, the *Shuffler* component which is responsible for keeping the received partitions in the node where the *reduce* tasks will run. The newly added component enables intermediate data transmission between the *map* and *reduce* phase, reducing the cost of the *shuffle* phase. However, as it was pointed out in [7] when the intermediate data to be emitted are rather small, as in our case, the benefits of online transmission are negligible.

7 Conclusion

In this paper we study the problem of scheduling real-time skewed MapReduce jobs in heterogeneous environments. We propose a holistic approach based on (a) a non-parametric regression technique for estimating the execution times of the tasks, (b) dynamic distributed least laxity first scheduling algorithm for scheduling jobs end-to-end, (c) techniques for identifying straggling nodes, and (d) dynamic partitioning algorithms to handle the impact of the data skewness on the execution times of the tasks. Our experimental results on Planetlab indicate a clear improvement in the system's performance. In our future work we aim at examining if it is possible to dynamically decide the number of *partitions* used per job. This decision will enable us to balance the trade-off between *reduce* phase execution time and the computation overhead of the *partitions'* assignment algorithm.

8 Acknowledgments

This research has been co-financed by the European Union (European Social Fund ESF) and Greek national funds through the Operational Program Education and Lifelong Learning of the National Strategic Reference Framework (NSRF) - Research Funding Program:Thalis-DISFER, Aristeia-MMD, Aristeia-INCEPTION Investing in knowledge society through the European Social Fund, the FP7 INSIGHT project and the ERC IDEAS NGHCS project.

References

[1] AHMAD, F., CHAKRADHAR, S., RAGHUNATHAN, A., AND VIJAYKUMA, T. Tarazu: Optimizing MapReduce On Heterogeneous Clusters. *ASPLOS, London, UK* (2012).

[2] AMAZON'S DYNAMODB. `http://aws.amazon.com/dynamodb/`.

[3] BODÍK, P., GRIFFITH, R., SUTTON, C., FOX, A., JORDAN, M., AND PATTERSON, D. Statistical Machine Learning Makes Automatic Control Practical for Internet Datacenters. *HotCloud* (2009).

[4] BOUTSIS, I., AND KALOGERAKI, V. Resource Management using Pattern-based Prediction to Address Bursty Data Streams. *ISORC 2013, Paderborn, Germany* (2013).

[5] BREUNIG, M. M., KRIEGEL, H.-P., T.NG, R., AND SANDER, J. LOF: Indentifying Density-Based Local Outliers. *SIGMOD* (2000).

[6] CHEN, Y., GANAPATHI, A., GRIFFITH, R., AND KATZ, R. The Case for Evaluating MapReduce Performance Using Workload Suites. *MASCOTS* (2011).

[7] CONDIE, T., CONWAY, N., ALVARO, P., HELLERSTEIN, J. M., ELMELEEGY, K., AND SEARS, R. MapReduce Online. *No. UCB/EECS-2009-136* (2009).

[8] CORMODE, G., AND HADJIELEFTHERIOU, M. Finding Frequent Items in Data Streams. *Proceedings of the VLDB Endowment Volume 1 Issue 2, August, Pages 1530-1541* (2008).

[9] CORMODE, G., AND MUTHUKRISHNAN, S. An Improved Data Stream Summary: The Count-Min Sketch and its Applications. *Journal of Algorithms Volume 55, Issue 1, April 2005, Pages 5875* (2005).

[10] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified Data Processing on Large Clusters. *OSDI, San Francisco, CA* (2004).

[11] DOBRA, A., GAROFALAKIS, M., GEHRKE, J., AND RASTOGI, R. Processing Complex Aggregate Queries over Data Streams. *SIGMOD* (2002).

[12] DOU, A. J., KALOGERAKI, V., GUNOPULOS, D., MIELIKINEN, T., AND TUULOS, V. Scheduling for real-time mobile mapreduce systems. *DEBS 2011, New York, New York* (2011).

[13] FOX, A., KICIMAN, E., AND PATTERSON, D. Combining Statistical Monitoring and Predictable Recovery for SelfManagement. *WOSS* (2004).

[14] GUFLER, B., AUGSTEN, N., REISER, A., AND KEMPER, A. Handling Data Skew In MapReduce. *CLOSER* (2011).

[15] GUFLER, B., AUGSTEN, N., REISER, A., AND KEMPER, A. Load Balancing in MapReduce Based on Scalable Cardinality Estimates. *ICDE* (2012).

[16] GUO, Y., RAO, J., AND ZHOU, X. iShuffle: Improving Hadoop Performance with Shuffle-on-Write. *Presented as part of the 10th International Conference on Autonomic Computing* (2013).

[17] HADOOP. `http://lucene.apache.org/hadoop`.

[18] HADOOP CAPACITY SCHEDULER. `http://hadoop.apache.org/common/docs/r0.19.2/capacity_scheduler.html`.

[19] HADOOP FAIR SCHEDULER. `http://hadoop.apache.org/mapreduce/docs/r0.21.0/fair_scheduler.html`.

[20] IBM INFOSPHERE BIGINSIGHTS. `http://www-01.ibm.com/software/data/infosphere/biginsights/`.

[21] IVERSON, M. A., ÖZGÜNER, F., AND J.FOLLEN, G. Run-Time Statistical Estimation of Task Execution Times for Heterogeneous Distributed Computing. *HPDC* (1996).

[22] JIN, H., YANG, X., SUN, X.-H., AND RAICU, I. ADAPT: Availability-aware MapReduce Data Placement for Non-Dedicated Distributed Computing. *ICDCS, page 516-525. IEEE* (2012).

[23] KALOGERAKI, V. Resource management for real-time fault-tolerant distributed systems. *PhD Thesis, Univ. of California Santa Barbara* (2000).

[24] KALOGERAKI, V., MELLIAR-SMITH, P. M., AND MOSER, L. E. Dynamic scheduling of distributed method invocations. *RTSS* (2000).

[25] KARDOSA, M., AND CHANDRA, A. Resource Bundles: Using Aggregation for Statistical Wide-Area Resource Discovery and Allocation. *ICDCS* (2008).

[26] KOLB, L., THOR, A., AND RAHM, E. Load Balancing for MapReduce-based Entity Resolution. *ICDE* (2012).

[27] KWON, Y., BALAZINSKA, M., HOWE, B., AND ROLIA, J. SkewTune: Mitigating Skew in MapReduce Applications. *SIGMOD* (2012).

[28] LIU, Y., LI, M., ALHAM, N. K., HAMMOUD, S., AND PONRAJ, M. Load balancing in MapReduce environments for data intensive applications. *Fuzzy Systems and Knowledge Discovery (FSKD), 2011 Eighth International Conference on Shanghai 26-28 July* (2011).

[29] MORTON, K., FRIESEN, A., BALAZINSKA, M., AND GROSSMAN, D. Estimating the Progress of MapReduce Pipelines. *ICDE 2010* (2010).

[30] PALANISAMY, B., SINGH, A., LIU, L., AND LANGSTON, B. Cura: A Cost-optimized Model for MapReduce in a Cloud. *IPDPS* (2013).

[31] POLO, J., CASTILLO, C., CARRERA, D., BECERRA, Y., WHALLEY, I., STEINDER, M., TORRES, J., AND AYGUADÉ, E. Resource-Aware Adaptive Scheduling for MapReduce Clusters. *Middleware'11 Proceedings of the 12th ACM/IFIP/USENIX international conference on Middleware Pages 187-207* (2011).

[32] RAMAKRISHNAN, S. R., SWART, G., AND URMANOV, A. Balancing reducer skew in MapReduce workloads using progressive sampling. *SoCC '12 Proceedings of the Third ACM Symposium on Cloud Computing* (2012).

[33] SCOTT, D. Multivariate density estimation: Theory, practice and visualization. *Theory, Practice and Visualization. Wiley & Sons* (1992).

[34] TAN, Y., GU, X., AND WANG, H. Adaptive System Anomaly Prediction for Large-Scale Hosting Infrastructures. *PODC* (2010).

[35] TAN, Y., NGUYEN, H., SHEN, Z., GU, X., VENKATRAMANI, C., AND RAJAN, D. PREPARE: Predictive Performance Anomaly Prevention for Virtualized Cloud Systems. *ICDCS* (2012).

[36] TWITTER. `http://twitter.com`.

[37] UNDER THE HOOD: SCHEDULING MAPREDUCE JOBS MORE EFFICIENTLY WITH CORONA. https://www.facebook.com/notes/facebook-engineering/under-the-hood-scheduling-mapreduce-jobs-more-efficiently-with-corona/10151142560538920.

[38] VERMA, A., CHERKASOVA, L., AND CAMBELL, R. H. Resource Provisioning Framework for MapReduce Jobs with Performance Goals. *Proceedings of the 12th ACM/IFIP/USENIX International Middleware Conference (Middleware'2011), Lisboa, Portugal, December 12-16* (2011).

[39] VERMA, A., CHERKASOVA, L., KUMAR, V. S., AND CAMBELL, R. H. Deadline-based Workload Management for MapReduce Environments: Pieces of the Performance Puzzle. *Network Operations and Management Symposium (NOMS), 2012 IEEE* (2012).

[40] W.HÄRDLE. *Applied nonparametric regression*. Cambridge University Press, 1990.

[41] YOUTUBE. `http://www.youtube.com`.

[42] ZAHARIA, M., KONWINSKI, A., JOSEPH, A. D., KATZ, R., AND STOICA, I. Improving MapReduce Performance in Heterogeneous Environments. *OSDI* (2008).

Colocation Demand Response: Why Do I Turn Off My Servers?

Shaolei Ren
Florida International University

Mohammad A. Islam
Florida International University

Abstract

Data centers are promising participants in demand response programs (i.e., reducing a large electricity demand upon utility's request), making power grid more stable and sustainable. In this paper, we focus on enabling colocation data center demand response. Colocation is an integral yet unique segment of data center industry, where multiple tenants house their servers in one shared facility. Nonetheless, differing from owner-operated data centers (e.g., Google), colocation data center suffers from "split incentive": colocation operator desires demand response for financial incentives but has no control over tenants' servers, while tenants who own the servers may not desire demand response due to lack of incentives. To break "split incentive", we propose a first-of-its-kind incentive mechanism, called iCODE (incentivizing COlocation tenants for DEmand response), based on reverse auction: tenants, who *voluntarily* submit energy reduction bids to colocation operator, will be financially rewarded if their bids are accepted. We formally model how each tenant decides its bids and how colocation operator decides winning bids. We perform a trace-based simulation to evaluate iCODE. We show that iCODE can reduce colocation energy consumption by over 50% during demand response periods, unleashing the potential of colocation demand response.

1 Introduction

Demand response program has been adopted as a national strategic plan for power grid innovation [12]. In a typical demand response program, participating customers, who reduce electricity demands upon requests by utility/load serving entity (LSE), receive financial compensation.[1] Demand response is also favorably recognized as an effective market-based mechanism for increasing the incorporation of renewables into the grid, via the provisioning of economic incentives for reshaping customers' real-time electricity demand subject to time-varying supply availability [16, 40].

Mega-scale data centers are ideal participants in demand response programs and can reduce a *large* electricity demand upon LSE's request, because of their huge yet flexible energy demand [15, 16, 28]. Nonetheless, the existing efforts [4, 5, 15, 16, 28, 29] have only focused on owner-operated data centers (e.g., Google and Amazon), while neglecting another important yet distinctly different type of data center — *colocation* data center, sometimes simply called "colocation" or "colo". In sharp contrast with owner-operated data center whose operator owns and has full control over the servers, colocation is a multi-tenant facility where multiple tenants put their own servers in one shared facility while the data center operator (i.e., facility manager) provides reliable power supply, cooling, and network access.

What makes colocation demand response challenging? A major hurdle for colocation demand response is **"split incentive"**: while colocation provider may desire demand response for incentives from LSE, its tenants may not, because tenants are typically charged based on their subscribed peak power and their bills are not subject to how much energy they consume or when they consume it [11, 35, 39].[2] LSE's incentive programs are not directly open to tenants either, since tenants only have interactions with colocation operator [39]. While colocation operator may manage the non-IT energy consumption (e.g., cooling) for demand response, such actions often have limited energy reduction (as corroborated by real-life field tests [16]) as well as possible detrimental effects on tenants' servers, which may not be as robust as state-of-the-art servers (such as Google's). Using on-site diesel generators to offset electricity usage for demand response is uneconomical for the colocation operator.

How to enable colocation demand response? This

[1] A comprehensive survey of various demand response programs can be found in [3].

[2] Energy-based pricing may also also available (especially for large wholesale tenants), but typically a flat electricity price is used, thereby making tenants "blind" to demand response opportunities.

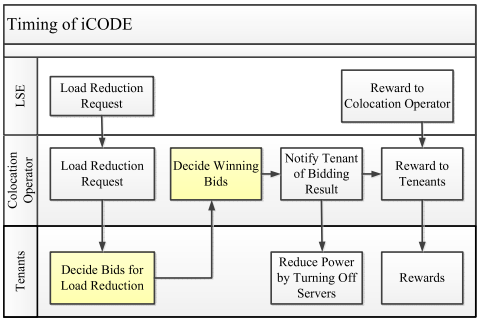


Figure 1: Timing of iCODE in colocation data center.

paper takes the first step to break "split incentive" hurdle for colcation demand response by properly incentivizing tenants. Specifically, we propose a first-of-its-kind market mechanism based on reverse auction that financially compensates tenants who are willing to shed their energy consumption (e.g., by turning off unused servers) for demand response. The proposed mechanism, called iCODE (incentivizing COlocation tenants for DEmand response), is fully voluntary and works in the following steps, as illustrated in Fig. 1. First, when demand response signals/requests are received by colocation operator and passed down to tenants, tenants can submit bids that include how much energy consumption they are willing to reduce and how much payment they want to receive as a compensation. Then, colocation operator selects winning bids that provide large energy reduction yet ask for reasonable payment, such that the energy reduction can be maximized while the total payment to the tenants does not exceed that received by the colocation operator from LSE. Finally, demand response is executed as planned, and payments are made accordingly. The practical implementation of iCODE is lightweight, requiring no manual efforts during execution.

We perform a trace-based simulation study to validate the effectiveness of iCODE in terms of energy reduction for demand response. Compared to the baseline case in which tenants are oblivious to demand response, iCODE can successfully incentivize tenants to reduce energy consumption by more than 50%, demonstrating a promising potential for colocation demand response.

2 Why colocation demand response?

We show two reasons that motivate our study of colocation demand response.

• Colocation is an essential and critically important business model in data center industry, offering a "halfway" solution for companies that do not want to build their own data centers or completely outsource their IT requirements to public cloud providers (e.g., for privacy concerns). Tenants in colocations include not only small and medium businesses for which building wholly-owned data centers is out of the question, but also content distribution providers (e.g., Akamai) and many of the top-branded IT companies (e.g., Amazon and Microsoft) that desire global footprints for their last-mile service latencies. Cloud computing also finds its *physical* home in colocations: e.g., medium-scale cloud providers, such as Salesforce and Box.com, provide their public cloud services in colocations, as building self-operated data centers is still uneconomical for them [34]. It is estimated that in the U.S. there are over 1,000 colocation data centers. With the explosive IT demands across all sectors, many colocation providers are also expanding their data center space [21], and recent analysis shows that colocation market is expected to grow at a compound annual growth rate of 11%, reaching US$ 43billion by 2018 [1].

• Colocations are even more suitable than owner-operated data centers for demand response. First, colocations have huge power demands, and the peak power demand of colocations in New York region exceeds 400MW (comparable to aggregate demand of Google's global data centers) [2, 9, 36]. As noted by a recent Google study [20], "most large data centers are built to host servers from multiple companies" (i.e., colocations). Second, even more importantly, many large colocations are often located in densely-populated metropolitan areas (e.g., Los Angeles [9]) where demand response is particularly desired for peak load shaving, whereas mega-scale owner-operated data centers (e.g., Google) are almost all located in rural areas with very low population densities where the need for demand response is less urgent.

3 Incentivizing colocation tenants for demand response

In this section, we first present an overview of iCODE, formalize the models for tenants and colocation operator, and then present the algorithms underlying iCODE (i.e., deciding tenants' bids and deciding winning bids).

3.1 Overview of iCODE

We begin by presenting an overview of the proposed iCODE mechanism framework to highlight its foundations and why we choose reverse auction.

Foundations of iCODE

iCODE relies the following foundations.

Technology. Turning unused servers off is one of the most extensively studied control knobs for energy saving [20, 25]. While tenants remotely house their servers in colocation, switching servers between active and sleep/off modes can be easily automated without manual efforts [20]. Thus, "turning off unused servers" without noticeably affecting tenants' business is technologically feasible.

Economics. Server energy reduction for demand response clearly requires cooperation from multiple tenants via non-technological mechanisms. Market knobs, such as pricing and incentives, have been leveraged to address various engineering issues [30, 32], and recent research has shown that owner-operated data centers are willing to shut down some servers for demand response incentives [15, 28]. Hence, we take the liberty that the proposed iCODE is worth investigating and promising for enabling colocation demand response.

Why reverse auction?

We first note that dynamically pricing energy usage for demand response, a widely-studied market mechanism (e.g., in smart grid [32]), may not be as plausible as it appears in the context of colocation. First, directly "reselling" energy and modifying energy price may be subject to strict government regulations [38]. Second, dynamically pricing tenants will implicitly enforce all tenants to face uncertain colocation costs, causing business reluctance and/or psychological concerns [30, 46]. Finally, we note that registering tenants to power utility's pricing is not feasible either, since tenants cannot plug their servers into utility's grid directly [39]; instead, tenants need colocation operator's *combined* facility support (e.g., secured access, reliable power, cooling, network), not only facility space [18].

We advocate a reverse auction-based incentive mechanism, as illustrated in Fig. 1. By "reverse", we mean that in our mechanism, it is not the colocation operator who proactively offers rewards to tenants for energy reduction; instead, it is the tenants who, at their own discretion, submit bidding information (including how much energy reduction and how much payment requested) upon receiving a demand response signal. iCODE is "non-intrusive" and tenants are not enforced for demand response or entitled any penalties if they do not participate in demand response.

3.2 Model

As in [15, 16, 28], we ignore the time index and focus on one-time demand response, whose duration T is determined by LSE (e.g., 15 minutes to one hour). Next, we present the models for tenants and colocation operator.

Tenants

We consider N tenants housing their servers in one colocation. Tenant i owns M_i homogeneous servers, while a tenant having multiple heterogeneous types of servers can be viewed as multiple *virtual* tenants each having homogeneous servers. Each server belonging to tenant i has a static/idle power of $p_{i,s}$, dynamic power $p_{i,d}$, and service rate of μ_i (measured in terms of the amount of workloads that may be processed in a unit time) [25]. During the demand response period, the workload arrival rate is denoted by λ_i which can be predicted to a fairly reasonable accuracy using, e.g., regression techniques [25, 37]. Our simulation will also investigate the robustness of iCODE against inaccurate knowledge of λ_i.

Server energy reduction. The baseline case is that tenants do not participate in demand response (e.g., due to lack of incentives, or even not knowing the demand response requests). In this case, all servers are active and workloads are evenly distributed across servers for optimized performance. Thus, the average power consumption of tenant i's servers is $p_i = M_i \cdot \left[p_{i,s} + p_{i,d} \cdot \frac{\lambda_i}{M_i \cdot \mu_i} \right] = M_i \cdot p_{i,s} + p_{i,d} \cdot \frac{\lambda_i}{\mu_i}$, where $\frac{\lambda_i}{M_i \cdot \mu_i}$ is the server utilization.

If tenant i decides to participate in demand response by turning off $m_i \geq 0$ servers, then its average power will be $p_i' = (M_i - m_i) \cdot p_{i,s} + p_{i,d} \cdot \frac{\lambda_i}{\mu_i}$. Hence, energy/load reduction by tenant i will be

$$\Delta e_i(m_i) = (p_i - p_i') \cdot T = m_i \cdot p_{i,s} \cdot T, \qquad (1)$$

where $p_{i,s}$ is the static power and T is the demand response duration.

Tenant cost. Turning off some servers will result in "costs". As an *example*, we consider switching cost and delay cost [25], while other costs (e.g., management costs) can also be factored in.

Switching cost: Turning servers into sleep/off mode and bringing them back to normal operation incur switching/toggling costs, such as wear-and-tear [25]. We denote tenant i's switching cost for one server by α_i (quantified in monetary units), and thus the total switching cost for tenant i is $\alpha_i \cdot m_i$.

Delay cost: We model the workload serving process at each server as an M/M/1 queue. Thus, the average delay for tenant i's workload is $\frac{1}{\mu_i - \frac{\lambda_i}{M_i - m_i}}$. The queueing model has been widely used as an analytic vehicle to provide a reasonable approximation for the actual service process [13, 27]. Note further that delay cost is incurred only when the average delay exceeds a soft threshold $d_{i,th}$: further reducing delay below the threshold makes no difference to human perception, and hence incurs no performance penalty. A large soft delay threshold means the tenant's workloads are more delay-tolerant. Next, we can express the total delay cost as $d_i(m_i) = \lambda_i \cdot \left[\frac{1}{\mu_i - \frac{\lambda_i}{M_i - m_i}} - d_{i,th} \right]^+$, where $[\,\cdot\,]^+ = \max\{0, \cdot\}$.

Colocation operator

Colocation operator provides reliable cooling and power supplies to tenants. Here, we capture the colocation's non-IT energy reduction (e.g., cooling, power distribution, etc.) using the PUE factor γ, which typically ranges from 1.1 to 2.0 [20]. That is, with a total IT energy reduction of $\sum_i \Delta e_i$ by tenants, the facility-level energy re-

duction will be $\gamma \cdot \sum_i \Delta e_i$. To procure a load reduction from customers (including colocation), LSE announces a price denoted by q. Thus, the rewards provided by LSE to colocation operator will be $q \cdot \gamma \cdot \sum_i \Delta e_i$.

3.3 Reverse auction in iCODE

Below, we specify these two elements of iCODE, as highlighted in Fig. 1.

Deciding tenants' bids. In order to participate in demand response, tenants need to be properly incentivized. Below, we denote tenant i's requested payment for turning off m_i servers by

$$c_i(m_i) = w_i \cdot [\alpha_i \cdot m_i + \beta_i \cdot d_i(m_i)], \tag{2}$$

where $w_i \geq 1$ is referred to as *greediness* of tenant i, and $\beta_i \geq 0$ converts delay cost to monetary values (i.e., the larger β_i, the more tenant i cares about its delay performance) [25]. Tenant i may submit multiple bids $(\Delta e_i, c_i)$, each corresponding to one value of $m_i \geq 0$ (i.e., the number of servers turned off). Moreover, tenant i may only choose to turn off up to $\bar{m}_i$ servers such that the delay performance is still tolerable. For convenience, we denote the set of tenant i's bids as $\mathbf{b}_i \subseteq \mathbf{B}_i = \{(\Delta \mathbf{e}_i, \mathbf{c}_i) \,|\, (\Delta \mathbf{e}_i(m_i), \mathbf{c}_i(m_i)),\ m_i = 0, 1, \cdots, M_i - 1\}$, such that $\mathbf{b}_i$ only contains valid bids (e.g., those bids satisfying tenant i's tolerable delay performance or equivalently, m_i is below a threshold $\bar{m}_i$).

We note that in iCODE, each tenant decides its bid purely at its own discretion. Tenants may ask for arbitrarily high payments, but doing so is not of tenants' interests because their bids will be less likely accepted and tenants will receive less payment without noticeably improving their delay performance (see Section 4.2).

Deciding winning bids. In our study, we consider the objective of maximizing the total energy reduction subject to the constraint that colocation operator does not need to compensate tenants out of its own pocket. Mathematically, the problem of deciding winning bids (DWB) can be formalized as:

$$\text{DWB}: \quad \max_{(\Delta e_i, c_i), \forall i \in I} \gamma \cdot \sum_{i \in I} \Delta e_i \tag{3}$$

$$\text{s.t.} \quad \sum_{i \in I} c_i \leq q \cdot \gamma \cdot \sum_{i \in I} \Delta e_i, \tag{4}$$

$$(\Delta e_i, c_i) \in \mathbf{b}_i \cup \{(0,0)\},\ \forall\, i \in I, \tag{5}$$

where I is the set of tenants who submit their bids to colocation operator, (3) specifies the objective of maximizing energy reduction, (4) indicates that the total compensation paid to tenants will not exceed the value received from the LSE (i.e., colocation operator will not lose profits due to active participation in demand response), and (5) specifies that colocation operator can only select "energy reduction, payment" pairs out of the bids submitted by tenants to honor their requests. We add $\{(0,0)\}$ in (5) to indicate that not necessarily all tenants' energy reduction requests will be accommodated (e.g., when they ask for very high payments).

The objective of energy reduction maximization benefits all parties involved: LSE can reduce peak power supply, tenants receive their requested monetary incentives if their bids are accepted, and colocation operator can reduce its energy bill and/or seek green certifications (e.g., LEED [41]) due to lower energy consumption. Note that iCODE can also be adapted for other purposes (e.g., maximizing colocation profit, if permitted by regulations).

While DWB is NP-hard and there exist various approximate solutions [31], we note one approach to solving DWB based on branch and bound technique that can yield a sub-optimal solution with a reasonably low complexity [6]. A sketch is provided below for brevity. Specifically, if we make a relaxation and allow e_i to take continuous values, then the requested payment in (2) is convex in e_i, and DWB becomes convex programming, for which there exists time-efficient methods [7]. The resulting energy reduction is an upper bound on the optimal value of DWB. On the other hand, if we choose a greedy-based approach (e.g., select the bids in ascending order of $\Delta e_i / c_i$), then we will obtain a lower bound on the optimal value of DWB. If the obtained upper and lower bounds are sufficiently close, then we can choose the greedy solution, because its energy reduction is close to the upper bound (and hence optimum, too). Otherwise, we can recursively solve DWB by fixing some bids to be selected and solving a smaller-scale sub-problem. To solve the sub-problem, we find lower/upper bounds via greedy/relaxation approach; if the bounds are still far apart, we further decompose the sub-problem into an even smaller-scale sub-problem. Repeat this process until the gap between the two bounds are sufficiently small or the maximum iteration number is reached. Finally, note that the computational complexity of solving DWB, although interesting by itself, is not a major bottleneck, as colocation operator receives demand response signal from LSE well beforehand and there is no need to solve DWB in real time [15, 28].

Remark. In this paper, we focus on how the colocation operator decides winning bids out of those submitted by tenants, while leaving the possibly strategic bidding process (e.g., tenants strategically place bids to maximize their own benefits) as a future study.

4 Performance evaluation

This section presents trace-based simulation studies to evaluate iCODE. We first present our settings, and then show the simulation results.

Table 1: Default model parameters.

Tenant	#1	#2	#3
Service rate μ (Jobs/hour)	360,000	180,000	30
Delay cost β (¢/ms/10^6jobs)	30	20	0.4
Switching cost α (¢/server)	0.5	0.5	0.5
Greediness factor w	1	1	1
Soft avg. delay threshold	12 ms	25 ms	175 s
Avg. delay constraint	20 ms	40 ms	300 s

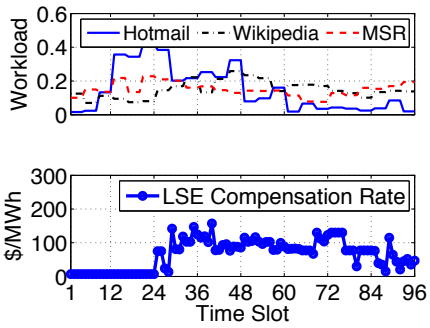


Figure 2: Traces of workload and compensation rate for energy reduction. Each time slot is 15 minutes.

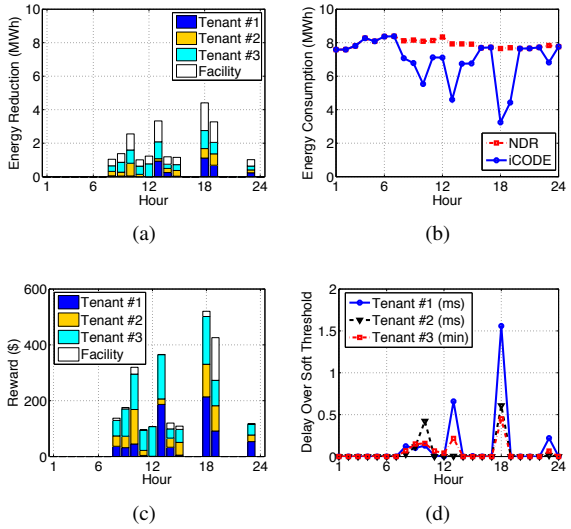


Figure 3: Comparison between iCODE and NDR. **(a)** Energy reduction by iCODE. **(b)** Energy consumption. **(c)** Incentives received. No incentives are provided in NDR. **(d)** Average delay exceeding the soft threshold in iCODE.

4.1 Settings

We consider a colocation facility located in New York, NY (a major market of colocation satisfying financial institution needs [9]), with a PUE of 1.6. The colocation participates in hour-ahead demand response program: when needed, LSE sends demand response request one hour ahead, while each demand respond period is 15 minutes. Nonetheless, due to figure space constraints, we consolidate four 15-minute time slots and show the hourly values for better presentation. Fig. 2 shows a snapshot of the LSE's one-day compensation rate for energy reduction on Feb. 24, 2014, obtained from [33].

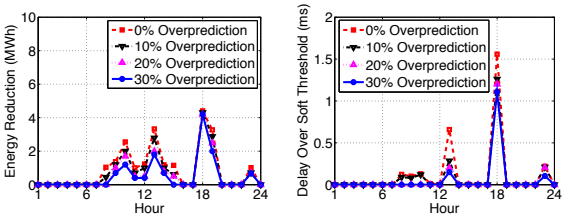
There are three tenants, each possibly representing multiple tenants in practice and having 10,000 homogeneous servers with 150W static and 100W dynamic power. Tenant #1 and #2 process delay-sensitive workload, while tenant#3 processes delay-tolerant workload. The default settings for tenant models are shown in Table 1. In particular, we note that the delay constraint (within a server) for tenant #1 is consistent with the existing interactive service requirement (e.g., web search [19]). Moreover, the switching cost of 0.5 cents for turning one server off for 15 minutes is already higher than the corresponding electricity cost saving achieved by tenants had they run servers in their own data centers (assuming a fair electricity price of 10 cents/kWh). In other words, because of higher cost savings, tenants should be more willing to turn off unused servers in colocation, than they would if they had in-house data centers (which has been extensively studied [37]). We obtain the three tenants' workload traces from [25] ("Hotmail" and "MSR") and [42] ("Wikipedia"). Fig. 2 illustrates a snapshot of the traces, where the workloads are normalized with respect to the maximum service capacity of each tenant's servers (with an average utilization of 15%).

4.2 Simulation results

In this subsection, we first compare our proposed iCODE with benchmark, called NDR. Next, investigate iCODE under various settings to demonstrate its effectiveness.

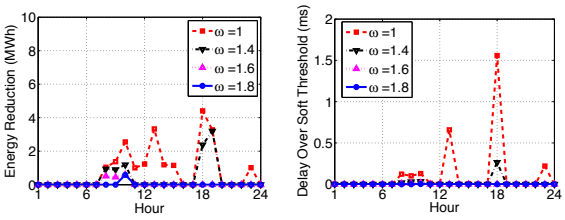
Benchmark: We choose the scenario in which no tenants participate in demand response as our benchmark, called NDR (Non-Demand Response), which is the status quo in colocation.

Comparison between iCODE and NDR. We now compare iCODE with NDR in Fig. 3. We first show the energy reduction by iCODE compared to NDR in Fig. 3(a). It can be seen that more than 4MWh energy reduction per hour can be achieved, which is a fairly significant energy reduction (equivalent to thousands of households) and demonstrates the big potential of colocation demand response. Next, we show the hourly energy consumptions by iCODE and NDR in Fig. 3(b), indicating that more than 50% energy can be slashed in some hours due to the low server utilization in colocation (e.g., 6-12% [17]). Fig. 3(c) shows the monetary incentive received by different tenants. We notice that there is some "residual" incentive paid to colocation operator by LSE, because sometimes tenants do not seek as high incentives as LSE provides. Fig. 3(d) shows the barely-



(a) (b)

Figure 4: Impact of workload overprediction. **(a)** Energy reduction. **(b)** Tenant #1's delay over soft threshold.



(a) (b)

Figure 5: Impact of greediness. **(a)** Energy reduction. **(b)** Tenant #1's delay over soft threshold.

noticeable performance degradation experienced by tenants compared to their soft delay thresholds. There is an up to around 1.5ms increase in average delay beyond the threshold for delay sensitive tenants #1 and 0.5ms for tenant #2. Tenant #3 has 30s delay, which is acceptable for delay tolerant workloads. If tenants cannot tolerate any delay exceeding their thresholds, they can easily remove those bids resulting in intolerable delay performance (see Section 3.3).

Impact of workload overprediction. To cope with unexpected possible traffic spikes, tenants can either turn on more servers as a backup or deliberately overestimate the workload arrival rate by a certain overprediction factor $\phi \geq 0$: the higher ϕ, the more overpredicts. We choose the later approach. Intuitively, when tenants are more conservative and tend more to overpredict workloads, fewer number of servers will be turned off. However, Fig. 4 shows that even when tenants overestimate the workloads by 30%, the energy reduction for demand response is not significantly compromised. We choose 30% because recent studies have shown that the workload prediction error is typically within 30% [26].

Impact of greediness. Tenants may be greedier in the sense that they desire more than their true costs for turning off servers. Here, we increase the greediness factor w_i for tenants. Equivalently, this captures the scenarios that tenants are less willing to participate in demand response unless they are provided sufficiently large incentives. Fig. 5 shows that as tenants are becoming more greedy, the performance becomes better and the energy reduction decreases. Nonetheless, we note that asking for higher payments than actual costs may not be of tenants' interests, because doing so will reduce tenants' financial rewards yet without improving their delay performances (as seen by comparing Fig. 3(d) and Fig. 5(b)).

5 Related work

In this section, we discuss the related work from the following perspectives.

• **Data center optimization:** Optimizing data center operation has received a surging interest recently [8,10,22]. Notably, turning on/off servers based on time-varying incoming workloads is a promising approach to enabling "power proportionality" and reducing energy consumption/cost of data centers [20, 25, 45]. By exploring geographic diversities, optimizing load balancing among multiple data centers can minimize electricity cost [36] and reduce carbon footprint [14]. These studies, however, are all intended for owner-operated data centers and hence cannot be directly applied to colocation unless tenants are properly incentivized.

• **Data center demand response:** Data centers are promising participants in demand response programs. For example, [16] conducts field tests, showing that data centers can reduce energy consumption by 10-25% upon receiving demand response signals. Focusing on owner-operated data centers, [4, 15, 23] study resource management optimization for demand response and frequency regulation in power grid, and [24,28,44] consider the interactions between data centers and utilities and study pricing strategies by utilities. [5] addresses frequency regulation by controlling facility energy consumption via battery charging/discharging, but this technique is difficult to scale due to limited battery size in practice [43].

To our best knowledge, our study makes the first step towards unifying interests of colocation operator and tenants to unleash the promising potential of colcation demand response.

6 Conclusions

In this paper, we studied colocation demand response and proposed an reverse auction-based incentive mechanism, iCODE, which offers tenants with financial rewards for energy reduction. iCODE just requires a lightweight and "non-intrusive" control module that can be automated during run time. We performed a trace-based simulation study to show that iCODE can reduce the hourly energy consumption by over 50%, which is a fairly large amount of energy reduction for demand response programs. iCODE is a first-of-its-kind mechanism to break "split incentive" between colocation operator and tenants, and can also be extended to address other issues in colocation (e.g., energy inefficiency).

References

[1] Colocation market - worldwide market forecast and analysis (2013 - 2018), http://www.marketsandmarkets.com/Market-Reports/colocation-market-1252.html.

[2] Telegeography colocation database, http://www.telegeography.com/research-services/colocation-database/.

[3] U.S. Federal Leadership in Environmental, Energy and Economic Performance - EXECUTIVE ORDER 13514, http://www.whitehouse.gov/administration/eop/ceq/sustainability.

[4] AIKEMA, D., SIMMONDS, R., AND ZAREIPOUR, H. Data centres in the ancillary services market. In *IGCC* (2012).

[5] AKSANLI, B., AND ROSING, T. S. Providing regulation services and managing data center peak power budgets. In *DATE* (2014).

[6] BOYD, S., GHOSH, A., AND MAGNANI, A. Branch and bound methods, http://www.stanford.edu/class/ee392o/bb.pdf, 2003.

[7] BOYD, S., AND VANDENBERGHE, L. *Convex Optimization*. Cambridge University Press, 2004.

[8] CHENG, D., JIANG, C., AND ZHOU, X. Heterogeneity-aware workload placement and migration in distributed sustainable datacenters. In *IPDPS* (2014).

[9] DATACENTERMAP. Colocation USA, http://www.datacentermap.com/usa/.

[10] DENG, N., STEWART, C., GMACH, D., AND ARLITT, M. F. Policy and mechanism for carbon-aware cloud applications. In *NOMS* (2012).

[11] ENAXIS CONSULTING. Pricing data center co-location services, 2009, http://enaxisconsulting.com/downloads/2/67f7fb873eaf29526a11a9b7ac33bfac/1317636458_data_center_pricing.pdf.

[12] FEDERAL ENERGY REGULATORY COMMISSION. Assessment of demand response and advanced metering, 2012.

[13] GANDHI, A., HARCHOL-BALTER, M., DAS, R., AND LEFURGY, C. Optimal power allocation in server farms. In *SIGMETRICS* (2009).

[14] GAO, P. X., CURTIS, A. R., WONG, B., AND KESHAV, S. It's not easy being green. *SIGCOMM Comput. Commun. Rev. 42*, 4 (Aug. 2012), 211–222.

[15] GHAMKHARI, M., AND MOHSENIAN-RAD, H. Energy and performance management of green data centers: a profit maximization approach. In *SmartGridCom* (2012).

[16] GHATIKAR, G., GANTI, V., MATSON, N. E., AND PIETTE, M. A. Demand response opportunities and enabling technologies for data centers: Findings from field studies, 2012.

[17] GLANZ, J. Power, pollution and the internet. In *The New York Times* (Sep. 22, 2012).

[18] HARBOR RIDGE CAPITAL. Colocation data centers: Overview, trends & m&a, http://www.harborridgecap.com.

[19] HE, Y., ELNIKETY, S., LARUS, J., AND YAN, C. Zeta: Scheduling interactive services with partial execution. In *SOCC* (2012).

[20] HOELZLE, U., AND BARROSO, L. A. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan and Claypool Publishers, 2009.

[21] KERRIGAN, J., AND HOROWITZ, D. Avison young data center practice newsletter, Jan. 2014.

[22] LI, S., ABDELZAHER, T., AND YUAN, M. Tapa: Temperature aware power allocation in data center with map-reduce. In *IGCC* (2011).

[23] LI, S., BROCANELLI, M., ZHANG, W., AND WANG, X. Data center power control for frequency regulation. In *PES* (2013).

[24] LI, Y., CHIU, D., LIU, C., PHAN, L. T., GILL, T., AGGARWAL, S., ZHANG, Z., LOO, B. T., MAIER, D., AND MCMANUS, B. Towards dynamic pricing-based collaborative optimizations for green data centers. In *ICDEW* (2013).

[25] LIN, M., WIERMAN, A., ANDREW, L. L. H., AND THERESKA, E. Dynamic right-sizing for power-proportional data centers. In *IEEE Infocom* (2011).

[26] LIU, Z., CHEN, Y., BASH, C., WIERMAN, A., GMACH, D., WANG, Z., MARWAH, M., AND HYSER, C. Renewable and cooling aware workload management for sustainable data centers. In *SIGMETRICS* (2012).

[27] LIU, Z., LIN, M., WIERMAN, A., LOW, S. H., AND ANDREW, L. L. Greening geographical load balancing. In *SIGMETRICS* (2011).

[28] LIU, Z., LIU, I., LOW, S., AND WIERMAN, A. Pricing data center demand response. In *Sigmetrics* (2014).

[29] LIU, Z., WIERMAN, A., CHEN, Y., RAZON, B., AND CHEN, N. Data center demand response: avoiding the coincident peak via workload shifting and local generation. In *SIGMETRICS* (2013).

[30] MA, J., DENG, J., SONG, L., AND HAN, Z. Incentive mechanism for demand side management in smart grid using auction. *IEEE Trans. Smart Grid* (2014 (PrePrint)).

[31] MARTELLO, S., AND TOTH, P. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons, Inc., New York, NY, USA, 1990.

[32] MOHSENIAN-RAD, H., WONG, V. W. S., JATSKEVICH, J., SCHOBER, R., AND LEON-GARCIA, A. Autonomous demand side management based on game-theoretic energy consumption scheduling for the future smart grid. *IEEE Trans. Smart Grid 1*, 3 (Dec. 2010), 320–331.

[33] NEW YORK ISO. http://www.nyiso.com/.

[34] NOVET, J. Colocation providers, customers trade tips on energy savings, Nov. 2013.

[35] PALASAMUDRAM, D. S., SITARAMAN, R. K., URGAONKAR, B., AND URGAONKAR, R. Using batteries to reduce the power costs of internet-scale distributed networks. In *SoCC* (2012).

[36] QURESHI, A., WEBER, R., BALAKRISHNAN, H., GUTTAG, J., AND MAGGS, B. Cutting the electric bill for internet-scale systems. In *SIGCOMM* (2009).

[37] RAO, L., LIU, X., XIE, L., AND LIU, W. Reducing electricity cost: Optimization of distributed internet data centers in a multi-electricity-market environment. In *IEEE Infocom* (2010).

[38] REGULATORY ASSISTANCE PROJECT. Electricity regulation in the US: A guide, Mar. 2011, `www.raponline.org`.

[39] SILICON VALLEY POWER. Data center program, `http://siliconvalleypower.com/index.aspx?page=2088`.

[40] U.S. DOE. `http://energy.gov/`.

[41] U.S. GREEN BUILDING COUNCIL. Leadership in energy & environmental design, `http://www.usgbc.org/leed`.

[42] VASUDEVAN, V., PHANISHAYEE, A., SHAH, H., KREVAT, E., ANDERSEN, D. G., GANGER, G. R., GIBSON, G. A., AND MUELLER, B. Safe and effective fine-grained tcp retransmissions for datacenter communication. *SIGCOMM Comput. Commun. Rev. 39*, 4 (Aug. 2009), 303–314.

[43] WANG, D., REN, C., SIVASUBRAMANIAM, A., URGAONKAR, B., AND FATHY, H. Energy storage in datacenters: what, where, and how much? In *SIGMETRICS* (2012).

[44] WANG, H., HUANG, J., LIN, X., AND MOHSENIAN-RAD, H. Exploring smart grid and data center interactions for electric power load balancing. *SIGMETRICS Perform. Eval. Rev. 41*, 3 (Jan. 2014), 89–94.

[45] YAO, Y., HUANG, L., SHARMA, A., GOLUBCHIK, L., AND NEELY, M. J. Data centers power reduction: A two time scale approach for delay tolerant workloads. In *Infocom* (2012).

[46] ZHONG, H., XIE, L., AND XIA, Q. Coupon incentive-based demand response: Theory and case study. *IEEE Trans. Power Systems 28*, 2 (May 2013), 1266–1276.

Self-Tuning Intel Transactional Synchronization Extensions

Nuno Diegues (ndiegues@gsd.inesc-id.pt) Paolo Romano (romano@inesc-id.pt)
INESC-ID / Instituto Superior Técnico, University of Lisbon

Abstract

Transactional Memory was recently integrated in Intel processors under the name TSX. We show that its performance can be significantly affected by the configuration of its interplay with the software-based fallback: in fact, there does not seem to exist a single configuration that can perform best independently of the application and workload. We address this challenge by introducing an innovative self-tuning approach that exploits lightweight reinforcement learning techniques to identify the optimal TSX configuration in a workload-oblivious manner, i.e. not requiring any off-line/*a-priori* sampling of the application's workload. To achieve total transparency for the programmer, we integrated the proposed algorithm in the GCC compiler. Our evaluation shows improvements up to 2× over state of the art approaches, while remaining within 5% from the performance achievable using optimal static configurations.

1 Introduction

Multi-core and many-core processors are nowadays ubiquitous. The consequence of this architectural evolution is that programmers need to deal with the complexity of parallel programming to fully unveil the performance potential of modern processors.

Transactional Memory (TM) [15] is a promising paradigm for parallel programming, that responds precisely to the need of reducing the complexity of building efficient parallel applications. TM brings to parallel programming the powerful abstraction of atomic transactions, thanks to which programmers need only to identify the code blocks that should run atomically, and not how atomicity should be achieved [22]. This is in contrast with the conventional lock-based synchronization schemes, where the programmer has to specify how concurrent accesses to shared state are synchronized to guarantee isolation. The TM runtime implements this with optimistic transactions, whose correctness is checked to ensure an illusion of serial executions (possibly aborting and rolling back the transaction), even though transactions run concurrently.

Recently, the maturing of TM research has reached an important milestone with the release of the first mainstream commercial processors providing hardware support for TM. In particular, Intel has augmented their instruction set for x86 with Transactional Synchronization Extensions (TSX), which represents the first generation of mainstream and commodity Hardware Transactional Memory (HTM): TSX is available in the 4^{th} generation Core processor, which is widely adopted and deployed ranging from tablets to server machines.

One important characteristic of this hardware support is its best-effort nature: due to inherent architectural limitations, TSX gives no guarantees as to whether transactions will ever commit in hardware, even in absence of conflicts. As such, programmers must provide a software fallback path when issuing a begin instruction, in which they must decide what to do upon the abort of a hardware transaction. One solution is to simply attempt several times before giving up to software. However, under which circumstances should one insist on using TSX before relying on software to synchronize transactions? We show that the answer to this question is clear: there is no one-size fits all solution that yields the best performance across all possible workloads. This means that the programmer is left with the responsibility of finding out the best choices for his application, which is not only a cumbersome task, but may also not be possible to achieve optimally with a static configuration. In fact, also Intel has recently acknowledged the importance of developing adaptive techniques to simplify the tuning of TSX [18].

1.1 Contributions and Outline

We study, to the best of our knowledge for the first time in literature, the problem of automatically tuning the policies used to regulate the activation of the fallback path of TSX. We first highlight the relevance of this self-tuning problem, and then present a solution that combines a set of lightweight reinforcement learning techniques designed to operate in a workload-oblivious manner. This means that we do not require any a priori knowledge of the application, and execute only runtime adaptation based on the online monitoring of applications' performance. We integrated the proposed self-tuning mechanisms within *libitm*, the TM library of the well-know GCC compiler. This allows achieving transparency to the programmers: a property of paramount importance that allows preserving the most compelling motivation of TM, namely its ease of use [14].

To assess our contributions we used the C++ TM specification [1] and relied on a comprehensive set of bench-

marks (most of which had to be ported to use this standard interface). More in detail, our evaluation study includes the well-known STAMP benchmark suite [21], which comprehends several realistic transactional applications with varying contention levels, size of transactions and frequency of atomic blocks. Besides STAMP, we also used a recent TM-based version of the popular Memcached [27], an in-memory web cache used to help scale web page servicing, which is widely used at Facebook. Finally we consider an example of a concurrent data-structure synchronized with TM, namely, a red-black tree. This is representative of important building blocks that are very hard to parallelize efficiently with locks, while generating typically short transactions (unlike some STAMP benchmarks).

In this large set of benchmarks our contributions allowed an average improvement of 2× over existing approaches, including both static heuristics and a state of the art adaptive solution [29] (although devised for software implementations of TM, and not HTM). We organized the rest of the paper as follows. Section 2 provides background on HTM and Intel TSX, whereas Section 3 motivates the relevance of devising self-tuning mechanisms for TSX. Then, in Sections 4 and 5 we present our learning techniques to self-tune TSX, as well as their integration in GCC in Section 6. Finally, we evaluate our proposals in Section 7, describe the related work in Section 8 and conclude in Section 9.

2 Background on Intel TSX

Intel TSX was released as part of the 4^{th} generation of Core processors (Haswell family). It has two main interfaces, called Hardware Lock Elision (HLE) and Restricted Transactional Memory (RTM). To support HLE, two new prefixes (XACQUIRE and XRELEASE) were created, which can be placed in LOCK instructions. In older processors these prefixes are ignored, and the instructions are executed normally, meaning that locks are acquired and released normally. However, in Haswell processors these prefixes allow to elide the lock, such that it is only read but not written, effectively allowing concurrent threads to execute the same critical section at the same time. To ensure correctness, namely prevent data races in such optimistic executions, the hardware keeps track of the footprint accessed speculatively and rolls-back the execution if such footprint is invalidated by cache coherency. In such event, the thread re-executes the critical section but this time acquires and releases the lock normally. Such acquisition aborts concurrent elisions of the same lock, because these hardware speculations had read the lock and as such the lock state became part of their transactional footprint.

RTM leverages on the same hardware but accomplishes better flexibility because it exposes new instructions, namely, XBEGIN and XEND. This interface maps directly to the usual constructions of transactional programming of beginning and committing a transaction. Additionally, the XBEGIN instruction requires the programmer to provide a software handler to deal with transaction aborts. This has the advantage of allowing other strategies rather than giving up immediately on hardware transactions, which is the strategy followed by HLE.

The reason for requiring such software fallback is the best-effort nature of Intel TSX. Due to the inherently limited nature of HTMs [7, 16], TSX cannot guarantee that a hardware transaction will ever succeed, even if run in absence of concurrency. Briefly, TSX uses the L1 cache (private to each physical core) to buffer transactional writes, and on the cache coherence protocol to detect data conflicts. A plausible reason for a transaction never to succeed is because its data footprint does not fit in the L1 cache. Hardware transactions are also subject to abort due to multiple reasons that are not justified by concurrency alone, such as page faults and system calls.

In Alg. 1 we illustrate how GCC compiles transactional applications to use TSX (in the latest stable version 4.8.2 of GCC). In this approach (that we refer as GCC) transactions are attempted in hardware at most twice; if a hardware transaction aborts, the flow of execution reverts back to line 2 with an error code (the transaction can be aborted at any point between lines 3-15). This means that if TSX was always successful, then lines 4-8 would not be executed. When all attempts are exhausted, the execution falls through the fallback and acquires a global lock to execute normally, i.e., without hardware speculation.

To ensure correctness, a hardware transaction reads the lock (line 9) and aborts, either immediately if it finds it locked, or if some concurrent pessimistic thread acquires it before the hardware transaction commits. This mechanism safeguards the correctness of pessimistic executions that run without any instrumentation [4].

Note that HLE can be seen as a degenerate case of Alg. 1 in which the variable *attempts* is initialized with the value 1. In Fig. 1 we use a concurrent red-black tree

Algorithm 1 TSX in GCC

```
1:  int attempts ← 2
2:  int status ← XBEGIN
3:  if status ≠ ok then
4:      if attempts = 0 then
5:          acquire(globalLock)
6:      else
7:          attempts−−
8:          goto line 2
9:  if is_locked(globalLock)
10:     XABORT
11: ▷ ...transactional code
12: if attempts = 0 then
13:     release(globalLock)
14: else
15:     XEND
```

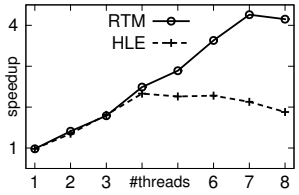


Fig. 1: Red-black tree synchronized with TSX.

Abort code
`retry`: Transient failure
`conflict`: Contention to data
`capacity`: Exceeded cache
`explicit`: XABORT invoked
`other`: Faults, preemption, ...

Fig. 2: Error codes in TSX.

synchronized with both interfaces of Intel TSX and show the speedup relatively to a sequential execution without synchronization. All the experimental results shown in this paper were obtained using an Intel Haswell Xeon E3-1275 processor with 32GB RAM, which has 8 virtual cores (4 physical, each with hyper-threading). Note that, for the time being, this is the maximum number of cores available in a machine with Intel TSX. We ran our experiments in a dedicated machine running Ubuntu 12.04, by using a minimum of 10 trials to obtain statistically meaningful results.

We focus our attention on the significant discrepancy in the performances in Fig. 1. This difference stems from the importance of insisting on the hardware support even when it fails (transiently). Since HLE gives up after one failed attempt, this creates a chain of failed speculations that acquire the lock and prevent concurrent speculations from proceeding via hardware — naturally, this occurs more often with higher concurrency, as the plot shows.

These considerations motivate the use of RTM over HLE (unless there are backwards compatibility concerns). However, as we shall discuss next, the use of RTM raises concerns of different nature, in particular related to the difficulty of properly tuning it.

3 Static Tuning of TSX

So far, we have motivated the usage of RTM, but only presented the simple approach that is implemented currently in GCC, which, as we shall see, is far from optimal. Indeed, in the light of recent findings [8, 17, 31], and based on the experience that we gathered after experimenting extensively with TSX, more effective mechanisms can be used to regulate the activation of the fallback path of TSX. We describe such mechanisms (which we call HEURISTIC) in Alg. 2.

The first point that we raise is that GCC (in Alg. 1) ignores the error codes returned by TSX's begin operation. The error codes available are briefly described in Fig. 2. Taking this into account, we consider that RETRY and CONFLICT codes represent ephemeral aborts, and as such we do not consume the attempts' budget in those cases (line 9). Furthermore, we consider CAPACITY errors to be permanent, and drop all attempts left (line 11). The objective is to avoid trying fruitlessly to use the hardware when it is not able to complete successfully, and short-cut right away to the fallback path.

Secondly, we set the attempts to 5 as that was reported by recent works as the best all-around figure [17, 31]. Choosing a given number is always going to be sub-optimal in some scenario, as it depends tremendously on the workload. Thirdly, we perform a lazy check for the global lock, which safely allows some concurrency with a pessimistic thread and hardware transactions [4].

Finally, we note that GCC suffers from the so-called *lemming effect* [8], in which one thread proceeding to the fallback (by acquiring the global-lock) causes all other concurrent transactions to do so too. This chain reaction can exhaust the attempts, and make it difficult for threads to resume execution of transactions in hardware. One way to avoid it, is by checking the lock before starting the transaction, and waiting in case it is locked (line 2).

Algorithm 2 HEURISTIC based approach for TSX.

```
1:  int attempts ← 5
    ▷ avoid the lemming effect
2:  while(is_locked(global-lock)) do pause  ▷ x86 instruction
3:  int status ← XBEGIN
4:  if status ≠ ok then
5:      if attempts = 0 then
6:          acquire(global-lock)
7:      else
8:          if status = explicit ∨ status = other then
9:              attempts ← attempts - 1      ▷ skipped if transient
10:         else if status = capacity then
11:             attempts ← 0  ▷ give up, likely that it always fails
12:         goto line 2
13: ▷ ...code to run in transaction
14: if attempts = 0 then
15:     release(global-lock)
16: else
17:     if is_locked(global-lock) then
18:         XABORT    ▷ check for concurrent pessimistic thread
19:     XEND
```

An alternative way to deal with the *lemming effect* is to use an auxiliary lock [2]. Briefly, the idea is to guard the global lock acquisition by another auxiliary lock. Aborted hardware transactions have to acquire this auxiliary lock before restarting speculation, which effectively serializes them. However, this auxiliary lock is not added to the read-set of hardware transactions, which avoids aborting concurrent (and successful) hardware transactions. If this procedure is attempted some times before actually giving up and acquiring the global lock, then the chain reaction effect can be avoided, as the auxiliary lock serves as a fallback manager preventing hardware transactions from continuously acquiring the fallback lock and preventing hardware speculations.

3.1 No One-size Fits All

The previous algorithms have a number of tuning knobs that needs to be properly configured. Summarizing: 1) How many times should a transaction be retried in hardware? 2) Should we trust the error codes to guide the decision to give up? and 3) What is the best way to manage the contention on the fallback path?

In order to assess the performance impact that these configuration options can have in practice, we conducted an experimental study in which we considered a configuration's space containing all possible combinations of feasible values of the following three parameters:

• Contention management — **wait** uses the simple wait

and pause approach of Alg. 2; **aux** uses the auxiliary lock [2]; **none** allows transactions to retry freely.

• Capacity aborts — **giveup** exhausts all attempts upon a capacity abort; **half** drops half of the attempts on a capacity abort; **stubborn** decreases one attempt only.

• Budget of attempts — varying from 1 to 16.

We tested all these combinations in our suite of benchmarks, with varying concurrency degrees, reaching the conclusion that there is no *one-size fits all* solution. In Table 1 we show some of the results from these experiments. Naturally, it is impossible to show all the 144 combinations for each benchmark and workload of our suite. We focus only on experiments with 4 threads, one workload per benchmark, and show only the best variant together with GCC (Alg. 1) and HEURISTIC (Alg. 2).

In general, the HEURISTIC algorithm yields some considerable improvements over GCC (notice Genome and Yada, with over 50% reduction in time), although in the other benchmarks it performs either similarly or slightly worse (notice Vacation, with 30% increase in time). However, the most important results are on the rightmost column, where we can see that the best performing variant varies a lot in its characteristics. Furthermore, the best result obtained is also typically better than those obtained by the algorithms seen so far: the geometric mean loss (i.e., additional time) of using GCC or HEURISTIC is of 30% and 21% (respectively) when compared to the best variant that we experimented with. These losses can extend up to 4× and 2× (e.g., Yada). To complement those results, we also show the performance of different TSX configurations in Genome when varying the number of threads, in Fig. 3: we can see that the best performance is obtained with widely different settings, and that even GCC and HEURISTIC can perform better than each other at different concurrency degrees.

The bottom line here is that static configurations of TSX can deliver suboptimal performance as a consequence of the high heterogeneity of the workloads generated by TM applications. The numbers above illustrate how much we could win in this set of benchmarks, workloads and concurrency degree, if we had a dynamic approach that adapts to the workload. Naturally, it is undesirable to require the programmer to come up with an optimal configuration for each workload, in particular because they may be unpredictable or even vary over time.

Another interesting result highlighted by this study is that the parameters' search space can be reduced by one dimension, i.e. contention management, as the **wait** or **aux** had in the vast majority of the cases similar speedups over **none**; whereas, whenever **none** reported to be the best, it was only by a very small margin. This finding suggests therefore to focus on the tuning of two main tuning knobs: i) the policy used to cope with capacity aborts and ii) the settings of the maximum number of attempts for a hardware transaction.

Table 1: Completion time (seconds, less is better) when using different RTM variants (described in Section 3.1).

Benchmark	GCC	HEURISTIC	Best Variant
genome	3.46	1.69	1.59 wait-giveup-4
intruder	7.06	7.92	4.79 wait-giveup-4
kmeans-h	0.41	0.42	0.37 none-stubborn-10
rbt-l-w	6.25	6.40	5.27 aux-stubborn-3
ssca2	5.92	5.97	5.72 wait-giveup-6
vacation-h	6.81	8.99	5.83 aux-half-5
yada	28.5	11.6	6.96 wait-stubborn-15

4 Self-Tuning TSX

The proposed self-tuning solution for TSX adopts an online, feedback based design, which performs lightweight profiling of the applications at runtime. This allows to better fit the workloads of typical irregular applications that benefit most from synchronization facilities such as TSX [14], for which fully offline solutions are likely to fall prey of the over-approximations of solutions based on static analysis techniques. Another appealing characteristic of the proposed approach is that it does not necessitate any preliminary training phase, unlike other self-tuning mechanisms for Software TM (STM) based on off-line machine-learning techniques [10, 24].

Clearly, keeping the overhead of our techniques very low is a crucial requirement, as otherwise any gain is easily shadowed, for instance due to profiling inefficiencies or constant decision-making. Another challenge is the constant trade-off between exploring alternative configurations versus exploiting the current one, with the risk of getting stuck in a possibly sub-optimal configuration.

The proposed technique pursues overhead minimization in a twofold way. It employs efficient and concurrency friendly profiling techniques, which infer system's performance by sampling the x86's TSC cycle counter (avoiding any system call) and relying solely on thread-local variables to avoid inter-thread synchronization/interference. Besides that, it employs a combination of lightweight techniques, borrowed from the literature on reinforcement learning and gradient descent algorithms, which were selected, over more complex techniques, precisely because of their high efficiency.

Another noteworthy feature of the proposed self-tuning mechanism is that it allows for individually tuning the configuration parameters of *each* application's atomic block, rather than using a single global configuration. This feature is particularly relevant in programs that include transactions with heterogeneous characteristics (e.g., large vs small working sets, are contention-prone or not, etc.), which could benefit from using radically different configurations.

Before detailing the proposed solution, we first overview a state of the art solution [3] for a classical reinforcement learning problem, the multi-armed ban-

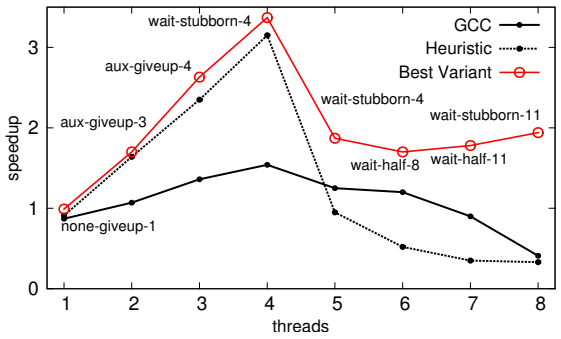


Figure 3: Speedup in Genome (higher is better).

dit [28]. This reinforcement learning technique is the key building block of the mechanism that we use to adapt the capacity abort management policy, which will be described in Section 4.2. We then explain the adaptation of how stubborn should one be in using TSX, i.e. the budget of attempts, in Section 4.3. The combination of those techniques is presented in Section 5.

4.1 Bandit Problem and UCB

The "bandit" (a.k.a. "multi-armed bandit") is a classic reinforcement learning problem that states that a gambling agent is faced with a bandit (a slot machine) with k arms, each associated with an unknown reward distribution. The gambler iteratively plays one arm per round and observes the associated reward, adapting its strategy to maximize the average reward. Formally, each arm i ($0 \leq i \leq k$) is associated with a sequence of random variables $X_{i,n}$ representing the reward of the arm i, where n is the number of times the lever has been used. The goal is to learn which arm i maximizes the average reward : $\mu_i = \sum_{n=1}^{\infty} \frac{1}{n} X_{i,n}$. To this purpose, the learning algorithm needs to try different arms to estimate their average reward. On the other hand, each suboptimal choice of an arm i costs, on average, $\mu^* - \mu_i$, where μ^* is the average obtained by the optimal lever. Several algorithms have been studied to minimize the regret (defined as $\mu^* n - \mu_i \sum_{i=1}^{K} E[T_i(n)]$, where $T_i(n)$ is the number of times arm i has been chosen).

Building on the idea of confidence bounds, the technique of Upper Confidence Bounds (UCB) creates an overestimation of the reward of each possible decision, and lowers it as more samples are drawn. Implementing the principle of optimism in the face of uncertainty, the algorithm picks the option with the highest current bound. Interestingly, this allows UCB to achieve a logarithmic bound on the regret value not only asymptotically, but also for any finite sequence of trials. More in detail, UCB assumes that rewards are distributed in the [0,1] interval, and associates each arm with a value:

$$\bar{\mu}_i = \bar{x}_i + \sqrt{2\frac{\log n}{n_i}} \quad (1)$$

where $\bar{\mu}_i$ is the current estimated reward for lever i; n is the number of the current trial; $\bar{x}_i$ is the reward for lever i; and n_i is the number of times the lever i has been tried. The right-hand part of the sum is an upper confidence bound that decreases as more information is acquired. By choosing, at any time, the option with maximum $\bar{\mu}_i$, the algorithm searches for the option with the highest reward, while minimizing the regret along the way.

4.2 Using UCB learning

We applied UCB to TSX by considering that each atomic block of the application has a slot machine (in the nomenclature of the previous Section 4.1), i.e., a corresponding UCB instance. With it, we seek to optimize the consumption of the attempts upon capacity aborts. In some sense, this models a belief on whether the capacity aborts witnessed are transient or deterministic, which cannot be assessed correctly based only on the error codes returned by aborted transactions. How many capacity aborts should we have to consider that an atomic block is failing deterministically? It is not obvious one can explicitly model such belief in that way; hence why UCB becomes appealing in this context.

We consider the three options identified in Section 3.1 with respect to capacity aborts. This creates three levers ($0 \leq i < 3$) in each UCB instance. We then use Eq. (1), for which we recall n is the number of the current trial (i.e., number of decisions so far for the atomic block), and n_i is the number of times the UCB instance chose lever i. We now specify the reward function for the levers (represented by $\bar{x}_i$). For this we used the number of processor cycles that it takes to execute the atomic block for each configuration. Hence we keep a counter c_i for each lever with the cycles that it consumed so far, and compute the reward $\bar{x}_i$ for lever i with:

$$\bar{x}_i = \frac{1}{c_i/n_i} \quad (2)$$

which means that we normalize the cycles of lever i, giving us a reward in the interval [0,1].

4.3 Using Gradient Descent Exploration

In order to optimize the number of attempts configured for each atomic block, we use an exploration technique, similar to hill climbing/gradient descent search [25]. The alternative of using UCB was dismissed because the parameter has a large space of search that does not map well to the lever exploration. This optimization problem is illustrated by the experiments in Fig. 4, where we show the

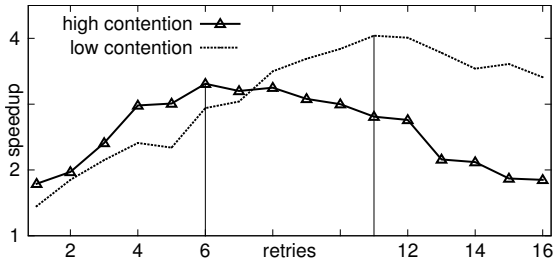


Figure 4: Speedup in Kmeans given attempts (8 threads).

performance improvement at 8 threads in Kmeans when varying the number of attempts for the configuration that yielded the best results. In the plot we show both low and high contention workloads of Kmeans, for which there are significantly different number of attempts yielding maximum values of improvement (namely, 6 and 11).

For this decision we also use the processor cycles that it takes to execute the atomic block, and at the end we execute gradient descent exploration. We augment it with probabilistic jumps to avoid getting trapped in a local maximum during the exploration. Furthermore, we memorize the best configuration seen so far to recover from unfortunate jumps.

For this, we store: the best configuration and performance seen so far (*best*); the last configuration and corresponding performance (*last*); and the current configuration (*current*). Note that here the configuration means simply the number of attempts. Then we use the following rules to guide exploration (strategy called GRAD):

1) with probability 1-p_{jump} play according to classic gradient descent; if performance improved along the current direction of exploration, keep exploring along that direction; otherwise reverse the direction of exploration.
2) With p_{jump} probability, select randomly the attempts with uniform probability for the next configuration. If after the jump performance decreased by more than *maxLoss*, then revert to the *best* known configuration.

Further, in order to enhance stability and avoid useless oscillations once identified the optimal solution, if, after a configuration change, performance did not change by more than *min*Δ, we block the gradient descent exploration and allow only probabilistic jumps (to minimize the risk of getting stuck in sub-optimal configurations).

Concerning the settings of the p_{jump}, *maxLoss*, and *min*Δ, we set them respectively to 1%, 40% and 5%, which are typical values for this type of algorithms [28] and whose appropriateness in the considered context will be assessed in Section 7.

5 Merging the Learning Techniques

So far we have presented: 1) UCB to optimize the consumption of attempts upon capacity aborts (Section 4.2); and 2) GRAD to optimize the allocation of the budget of attempts (Section 4.3). We now present their integration in our algorithm called TUNER.

The concern with the integration in TUNER is that the two optimization strategies overlap slightly in their responsibilities. The advantage is that this allows to simultaneously optimize the configuration accurately for atomic blocks that sometimes exceed capacity in a deterministic way, whereas, in other scenarios, can execute efficiently using TSX. This may be, for instance, depending on the current state of shared memory, or some input parameter used in the atomic block. It is possible to achieve this because UCB shall decide to short-cut the attempts when capacity aborts happen, whereas GRAD can keep the attempts' budget high to allow successful TSX execution when capacity aborts are rare.

One problematic scenario arises when an atomic block is not suitable for execution in hardware: either GRAD can reduce the attempts to 0, or UCB can choose the **giveup** mode. However, we may be unlucky and get an inter-play of the two optimizers such that they affect each other and prevent convergence of the decisions.

To solve this problem with their integration, we create a hierarchy among the two optimizers, in which UCB can force GRAD to explore in some direction and avoid ping-pong optimizations between the two. For this, we create a rule that is activated when the attempts' budget is exhausted: in such event we trigger a random jump to force GRAD to explore in the direction that is most suitable according to UCB, that is, explore more attempts if the UCB belief is **stubborn** and less attempts otherwise.

We compute the extension of the random jump for GRAD (based on the direction decided by UCB), by taking into account information about the types of aborts incurred so far. Namely, we collect the number of aborts due to capacity (*ab-cap*) and due to other reasons (*ab-other*). Then, if UCB suggests exploring more attempts (i.e., UCB belief is **stubborn**), we choose the length of the jump, noted J, proportionally to the relative frequency of *ab-other*:

$$J = \frac{\textit{ab-cap}}{\textit{ab-cap} + \textit{ab-other}} \cdot (\textit{maxTries} - \textit{cur})$$

where *cur* is the current configuration of the budget of attempts and $\textit{maxtries} = 16$. If UCB is different from **stubborn**, the jump has negative direction, and length:

$$J = \frac{\textit{ab-other}}{\textit{ab-cap} + \textit{ab-other}} \cdot \textit{cur}$$

We now assess the efficiency of each of the optimization techniques alone, and their joint approach described above as TUNER. In this joint strategy we seek to understand if the two optimization techniques work well together: Fig. 5 shows the speedup of TUNER relatively to UCB and GRAD individually — we average the results across benchmarks since they yielded consistent results.

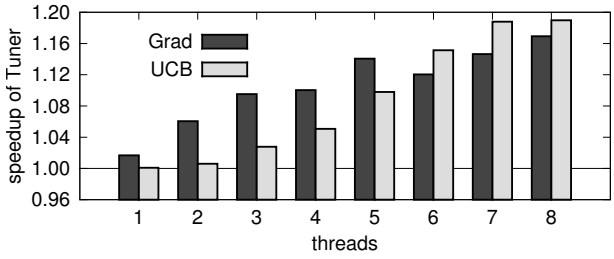


Figure 5: Geometric mean speedup of TUNER over UCB and GRAD across benchmarks and threads.

We can see from our experiments that the joint strategy provided average results that are always better than at least one of the approaches alone. More than that, for most cases TUNER improved over both individual strategies, which shows that employing them in synergy provides better results than the best approach alone. This is an encouraging result because tuning the attempts and dealing with capacity aborts is not entirely a disjoint concern. Overall, the results show that the joint approach yielded up to 20% improvement. Notice that each technique individually already improves over the baselines presented earlier, so any improvement when merged further reduces the gap with respect to the optimal result.

6 Integration in GCC

In this section we detail our implementation of TUNER, which we have integrated in the latest stable version of the Gnu C Compiler (GCC version 4.8.2), inside its *libitm* component. This component is responsible for implementing the C++ TM Specification [1] in GCC, which allows programmers to write atomic constructs in their programs that are compiled to calls to the TM runtime.

One important aspect of *libitm* is that it defines an interface that can be implemented by external libraries to plug in different TM runtimes and replace the implementations available inside GCC. Our initial expectation was that we could craft a TSX based runtime relying on TUNER as an external library. However, *libitm* does not completely delegate its work to such external library; it still keeps control on matters such as irrevocability (atomic blocks that cannot execute optimistically; e.g. those with I/O operations). This may cause performance loss because a single-lock TSX benefits from merging the irrevocability lock with the fallback lock. Furthermore, the choice of integrating TUNER into GCC allows achieving total transparency and maximum ease of use for the programmer.

We begin by laying out a high-level description of TUNER in Fig. 6. The flow starts every time a thread enters an atomic block. Since TUNER uses per atomic block statistics and configurations, we use the program counter as an identifier of the atomic block and retrieve the corresponding metadata kept by our algorithm. Every per block metadata is maintained in thread-local vari-

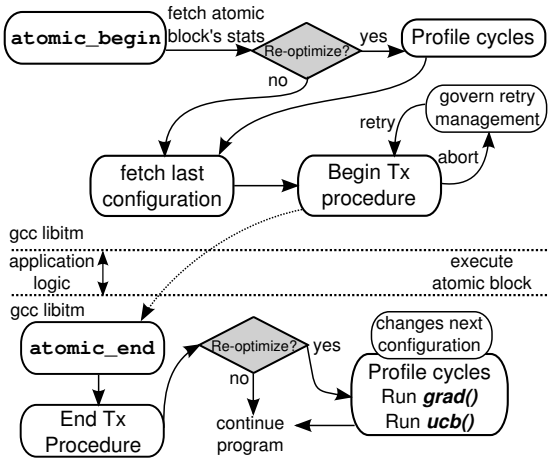


Figure 6: Workload-Oblivious tuning of TSX.

ables: hence threads perform self-tuning in an independent fashion. This has the advantage of avoiding synchronization and allowing threads to reach different configurations, which can be useful in case the various application threads are specialized to process different tasks (and generate different workloads).

After fetching the metadata, we check whether it is time to re-optimize the configuration for that atomic block. This condition is a result of the sampling that we use to profile the application. For this, we keep a counter of executions in the metadata of the atomic block (recall that it is thread local) so that we only re-optimize periodically. This classic technique allows to keep the overheads low without missing noticeable accuracy in the decisions taken [19, 29, 30]. Hence we place the check for re-optimization in the begin and end of the atomic block. In the negative case, we simply execute the atomic block with the last configuration set up for it and proceed without any extra logic or profiling.

In the case that we re-optimize, this enables profiling of the cycles that it takes to execute the atomic block. For this, we use the `RDTSC` instruction in x86, which we use as a lightweight profiling tool to measure the relative cost of executing the block in different configurations. After this we attempt to start the transaction itself, which is better described in Alg. 3. Lines 8-16 describe the retry management policy. During a re-optimization period, if the attempts' budget is exhausted, this triggers the forced random jump over GRAD according to the description of TUNER in Section 5 (line 9), before proceeding to the fallback path. Note also that upon a capacity abort we adequately reduce the available budget according to the belief of UCB set in the current configuration (line 13).

After the application executed the atomic block, it calls back to *libitm*, and TUNER executes the usual procedure to finish the transaction. After this, it checks whether it is re-optimizing the atomic block, and in the positive case it runs GRAD and UCB to adapt the configuration for the next executions. To do so, it uses

Algorithm 3 TUNER adaptive configuration.

```
 1: int ucbBelief ← ▷ last configuration used
 2: int attempts ← ▷ last configuration used
 3: if reoptimize() then
 4:     long initCycles ← obtainRDTSC()
 5: while is_locked(global-lock) do pause
 6: int status ← XBEGIN
 7: if status ≠ ok then
 8:     if attempts = 0 then
 9:         if reoptimize() then tuneAttempts(ucbBelief)
10:         acquire(global-lock)
11:     else
12:         if status = capacity then
13:             ▷ set attempts according to ucbBelief
14:         else if status = explicit ∨ status = other then
15:             attempts ← attempts - 1
16:         goto line 5
17: ▷ ...code to run in transaction
18: if attempts = 0 then
19:     release(global-lock)
20: else
21:     if is_locked(global-lock) then XABORT
22:     XEND
23: if reoptimize() then
24:     long totalCycles ← obtainRDTSC() - initCycles
25:     ucbBelief ← UCB(totalCycles)    ▷ rules of Section 4.2
26:     attempts ← GRAD(totalCycles)    ▷ rules of Section 4.3
```

the processor cycles consumed, and applies the rules described throughout Section 4 to configure the budget and consumption of attempts in the metadata of the atomic block. Similarly to other metrics, assessing performance via processor cycles is also subject to thread preemption, which may inflate the actual cost of executing the atomic block. We mitigate this by binding threads to logical cores, and evaluating scenarios with up to as many threads as logical cores, as more than those typically deteriorates performance anyway.

7 Evaluation

We now present our final set of experiments, in which we compare TUNER with the following baselines:

- GCC - corresponding to Alg. 1, which is the implementation available in *libitm* in GCC 4.8.2.
- HEURISTIC - corresponding to Alg. 2 for which we tried to use static heuristics to better tune TSX.
- ADAPTIVELOCKS - proposed to decide between locks and TM for atomic blocks [29]; an analytical model is used and fed with statistics sampled at run-time (similarly to TUNER). We adapted their code (using CIL) to our environment integrated in GCC.
- TUNER - our contribution described in Alg. 3.
- Best Variant - an upper bound on the best result possible, obtained by picking the best settings of the considered parameters among all possible configurations for each benchmark and degree of parallelism. As such, this alternative does not correspond to a real tuning algorithm, but rather to an optimal, static configuration.

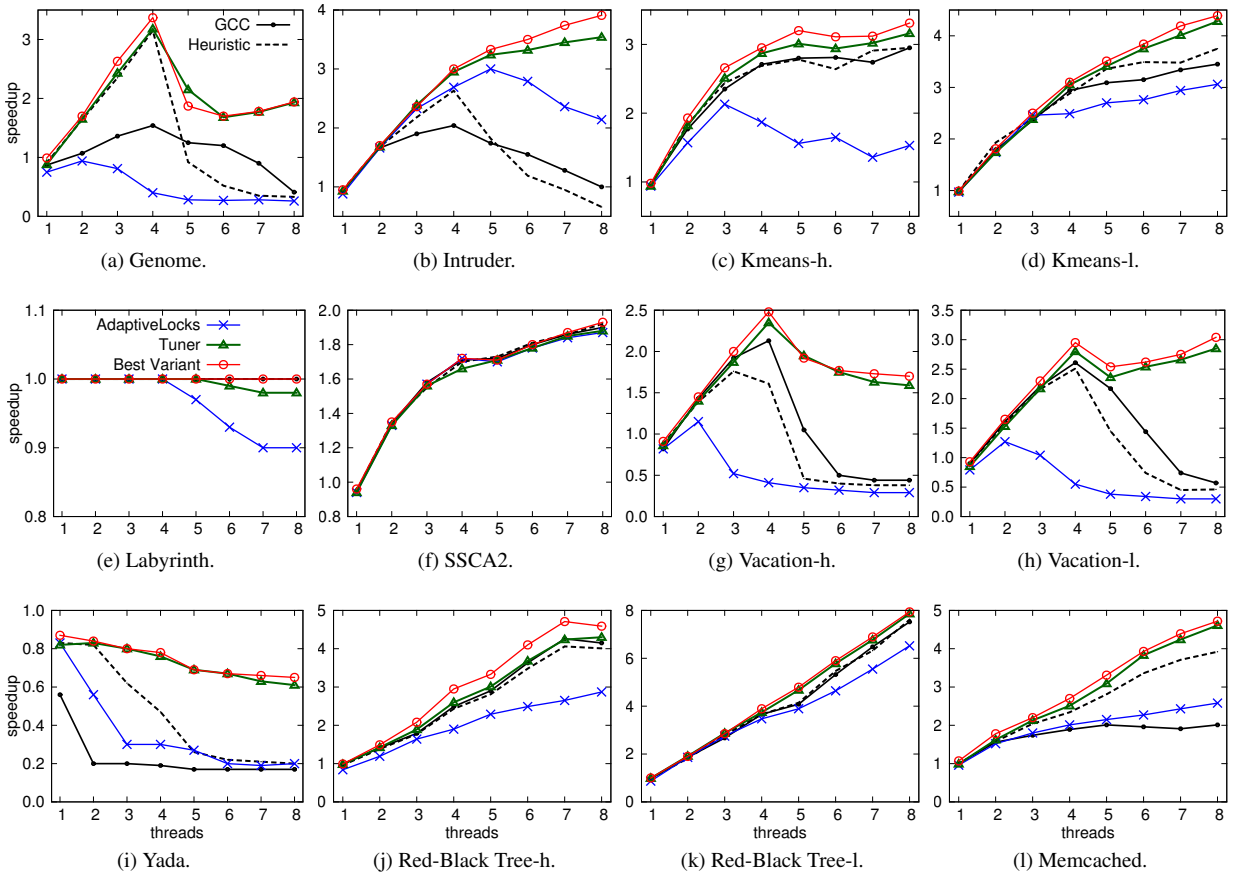
We used the standard parameters for the STAMP benchmarks and show workloads for low and high contention when available. For the red-black tree we used two workloads: low contention with 1 million items and 10% transactions inserting/removing items whereas the rest only performs fetch operations; and high contention with 1 thousand items and 90% transactions mutating the tree. For these benchmarks we present the speedup of each variant relatively to a sequential, non-synchronized execution. Finally we use a balanced workload for Memcached, configured with 50% gets and sets, and always set an equal number of worker and client threads. For this, we used the memslap tool in a similar fashion to [27]. In Memcached there is no sequential execution since there is always concurrency due to maintenance threads. As such, we use speedups relative to GCC at 1 thread, by having each execution last 60 seconds and measuring its throughput.

In general this set of experiments (Fig. 7) shows a typical gap in performance between the static configurations and the best possible variant. This gap is usually more noticeable as the concurrency degree increases — as we can see for instance in Kmeans-l — which is expected, since that is when the configuration parameters matter most to decide when it is profitable to insist on the hardware transactions of TSX. In short, these gaps in performance between the static alternatives and the best variant possible is exactly the room of improvement that we try to explore with TUNER in this paper.

In fact, TUNER is able to achieve performance improvements in all benchmarks with the exception of Labyrinth and SSCA2, in which it yields the same performance as the static approaches. In Labyrinth transactions are always too large to execute in hardware, and the benchmark executes about five hundred such large operations, which means the length of the transaction dominates the benchmark and no noticeable performance changes exist with regard to different configurations that do not insist too much on the hardware. In SSCA2 there is little time spent in atomic blocks, and some barriers synchronizing the phases of the workload, resulting in bottlenecks that are independent of the atomic blocks and that make all configurations perform similarly.

Table 2: Geometric mean speedup (across benchmarks) of each algorithm relatively to sequential executions.

Algorithms	threads			
	2	4	6	8
GCC	1.25	1.74	1.51	1.29
HEURISTIC	1.46	2.01	1.37	1.28
ADAPTIVELOCKS	1.26	1.19	1.10	1.11
TUNER	1.46	2.25	2.34	2.54
Best Variant	1.51	2.35	2.41	2.66

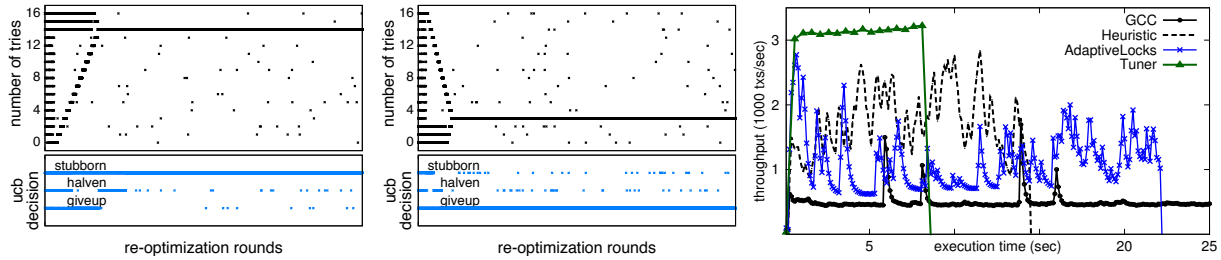


(a) Genome. (b) Intruder. (c) Kmeans-h. (d) Kmeans-l.

(e) Labyrinth. (f) SSCA2. (g) Vacation-h. (h) Vacation-l.

(i) Yada. (j) Red-Black Tree-h. (k) Red-Black Tree-l. (l) Memcached.

Figure 7: Speedup of different approaches to tune TSX relative to sequential executions in all benchmarks.



(a) Yada atomic block A. (b) Yada atomic block B. (c) Throughput in Yada with 8 threads.

Figure 8: Exploration and adaptation of TUNER on two different atomic blocks (left) and global throughput (right).

For all other benchmarks and workloads we find TUNER typically close to the best variant. Table 2 summarizes our findings across this extensive set of benchmarks: at 8 threads, the maximum hardware parallelism available for TSX, TUNER obtains an approximate improvement of 2× over GCC, HEURISTIC and ADAPTIVELOCKS, while remaining roughly 5% off the optimal solution identified by means of the exhaustive, offline exploration of the parameters' space.

We also note that the current hardware is limited in terms of hardware parallelism: in fact, some times going over 4 threads is not profitable as hyper-threading is not beneficial due to the extra pressure on L1 caches [13]. This, however, is an issue that has been tackled by related work (e.g. [9]) and whose importance shall be relatively diminished by the availability of new hardware to be released with more cores and without hyper-threading.

Finally, manual profiling and inspection revealed that TUNER consistently converged to configurations similar to the ones that performed best in our extensive offline testing. We present an example of the adaptation performed by TUNER in Fig. 8 in the Yada benchmark (we show the adaptation of one thread among 8 running concurrently). There, we can see the configuration of two atomic blocks being re-optimized, and converging to two drastically different configurations: the left

block executes efficiently with TSX whereas the right one does not. This illustrates two advantages of our solution: 1) the adaptation allows heterogeneous threads and atomic blocks to converge to different configurations; 2) an atomic block, such as that in Fig. 8b, can still insist moderately on using TSX as long as capacity aborts do not occur, but react quickly in case they appear. As a result, we can see the significant performance increase of our solution, depicted in Fig. 8c with the throughput of each solution as the benchmark executes. We highlight the steadiness of TUNER against the irregular and spiky performance of the static solutions that can, at best, only fit the workload for limited time periods.

8 Related Work

Transactional Memory was initially proposed as an extension to multi-cores' cache coherence protocols [15]. Due to the inaccessibility of rapidly prototyping in such environment, researchers resorted to software implementations (STM) to advance the state of the art [14]. These STMs require instrumenting the code (either manually or compiler-assisted) to invoke the TM software runtime in every read and write to shared memory. As a result, STMs impose some overhead in sequential executions, but they are efficient enough to pay off with some meaningful degree of parallelism [11, 12].

Recently, implementations in hardware (HTM) became available in commercial processors delivered by major industry players. Beyond Intel, IBM also provided support for HTM [16], in processors mostly used on high performance computing. We only had access to an Intel machine, but we believe the techniques described here should also be applicable to IBM's HTMs due to their similar nature. Furthermore, the mainstream nature of Intel processors increases significantly the relevance of works, like this, aimed to optimize its performance.

We are not aware of any work that self-tunes TSX (or any similar HTM). Works that studied TSX's performance [17, 31] obtained promising results, but relied on manual tuning and provided only brief textual insights as to how the decisions to configure it were taken.

Given the best-effort nature of this first generation of HTM in commodity processors, it is desirable to consider an efficient solution for the fallback path in software. One interesting idea is to use STMs combined with HTM (HybridTMs), so that transactions that are not successful in hardware can execute in software without preventing all concurrency, as is the case of pessimistic coarse locking-based schemes. In this scope, some work has obtained promising results with a simulator for HTM support from AMD [6,23]. However, there are no official plans to integrate AMD's proposal [5] in a commercial processor. More recently, the Reduced TM technique has been proposed for Intel TSX [20], but it was only evaluated in an emulated environment. In this paper we take a step back, and try to optimize as much as possible the HTM usage, before trying to integrate it with more complex fallback paths than that of a global lock. We believe that for most common situations this should be enough, as evidenced by the recent application of Intel TSX in SAP Hana database [17].

Additionally, there have been other proposals for adaptation in TMs in software. Adaptive Locks [29], VOTM [19], and Dynamic Pessimism [26] adapt between optimistic (with STM) and pessimistic (via locking) execution of atomic blocks. Unfortunately, these works do not map directly to best-effort HTMs, as we showed in our evaluation (by considering Adaptive Locks, as the authors kindly provided us with their code). More complex adaptation schemes have been proposed to self-tune the choice between different STM algorithms in AutoTM [30]. The main drawback of these kind of works, with regard to the HTM setting studied in this paper, is that these self-tuning proposals require knowledge that is not available from the HTM support that we have, such as the footprint of transactions (their read- and write-sets). That is, unless we instrument reads and writes to obtain it, which would defeat the purpose of HTM to lower the overhead of TM over STMs.

9 Conclusions

In this paper we studied the performance of the hardware support available via Intel TSX in the latest generation of x86 Core processors. This interface allows some flexibility in the definition of the software fallback mechanism triggered upon transactional aborts, with regard to when and how to give up executing hardware transactions. We showed that no single configuration of the software fallback can perform efficiently in every workload and application. Motivated by these findings, we presented TUNER, a novel self-tuning approach that combines reinforcement learning techniques and gradient-descent exploration-based algorithms to self-tune TSX in a workload-oblivious manner. This means that TUNER does not require a priori knowledge of the application, and executes fully online, based on the feedback on system's performance gathered by means of lightweight profiling techniques. We integrated TUNER in the well known GCC compiler, achieving total transparency for the programmer. We evaluated our solution against available alternatives using a comprehensive set of applications showing consistent gains.

Acknowledgements: We thank Konrad Lai, Ravi Rajwar and Richard Yoo from Intel for the insightful discussions about TSX during the conduct of this research.

This work was supported by national funds through FCT (Fundação para a Ciência e Tecnologia) under project PEst-OE/EEI/LA0021/2013 and by project GreenTM EXPL/EEI-ESS/0361/2013.

References

[1] ADL-TABATABAI, A., SHPEISMAN, T., AND GOTTSCHLICH, J. Draft Specification of Transactional Language Constructs for C++. In *Intel* (2012).

[2] AFEK, Y., LEVY, A., AND MORRISON, A. Programming with Hardware Lock Elision. In *Proceedings of the 18th Symposium on Principles and Practice of Parallel Programming (PPoPP)* (2013), pp. 295–296.

[3] AUER, P., CESA-BIANCHI, N., AND FISCHER, P. Finite-time Analysis of the Multiarmed Bandit Problem. *Mach. Learn. 47*, 2-3 (May 2002), 235–256.

[4] CALCIU, I., SHPEISMAN, T., POKAM, G., AND HERLIHY, M. Improved Single Global Lock Fallback for Best-effort Hardware Transactional Memory. In *9th Workshop on Transactional Computing (TRANSACT)* (2014).

[5] CHRISTIE ET AL., D. Evaluation of AMD's Advanced Synchronization Facility Within a Complete Transactional Memory Stack. In *Proceedings of the 5th EuroSys* (2010), pp. 27–40.

[6] DALESSANDRO ET AL., L. Hybrid NOrec: A Case Study in the Effectiveness of Best Effort Hardware Transactional Memory. In *Proceedings of the 16th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2011), pp. 39–52.

[7] DICE, D., LEV, Y., MOIR, M., AND NUSSBAUM, D. Early Experience with a Commercial Hardware Transactional Memory Implementation. In *Proceedings of the 14th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2009), pp. 157–168.

[8] DICE ET AL., D. Applications of the Adaptive Transactional Memory Test Platform. In *3rd Workshop on Transactional Computing (TRANSACT)* (2008).

[9] DIDONA, D., FELBER, P., HARMANCI, D., ROMANO, P., AND SCHENKER, J. Identifying the Optimal Level of Parallelism in Transactional Memory Applications. In *Proceedings of the 1st Inernational Conference on Networked Systems (NETYS)* (2013), pp. 233–247.

[10] DIDONA, D., ROMANO, P., PELUSO, S., AND QUAGLIA, F. Transactional auto scaler: elastic scaling of in-memory transactional data grids. In *Proceedings of the International Conference on Autonomic Computing (ICAC)* (2012), pp. 125–134.

[11] DIEGUES, N., AND CACHOPO, J. Practical Parallel Nesting for Software Transactional Memory. In *Proceedings of the 27th International Symposium on Distributed Computing (DISC)* (2013), pp. 149–163.

[12] DIEGUES, N., AND ROMANO, P. Time-warp: Lightweight Abort Minimization in Transactional Memory. In *Proceedings of the 19th Symposium on Principles and Practice of Parallel Programming (PPoPP)* (2014), pp. 167–178.

[13] DIEGUES, N., ROMANO, P., AND RODRIGUES, L. Virtues and Limitations of Commodity Hardware Transactional Memory. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation Techniques (PACT)* (2014).

[14] DRAGOJEVIĆ, A., FELBER, P., GRAMOLI, V., AND GUERRAOUI, R. Why STM Can Be More Than a Research Toy. *Commun. ACM 54*, 4 (Apr. 2011), 70–77.

[15] HERLIHY, M., AND MOSS, J. E. B. Transactional Memory: Architectural Support for Lock-free Data Structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA)* (1993), pp. 289–300.

[16] JACOBI, C., SLEGEL, T., AND GREINER, D. Transactional Memory Architecture and Implementation for IBM System Z. In *Proceedings of the 45th Symposium on Microarchitecture (MICRO)* (2012), pp. 25–36.

[17] KARNAGEL ET AL., T. Improving In-Memory Database Index Performance with Intel TSX. In *Proceedings of the 20th Symposium on High Performance Computer Architecture (HPCA)* (2014).

[18] KLEEN, A. Scaling Existing Lock-based Applications with Lock Elision. *Queue 12*, 1 (Jan. 2014), 20:20–20:27.

[19] LEUNG, K.-C., CHEN, Y., AND HUANG, Z. Restricted admission control in view-oriented transactional memory. *The Journal of Supercomputing 63*, 2 (2013), 348–366.

[20] MATVEEV, A., AND SHAVIT, N. Reduced Hardware Transactions: A New Approach to Hybrid Transactional Memory. In *Proceedings of the 25th Symposium on Parallelism in Algorithms and Architectures (SPAA)* (2013), pp. 11–22.

[21] MINH, C. C., CHUNG, J., KOZYRAKIS, C., AND OLUKOTUN, K. STAMP: Stanford Transactional Applications for Multi-Processing. In *Proceedings of the International Symposium on Workload Characterization (IISWC)* (2008), pp. 35–46.

[22] PANKRATIUS, V., AND ADL-TABATABAI, A. A Study of Transactional Memory vs. Locks in Practice. In *Proceedings of the 23rd Symposium on Parallelism in Algorithms and Architectures (SPAA)* (2011), pp. 43–52.

[23] RIEGEL, T., MARLIER, P., NOWACK, M., FELBER, P., AND FETZER, C. Optimizing Hybrid Transactional Memory: The Importance of Nonspeculative Operations. In *Proceedings of the 23rd Symposium on Parallelism in Algorithms and Architectures (SPAA)* (2011), pp. 53–64.

[24] RUGHETTI, D., DI SANZO, P., CICIANI, B., AND QUAGLIA, F. Machine Learning-Based Self-Adjusting Concurrency in Software Transactional Memory Systems. In *Proceedings of the International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)* (2012), pp. 278–285.

[25] RUSSELL, S. J., AND NORVIG, P. *Artificial Intelligence: A Modern Approach*, 3rd ed. Prentice Hall, 2009.

[26] SONMEZ, N., HARRIS, T., CRISTAL, A., UNSAL, O., AND VALERO, M. Taking the heat off transactions: Dynamic selection of pessimistic concurrency control. In *Proceedings of the International Symposium on Parallel Distributed Processing (IPDPS)* (2009), pp. 1–10.

[27] SPEAR, M., VYAS, T., AND RUAN, WENJIA LIU, Y. Transactionalizing Legacy Code: An Experience Report Using GCC and Memcached. In *Proceedings of the 19th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2014), pp. 399–412.

[28] SUTTON, R. S., AND BARTO, A. G. *Introduction to Reinforcement Learning*, 1st ed. MIT Press, Cambridge, MA, USA, 1998.

[29] USUI, T., BEHRENDS, R., EVANS, J., AND SMARAGDAKIS, Y. Adaptive Locks: Combining Transactions and Locks for Efficient Concurrency. In *Proceedings of the 18th Conference on Parallel Architectures and Compilation Techniques (PACT)* (2009), pp. 3–14.

[30] WANG, Q., KULKARNI, S., CAVAZOS, J., AND SPEAR, M. A Transactional Memory with Automatic Performance Tuning. *ACM Trans. Archit. Code Optim. 8*, 4 (Jan. 2012), 54:1–54:23.

[31] YOO, R. M., HUGHES, C. J., LAI, K., AND RAJWAR, R. Performance Evaluation of Intel Transactional Synchronization Extensions for High-performance Computing. In *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (2013), pp. 1–11.

CloudPowerCap: Integrating Power Budget and Resource Management across a Virtualized Server Cluster

Yong Fu
Washington University in St. Louis

Anne Holler
VMware

Chenyang Lu
Washington University in St. Louis

Abstract

In many data centers, server racks are highly underutilized due to maintaining the sum of the server nameplate power below the power provisioned to the rack. The root cause of this rack underutilization is that the server nameplate power is often much higher than can be reached in practice. Although statically setting per-host power caps can ensure the sum of the servers' maximum power draw does not exceed the rack's provisioned power, it burdens the data center operator with managing the rack power budget across the hosts. In this paper we present CloudPowerCap, a practical and scalable solution for power cap management in a virtualized cluster. CloudPowerCap, closely integrated with a cloud resource management system, dynamically adjusts the per-host power caps for hosts in the cluster to respect not only the rack power budget but also the resource management system's constraints and objectives. Evaluation based on an industrial cloud simulator demonstrates effectiveness and efficacy of CloudPowerCap.

1 Introduction

In many datacenters, server racks are as much as 40 percent underutilized [7]. Rack slots are intentionally left empty to keep the sum of the servers' nameplate power below the power provisioned to the rack, and the servers that are placed in the rack cannot make full use of the rack's provisioned power. The root cause of this rack underutilization is that a server's peak power consumption is in practice often significantly lower than its nameplate power [5]. This server rack underutilization can incur substantial costs. In hosting facilities charging a fixed price per rack, which includes a power charge that assumes the rack's provisioned power is fully consumed, paying a 40 percent overhead for rack underutilization is nontrivial. And in a private datacenter, the amortized capital costs for the infrastructure to deliver both the racks' provisioned power and the cooling capacity to handle the racks' fully populated state comprises 18 percent of a datacenter's total monthly costs [10]. If that infrastructure is 40 percent underutilized, then 7 percent of the data center's monthly costs are wasted for this reason.

Due to the significant cost of rack underutilization, major server vendors are now shipping support for per-host power caps, which provide a hardware or firmware-enforced limit on the amount of power that the server can draw [12, 4, 13]. These caps work by changing processor power states [11] or by using processor clock throttling, which is effective since the processor is the largest consumer of power in a server and its activity is highly correlated with the server's dynamic power consumption [5, 12]. Using per-host power caps, data center operators can set the caps on the servers in the rack to ensure that the sum of those caps does not exceed the rack's provisioned power. While this approach improves rack utilization, it burdens the operator with manually managing the rack power budget allocated to each host in a rack. In addition, it does not lend itself to flexible allocation of power to handle workload spikes or to respond to the addition or removal of a rack's powered-on server capacity.

Many data centers use their racked servers to run virtual machines (VMs). Several research projects have investigated power cap management for virtualized infrastructure [21, 18, 19, 16, 28, 3]. While this prior work has considered some aspects of VM Quality-of-Service (QoS) in allocating the power budget, it has not explored operating in a coordinated fashion with a comprehensive resource management system for virtualized infrastructure. Sophisticated cloud resource management systems such as VMware Distributed Resource Scheduler (DRS) support admission-controlled resource reservations, resource entitlements based fair-share scheduling, load-balancing to maintain resource headroom for demand bursts, and respect for constraints to handle user's business rules [9]. However the operations of cloud resource management systems aforementioned can be compromised if coordination is not carefully considered in design of the integrated power cap management system. For example, changing host power cap naively may affect resource to VMs, impacting end-users' Service-Level Agreements (SLAs), fairness, robustness and peak performance. Similarly, if the resource management system consolidates VMs and powers off unneeded hosts to save power, the host power cap setting system needs to be aware of that activity or it may cause the power budget to be inefficiently allocated to hosts, impacting the amount of powered-on computing capacity available for a given power budget.

This paper presents CloudPowerCap, an autonomic computing approach to power budget management in a virtualized environment. CloudPowerCap manages the power budget for a cluster of virtualized servers, dynamically adjusting the per-host power caps for servers in the cluster. It allocates the power budget in close coordination with a cloud resource management system, operating in a manner consistent with the system's resource management constraints and goal of ensuring VMs receive

the resources to which they are entitled. To facilitate interoperability between power cap and resource management, CloudPowerCap maps a servers power cap to its CPU capacity and coordinates with the resource management system through well defined interfaces and protocols. The integration of power cap and resource management results in the following novel capabilities in cloud management.

- **Constraint satisfaction via power cap reallocation:** Dynamic power cap reallocation enhances the system's capability to satisfy VM constraints, including resource reservations and business rules.
- **Power-cap-based entitlement balancing:** Power cap redistribution provides an efficient mechanism to achieve entitlement balancing among servers. Power-cap-based entitlement balancing can reduce or eliminate *real* migrations of VMs and associated overhead.
- **Power cap redistribution for power management:** CloudPowerCap can redistribute power caps among servers to handle server power-off/on state changes caused by dynamic power management. Power cap redistribution reallocates the power budget freed up by powered-off hosts, while reclaiming budget to power-on those hosts when needed.

We have implemented and integrated CloudPowerCap with VMware Distributed Resource Scheduler (DRS). Evaluation based on an industrial cloud simulator demonstrated the efficacy of integrated power budget and resource management in virtualized server clusters.

2 Motivation

In this section, we motivate the problem CloudPowerCap is intended to solve. We first describe the power model mapping a host's power cap to its CPU capacity, which enables CloudPowerCap to integrate power cap management with resource management in a coordinated fashion. We next discuss some trade-offs in managing a rack power budget. After a brief introduction of the resource management model, we then provide several examples of the value of combining dynamic rack power budget management with a cloud resource management system.

2.1 CloudPowerCap Power Model

The power model adopted by CloudPowerCap maps the power cap of the host to the CPU capacity of the host, which is in turn managed by a resource management system directly. A host's power consumption $P_{consumed}$ is commonly estimated by its CPU utilization U and the idle P_{idle} and peak P_{peak} power consumption of the host via a linear function, which is validated by real-world workloads in previous measurements and analysis [17, 5],

$$P_{consumed} = P_{idle} + (P_{peak} - P_{idle})U. \quad (1)$$

The power P_{idle} represents the power consumption of the host when the CPU is idle. P_{idle} intentionally includes the power consumption of the non-CPU components, such as networking and memory, whose power draw currently does not vary significantly with utilization. We note that in enterprise virtualized data centers, either the local disks are kept busy enough not to spin down or (more typically) shared storage is employed. The power P_{peak} represents the power consumption of the host when the CPU is 100% utilized at its maximum CPU capacity C_{peak}, with the CPU utilization U expressed as a fraction of the maximum capacity.

Equation (1) is an upper-bound estimation of $P_{consumed}$ if a host power management technology such as dynamic voltage and frequency scaling (DVFS) is used since DVFS can deliver amount of CPU capacity at a lower power consumption. For example, DVFS could deliver the equivalent of 50 percent utilization of a 2 GHz processor at lower power consumption by running the processor at 1 GHz with 100 percent utilization. Computing $P_{consumed}$ as an upper bound is desirable for the resource management use case, to ensure sufficient power budget for worst case.

Considering upper-bound power estimation from Equation (1), for a host power cap P_{cap} set below P_{peak}, we could solve for the *lower-bound* of the CPU capacity C_{capped} corresponding to P_{cap}, i.e., the host's effective CPU capacity limit which we refer to as its *power-capped capacity*. In this case, we rewrite Equation (1) as:

$$P_{cap} = P_{idle} + (P_{peak} - P_{idle})(C_{capped}/C_{peak}). \quad (2)$$

and then solve for C_{capped} as:

$$C_{capped} = C_{peak}(P_{cap} - P_{idle})/(P_{peak} - P_{idle}). \quad (3)$$

2.2 Managing a Rack Power Budget

To illustrate some trade-offs in managing a rack power budget, we consider the case of a rack with a budget of 8 KWatt, to be populated by a set of servers. Each server has 34.8 GHz CPU capacity comprising 12 CPUs, each running at 2.9 GHz, along with the other parameters shown in Table 1.

CPU	Memory	Nameplate	Peak	Idle
34.8 GHz	96 GB	400 W	320 W	160 W

Table 1: The configuration of the server in the rack.

Given the power model presented in the previous section and the servers in Table 1, the rack's 8 KWatt power budget can accommodate various deployments including those shown in Table 2. Based on the 400 Watts nameplate power, only 20 servers can be placed in the rack. Instead setting each server's power cap to its peak attainable power draw of 320 Watts allows 25 percent

more servers to be placed in the rack. This choice maximizes the amount of CPU capacity available for the rack power budget, since it best amortizes the overhead of the servers' powered-on idle power consumption. However, if memory may sometimes become the more constrained resource, setting each server's power cap at 250 Watts to allow 32 hosts to be placed in the rack significantly increases the memory available for the given power budget. By dynamically managing the host power caps of the servers in Table 1, trade-offs between CPU and memory capacity are illustrated in Table 2.

Power Cap(W)	Servers	CPU		Memory	
		Total(GHz)	Ratio	Total(GB)	Ratio
400	20	696	1.00	1920	1.00
320	25	870	1.25	2400	1.25
285	28	761	1.09	2688	1.40
250	32	626	0.90	3072	1.60

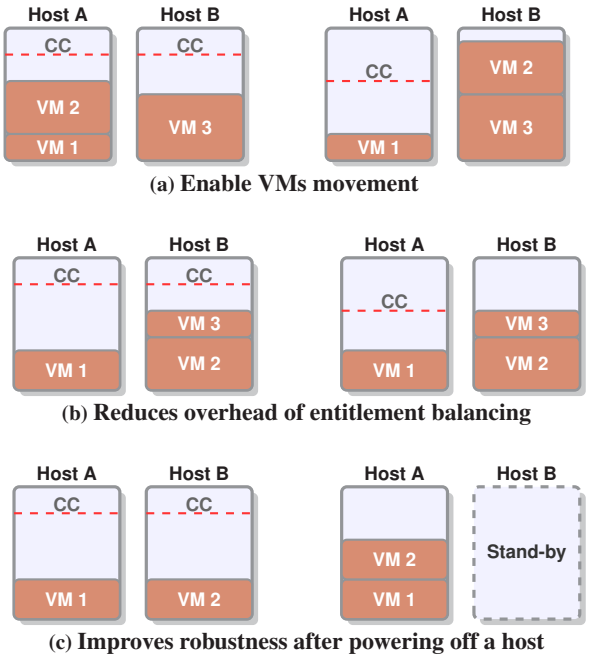
Table 2: Trade-offs between CPU and memory with different power caps

2.3 Powercap Distribution Examples

The cloud resource management system with which CloudPowerCap is designed to interoperate computes each VM's entitled resources and handles the ongoing location of VMs on hosts so that the VMs' entitlements can be delivered while respecting constraints, providing fair resource allocation by entitlement balancing, and optionally reducing power consumption.

In this section we use several scenarios to illustrate how CloudPowerCap can redistribute host power caps to support cloud resource management, including enabling VM migration to correct constraint violations, providing spare resource headroom for robustness in handling bursts, and avoiding migrations during entitlement balancing. In these scenarios, we assume a simple example of a cluster with two hosts. Each host has an uncapped capacity of 2x3GHz (two CPUs, each with a 3GHz capacity) with a corresponding peak power consumption of 600W (values chosen for ease of presentation).

Enforcing constraints: Host power caps should be redistributed when VMs are placed initially or relocated, if necessary to allow constraints to be respected or constraint violations to be corrected. For example, a cloud resource management system would move VM(s) from a host violating affinity constraints to a target host with sufficient capacity. However, in the case of static power cap management, this VM movement may not be feasible because of a mismatch between the VM reservations and the host capacity. As shown in Figure 1a, host A and B have the same power cap of 480 W, which corresponds to a power-capped capacity of 4.8 GHz. Host A runs two VMs, VM 1 with reservation 2.4 GHz and VM 2 with reservation 1.2 GHz. And host B runs only one 3 GHz reservation VM. When VM 2 needs to be colocated with VM 3 due to a new VM-VM affinity rule between the two VMs, no target host in the cluster has sufficient power-capped capacity to respect their combined reservations. However, if CloudPowerCap redistributes the power caps of host A and B as 3.6 GHz and 6 GHz respectively, then VM 2 can successfully be moved by the cloud resource management system to host B to resolve the rule violation in the cluster. Note that host A's capacity cannot be reduced below 3.6 GHz until VM 1's migration to host B is complete or else the reservations on host A would be violated.



(a) Enable VMs movement

(b) Reduces overhead of entitlement balancing

(c) Improves robustness after powering off a host

Figure 1: Power cap distribution scenarios. Left-hand figures correspond to hosts' status before distribution; right-hand figures show hosts' status after. Power-capped capacity is not shown when the power cap of the host equals its peak power. (CC: Power-capped capacity)

Enhancing robustness to demand bursts: Even when VM moves do not require changes in the host power caps, redistributing the power caps can still benefit the robustness of the hosts to handling VM demand bursts. For example, suppose as in the previous example that VM 1 needs to move from host A to host B because of a rule. In this case, a cloud resource management system can move VM 1 to host B while respecting the VMs' reservations. However, after the migration of VM 1, the *headroom* between the power capped capacity and VMs' reservations is only 0.6 GHz on host B, compared with 2.4 GHz on host A. Hence, host B can only accommodate as high as a

15% workload burst without hitting the power cap while host A can accommodate 100%, that is, host B is more likely to introduce a performance bottleneck than host A. To handle this imbalance of robustness between the two hosts, CloudPowerCap can redistribute the power caps of host A and B as 3.6 GHz and 6 GHz respectively. Now both hosts have essentially the same robustness in term of *headroom* to accommodate workload bursts.

Reduce overhead of VM migration: Before entitlement balancing, power caps should be redistributed to reduce the need for VM migrations. Load balancing of the resources to which the VMs on a host are entitled is a core component of cloud resource management since it can avoid performance bottlenecks and improve system-wide throughput. However, some migrations of VMs for load balancing are unnecessary. As shown in Fig 1b, the VM on Host A has an entitlement of 1.8 GHz while the VMs on host B have a total entitlement of 3.6 GHz. The difference in entitlements between host A and B are high enough to trigger entitlement balancing, in which VM 3 is moved from host B to host A. After entitlement balancing, host A and B have entitlements of 3 GHz and 2.4 GHz respectively, that is, the workloads of both hosts are more balanced. However, VM migration has an overhead related to copying the VM's CPU context and memory content between the hosts involved, inducing some latency [23]. In contrast, changing a host power cap only involves issuing a simple baseboard management system command which completes in less than one millisecond [12]. Hence, instead of entitlement balancing, CloudPowerCap can perform the cheaper action of redistributing the power caps of hosts A and B, increasing host B's power capped capacity to 6 GHz after decreasing host A's power capped capacity to 3.6 GHz, which also results in more balanced entitlements for host A and B. In general, the redistribution of power caps before entitlement balancing, called *powercap based entitlement balancing*, can reduce or eliminate the number of VM migrations for load balancing and the overhead of the migrations.

Adapting to host power on/off: Power caps should be redistributed when cloud resource management powers on/off host(s) to improve cluster efficiency. A cloud resource management system detects when there is ongoing under-utilization of cluster host resources leading to *power-inefficiency* due to the high host idle power consumption, and it consolidates workloads onto fewer hosts and powers the excess hosts off. In the example shown in Figure 1c, host B can be powered off after VM 2 is migrated to host A. However, after host B is powered-off, it does not consume power and hence does not need its power cap. Therefore the utilization of host A can be increased due to migrated VM 2, which impacts the capacity *headroom* of host A. Power cap redistribution after powering off host B can increase the power cap of host A to 6 GHz, allowing the *headroom* of host A to increase to 3 GHz and hence increase system robustness and reduce the likelihood of resource throttling. Similarly, powercap redistribution can improve robustness when resource management powers on hosts.

3 CloudPowerCap Design

CloudPowerCap is designed to provide power budget management to existing resource management systems, in such a way as to support and reinforce such systems' design and operation. Such resource management systems are designed to satisfy VMs' resource entitlements subject to a set of constraints, while providing balanced headroom for demand increases and, optionally, reduced power consumption. CloudPowerCap improves the operation of resource management systems, via power cap allocation targeted to their operation.

Existing resource management systems typically involve nontrivial complexity. Fundamentally reimplementing them to handle hosts of varying capacity due to power caps would be difficult and the benefit of doing so is unclear, given the coarse-grained scales at which cloud resource management systems operate. In CloudPowerCap, we take the practical approach of introducing power budget management as a separate manager that coordinates with an existing resource management system such that the existing system works on hosts of *fixed* capacity, with specific points at which that capacity may be modified by CloudPowerCap in accordance with the existing system's operational phase. Our approach therefore enhances modularity by separating power cap and resource management, while coordinating them effectively through well defined interfaces and protocols, as described below. Due to practical and modular design, CloudPowerCap is the same scalablity of the clous resource management it integrated with.

Since the aim of CloudPowerCap is to enforce the cluster power budget while dynamically managing hosts' power caps by closely coordinating with the cloud resource management system, CloudPowerCap consists of three components, as shown in Figure 2, corresponding to the three major functions of the cloud resource management system. The three components, corresponding to main components in DRS, execute step by step and work on two-way interaction with components in DRS.

Powercap Allocation: During the powercap allocation phase, potential resource management constraint correction moves may require redistribution of host power caps. Because CloudPowerCap can redistribute the host power caps, the cloud resource management system is able to correct more constraint violations than would be possible with statically-set host power caps.

Powercap-based Entitlement Balancing: If the resource management system detects entitlement imbalance over the user-set threshold, powercap based entitlement balancing first tries to reduce the imbalance, by redistributing power caps without actually migrating VMs

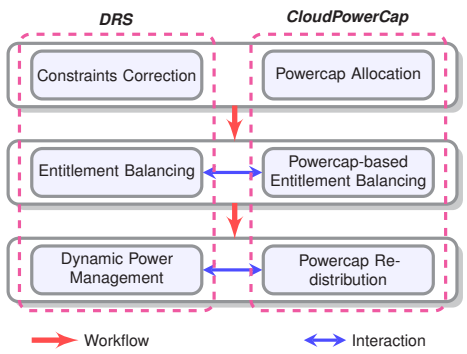


Figure 2: Structure and two-way interaction of CloudPowerCap working with DRS and DPM.

between hosts. This is valuable because redistributing power caps, which is less than 1 millisecond [12], is much faster than VM migration [26]. Powercap-based entitlement balancing may not be able to fully address imbalance due to inherent physical host capacity limits. If powercap balancing cannot reduce the imbalance below the imbalance threshold, the resource management entitlement balancing can address the remaining imbalance by actual VM migration.

Powercap Redistribution: If the resource management system powers on a host to match a change in workload demands or other requirements, CloudPowerCap performs a two-pass power cap redistribution. First it attempts to re-allocate sufficient power cap for that host to power-on. If that is successful and if the system selects the host in question after its power-on evaluation, then CloudPowerCap redistributes the cluster power cap across the updated hosts, to address any unfairness in the resulting power cap distribution. Powered-off host powercap redistribution improves the availability of power for the remaining powered-on hosts, allowing them to be more responsive to workload bursts. And when powered-on hosts can handle bursts without powering on additional hosts, this redistribution improves power efficiency. We note that the effect of off/on a host due to fault and recovery by high availability (HA) is similar to powering on/off hosts by a power management system. Hence, powercap redistribution can be adapted to work with HA.

4 CloudPowerCap Implementation

We implemented CloudPowerCap to work with the VMware Distributed Resource Scheduler (DRS) [24] along with its optional Distributed Power Management (DPM) [25] feature. CloudPowerCap could also complement some other distributed resource management systems for virtualization environments. In this section, we first present an overview of DRS and then detail the design of each CloudPowerCap component and its interaction with its corresponding DRS component. Due to space restriction, the implementation of Powercap redistribution is not presented in this paper. Please refer to the technical report for details [6].

4.1 DRS Overview

VMware DRS performs resource management for a cluster of ESX hypervisor hosts. By default, DRS is invoked every five minutes. At the beginning of each DRS invocation, DRS runs a phase to generate recommendations to correct any cluster constraint violations by migrating VMs between hosts. Examples of such corrections include evacuating hosts that the user has requested to enter maintenance or standby mode and ensuring VMs respect user-defined affinity and anti-affinity business rules. DRS next performs entitlement balancing. The load metric of each host in DRS is *normalized entitlement*, which is defined as the sum of the per-VM entitlements for each VM running on the host divided by the capacity of the host. DRS chooses to migrate the VM that reduces imbalance most and the move-selection step repeats until either the load imbalance is below a user-set threshold or the number of moves generated in the current pass hits a configurable limit based on an estimate of the number of moves that can be executed in five minutes. Finally DRS optionally runs dynamic power management, which opportunistically saves power by dynamically right-sizing cluster capacity to match recent workload demand, while respecting the cluster constraints and resource controls.

4.2 Powercap Allocation

Powercap Allocation redistributes power caps if needed to allow DRS to correct constraint violations. DRS's ability to correct constraint violations is impacted by host power caps, which can limit the available capacity on target hosts. However, as shown in Fig 1a, by increasing the host power cap, the DRS algorithm can be more effective in correcting constraint violations by redistributing the cluster's unreserved power budget.

CloudPowerCap and DRS work in coordination, as shown in Figure 3, to enhance the system's capability to correct constraints violations.

1) Powercap Allocation first calls *GetFlexiblePower* to get *flexiblePower*, which is a special clone of the current cluster snapshot in which host power cap of each host is set to its reserved power cap, i.e., the minimum power cap needed to support the capacity corresponding to the reservations of the VMs currently running on that host.

2) The *flexiblePower* is used as a parameter to call *ConstraintsCorrection* function in DRS, which recommends VM migrations to enforce constraints and update hosts' reserved power caps for the new VM placements after the recommended migrations.

3) As a result of performing *ConstraintsCorrection*, DRS generates VM migration actions to correct con-

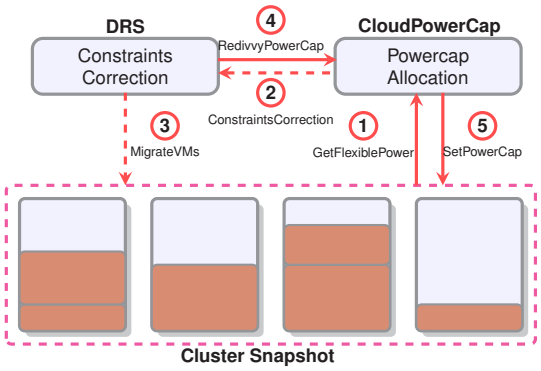


Figure 3: Coordination between CloudPowerCap and DRS to correct constraints. Solid arrows indicate invocations of CloudPowerCap functions while dashed arrows indicate invocations of DRS functions.

straints. Note that when applying VMs migration actions on hosts in the cluster, dependencies are respected between these actions and any prerequisite power cap setting actions generated by CloudPowerCap.

4) If some constraints are corrected by DRS, the power caps of source and target hosts may need to be reallocated to ensure fairness. For this case, *RedivvyPowerCap* of CloudPowerCap is called to redistribute the power cap.

5) Finally Powercap Allocation generates actions to set the power cap of hosts in the cluster according to the results of *RedivvyPowerCap*.

The key function in Powercap Allocation is *RedivvyPowerCap*, in which the unreserved power budget is redistributed after the operations for constraint violation correction. The objective of RedivvyPowerCap is to distribute the cluster power budget according to *proportional resource sharing* [27] for maintaining fairness of unreserved power budget distribution across hosts after the constraint correction. RedivvyPowerCap may introduce new opportunity to move VMs to correct constraints after redistributing the unreserved power budget. However, for simplicity, Powercap Allocation stops after invoking RedivvyPowerCap only once.

4.3 Entitlement Balancing

Entitlement balancing is critical for systems managing distributed resources, to deliver resource entitlements and improve the responsiveness to bursts in resource demand. For resource management systems like DRS without the concept of dynamic host capacity, entitlement balancing achieves both of these goals by reducing imbalance via migrating VMs between hosts. However, with dynamic power cap management, CloudPowerCap can alleviate imbalance by increasing the power caps of heavy loaded hosts while reducing the power caps of lightly loaded hosts rather than actually migrating VMs between those hosts as shown in Figure 1b. Considering the almost negligible overhead of power cap reconfiguration comparing to VM migration, Powercap-based Entitlement Balancing is preferred to DRS entitlement balancing when the cluster is imbalanced. However, because power cap adjustment has a limited range of operation, Powercap-based Entitlement Balancing may not fully eliminate imbalance in the cluster.

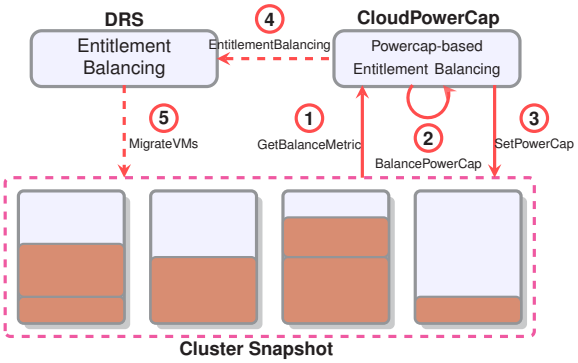


Figure 4: Work flow of Powercap-based Entitlement Balancing and its interaction with DRS entitlement balancing. Solid arrows indicate invocations of CloudPowerCap functions while dashed arrows indicate invocations of DRS functions.

The process of powercap based entitlement balancing and its interaction with DRS load balancing are shown in Figure 4.

1) To acquire the status of entitlement imbalance of the cluster, Powercap-based Entitlement Balancing first calculates the DRS imbalance metric for the cluster.

2) Then Powercap-based Entitlement Balancing tries to reduce the entitlement imbalance among hosts by adjusting their power caps in accordance with their normalized entitlements.

3) If Powercap-based Entitlement Balancing is able to impact cluster imbalance, its host power cap redistribution actions are added to the recommendation list, with the host power cap reduction actions being prerequisites of the increase actions.

4) If Powercap-based Entitlement Balancing has not fully balanced the entitlement among the hosts, DRS entitlement balancing is invoked on the results of Powercap-based Entitlement Balancing to reduce entitlement imbalance further.

5) DRS may generate actions to migrate VMs.

The key function *BalancePowerCap* was developed along the lines of *progressive filling* to achieve max-min fairness [2]. The function progressively increases the host power cap of the host(s) with highest normalized entitlement while progressively reducing the host power cap of the host(s) with lowest normalized entitlement . This

process is repeated until either the DRS imbalance metric crosses the balance threshold or any of the host(s) with highest normalized entitlement reach their peak capacity and hence further reduction in overall imbalance is limited by those hosts.

4.4 Powercap Redistribution

Powercap Redistribution responds to DPM dynamically powering on/off hosts. When CPU or memory utilization becomes high, DPM powers on hosts and Powercap Redistribution ensures that sufficient power cap is assigned to the powering-on host. On the other hand, when both CPU and memory utilization are low for a sustained period, DPM may consolidates VMs onto fewer hosts and powers off the remaining hosts to save energy. In this case, Powercap Redistribution distributes the power caps of the powered-off hosts among the active hosts to increase their capacity.

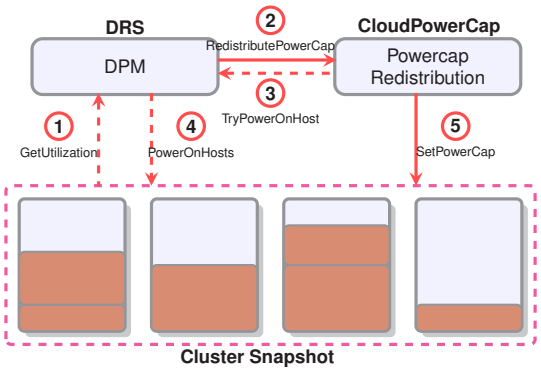


Figure 5: Coordination between CloudPowerCap and DRS and DPM in response to power on/off hosts. Solid arrows indicate to invoke CloudPowerCap functions while dashed arrows indicate to invoke DRS functions.

The coordination between Powercap Redistribution and DPM when DPM attempts to power on a host is depicted in Figure 5.

1) If there is sufficient unreserved cluster power budget to set the target host's power cap to peak, the host obtains its peak host power cap from the unreserved cluster power budget and no power cap redistribution is needed.
2) If the current unreserved cluster power budget is not sufficient, *RedistributePowerCap* is invoked to allow the powering-on candidate host to acquire more power from those hosts with lower CPU utilization.
3) DPM decides whether to power on the candidate host given its updated power cap after redistribution and its ability to reduce host high utilization in the cluster.
4) If the host is chosen for power-on, the normal DPM function is invoked to generate the action plan for powering on the host.
5) If DPM decides to recommend the candidate power-on, any needed host power cap changes are recommended as prerequisites to the host power-on.

4.5 Implementation Details

We implemented CloudPowerCap on top of VMware's production version of DRS. Like DRS, CloudPowerCap is written in C++. The entire implementation of CloudPowerCap comprises less than 500 lines of C++ code, which demonstrates the advantage of instantiating power budget management as a separate module that coordinates with an existing resource manager through well-defined interfaces.

As described previously in this section, DRS operates on a snapshot of the VM and host inventory it is managing. The main change we made for DRS to interface with CloudPowerCap was to enhance the DRS method for determining a host's CPU capacity to reflect the host's current power cap setting in the snapshot. Other small changes were made to support the CloudPowerCap functionality, including specifying the power budget, introducing a new action that DRS could issue for changing a host's power cap, and providing support for testability.

During CloudPowerCap initialization, for each host, the mapping between its current power cap and its effective capacity is established by the mechanisms described in Section 2.1. For a powered-on host, the power cap value should be in the range between the host's idle and peak power. When computing power-capped capacity of a host based on the power model (3), it is important to ensure that the capacity reserved by the hypervisor on the host is fully respected. Hence, the power-capped capacity $C_{mcapped}$ managed by the resource management system, i.e., managed capacity, is computed as:

$$C_{mcapped} = C_{capped} - C_H, \tag{4}$$

where the power-capped raw capacity C_{capped} is computed using Equation (1) and C_H is the capacity reserved by the hypervisor.

The implementation of Powercap Allocation entailed updating corresponding DRS methods to understand that a host's effective capacity available for constraint correction could be increased using the unreserved power budget, and adding a powercap redivvy step optionally run at the end of the constraint correction step. Powercap Balancing, which leverages elements of the powercap redivvying code, involved creating a new method to be called before the DRS balancing method. Powercap Redistribution changed DPM functions to consider whether to turn on/off hosts based not only on utilization but also on the available power budget.

5 Evaluation

In this paper we evaluate CloudPowerCap in the DRS simulator under three interesting scenarios. The first experiment evaluates CloudPowerCap's capability to rebalance normalized entitlement among hosts while avoiding the overhead of VM migration. The second experiment shows how CloudPowerCap allows CPU and memory capacity trade-offs to be made at runtime. This experiment includes a relatively large host inventory to show the capacity trade-offs at scale. The third experiment shows how CloudPowerCap reallocates the power budget of a powered-off host to allow hosts to handle demand bursts.

In these experiments, we compare CloudPowerCap against two baseline approaches of power cap management: *StaticHigh* and *Static*. Both approaches assign equal power cap to each host in the cluster at the beginning and maintain those power caps throughout the experiment. *StaticHigh* sets power cap of the host to its peak power, maximizing throughput of CPU intensive applications. However for applications in which memory or storage become constrained resources, it can be beneficial to support more servers to provision more memory and storage. Hence in *Static*, the power cap of a host is intentionally set lower than the peak power of the host. Compared with *StaticHigh*, more servers may be placed with *Static* to enhance the throughput of applications with memory or storage as constrained resources. However both approaches lack the capability of flexible power cap allocation to respond to workload spikes and demand variation.

5.1 DRS Simulator

The DRS simulator [8] is used in developing and testing all DRS algorithm features. It is a high-fidelity simulator and provides a realistic execution environment, while allowing much more flexibility and precision in specifying VM demand workloads and obtaining repeatable behavior close to the real hardware.

The DRS simulator simulates a cluster of hosts and VMs. A host can be defined using parameters including number of physical cores, CPU capacity per core, total memory size, and power consumption at idle and peak. A VM can be defined in terms of number of configured virtual CPUs (vCPUs) and memory size. Each VM's workload can be described by an arbitrary function over time, with the simulator generating CPU and memory demand for that VM based on the specification.

Given the input characteristics of hosts and the VMs' resource demands and specifications, the simulator mimics CPU and memory schedulers, allocating resources to the VMs in a manner consistent with the behavior of hosts in a real DRS cluster. The simulator calculates VMs' migration cost in accordance with several realistic factors, for example, VMs' read/write memory access and the available I/O and network bandwidth. The simulator also models the hypervisor CPU and memory overheads.

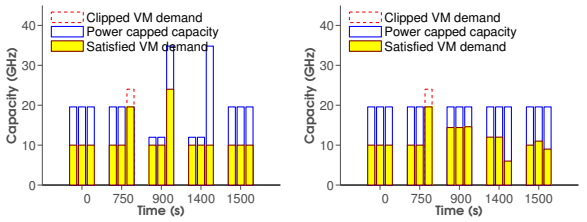
The simulator is able to estimate the power consumption of the hosts based on the power model given in Equation (1) in Section 2.2. For this work, the simulator was updated to respect the CPU capacity impact associated with a host's power cap.

5.2 Headroom Rebalancing

CloudPowerCap can reassign power caps to balance headroom for bursts, providing a quick response to workload imbalance due to VM demand changes. Such reassignment of power caps can improve robustness of the cluster and reduce or avoid the overhead of VM migration for load balancing. To evaluate impact of CloudPowerCap on headroom balancing, we perform an experiment in which 30 VMs, each with 1vCPU and 8GB memory, run on 3 hosts with the configuration shown in Table 1. Figures 6a and 6b plot the simulation results under CloudPowerCap and Static with a static power cap allocation of 250W per host, respectively. Initially, at time 0 seconds, the VMs are each executing similar workloads of 1 GHz CPU and 2 GB memory demand, and are evenly distributed across the hosts. At time 750 seconds, the VMs on one host spike to 2.4 GHz demand, thereby increasing the demand on that host above its power-capped capacity. When DRS is next invoked at time 900 seconds (running every 300 seconds by default), its goal is to rebalance the hosts' normalized entitlements. Under the static power cap, DRS migrates the VMs to balance the normalized entitlements. In contrast, CloudPowerCap reassigns the hosts' power caps to reduce the caps on the light-loaded hosts (to 215W) and increase them on the heavy-loaded host (to 320W). This addresses the host overutilization and imbalance without latency and overhead associated with VM migration, which is particularly important in this case, since the overhead further impacts the workloads running on the overutilized host. At time 1400 seconds, the 2.4 GHz VM demand spike ceases, and those VMs resume running at their original 1 GHz demand until the experiment ends at time 2100 seconds. Again, CloudPowerCap avoids the need for migrations by reassigning the host power caps to their original values. In contrast, Static performs two DRS entitlement balancing phases and migrates several VMs at time 900 seconds and 1500 seconds.

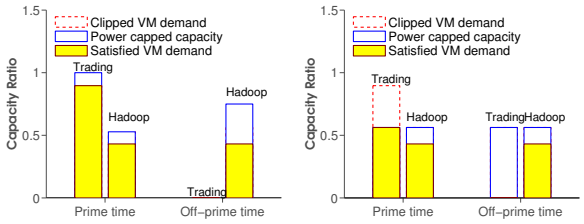
	CPU Payload Ratio	Migration
CPC	0.99	0
Static	0.89	7
StaticHigh	1.00	0

Table 3: CloudPowerCap rebalancing without migration overhead



(a) CloudPowerCap (b) Static

Figure 6: Headroom balancing on a group of 3 hosts. Hosts are grouped at each event time.



(a) CloudPowerCap (b) Static

Figure 7: Trade-offs between dynamic resource capacities. *Trading* indicates a group of servers running production trading VMs while *Hadoop* represents servers run production Hapdoop compute VMs.

Table 3 compares the CPU payload ratio, which is the ratio between actual CPU capacity and allocated CPU capacity to the VMs, under CloudPowerCap, Static using 250W static host power caps, as well as StaticHigh using the power caps equivalent to the peak capacity of the host. For Static, the CPU overhead caused by VMs migration has a significant overall impact on the CPU payload delivered to the VMs because the cycles needed for VMs migration directly impact VMs' performance. For CloudPowerCap, there is a relatively small impact to performance after the burst and before DRS can run CloudPowerCap to reallocate the host power caps. The power cap setting can be executed by the host within 1 millisecond and introduces minor payload overhead.

5.3 Flexible Resource Capacity

CloudPowerCap supports flexible use of power to allow trade-offs between resource capacities to be made dynamically. To illustrate such a trade-off at scale, we consider a cluster of hosts as described in Section 2.1. We model the situation in which the cluster is used to run both production trading VMs and production hadoop compute VMs. The trading VMs are configured with 2 vCPUs and 8 GB and they are idle half the day (off-prime time), and they run heavy workloads of 2x2.6 GHz and 7 GB demand the other half of the day (prime time). They access high-performance shared storage and hence are constrained to run on hosts with access to that storage, which is only mounted on 8 hosts in the cluster. The hadoop compute VMs are configured with 2 vCPUs and 16 GB and each runs a steady workload of 2x1.25 GHz and 14 GB demand. They access local storage and hence are constrained to run on their current hosts and cannot be migrated. During prime time, the 8 servers running the trading VMs do not receive tasks for the hadoop VMs running on those servers; this is accomplished via an elastic scheduling response to the reduced available resources [29]. Figure 7 shows the simulation results of the cluster under CloudPowerCap and the Static configuration of power caps.

Table 4 compares the CPU and memory payload delivered for three scenarios, and shows the impact on the trading VMs. The staticHigh scenario involves deploying 25 servers with power caps of 320 W, which immediately and fully supports the trading VMs prime time demand but limits the overall available memory and local disks in the cluster associated with the 25 servers. The Static scenario instead involves deploying 32 servers with each host power cap statically set to 250 Watts. This scenario allows more memory and local disks to be accessed, increasing the overall CPU and memory payload delivered because more hadoop work can be accomplished, but limits the peak CPU capacity of each host, meaning that the trading VMs run at only 62 percent of their prime time demand. With CloudPowerCap, the benefits to the hadoop workload of the static scenario are retained, but the power caps of the hosts running the trading VMs can be dynamically increased, allowing those VMs' full prime time demand to be satisfied.

	CPU Ratio	Mem Ratio	Trading Ratio
CPC	1.24	1.28	1.00
Static	1.21	1.28	0.62
StaticHigh	1.00	1.00	1.00

Table 4: CloudPowerCap enabling flexible resource capacity. Trading ratio indicates the ratio that production trading VMs demands in prime time are satisfied.

5.4 Standby Host Power Reallocation

CloudPowerCap can reallocate standby hosts' power cap to increase the capacity of powered-on hosts and thereby their efficiency and ability to handle bursts. To demonstrate this, we consider the same initial setup in terms of hosts and VMs as in the previous experiment. In this case, all VMs are running a similar workload of 1.2 GHz and 2 GB memory demand. At time 750 seconds, each VM's demand reduces to 400 MHz, and when DRS is next invoked at time 900 seconds, DPM recommends that the VMs be consolidated onto two hosts and that another host is powered-off. After the host has been evacuated and

powered-off at time 1200 seconds, CloudPowerCap reassigns its power cap to 0 and reallocates the rack power budget to the two remaining hosts, setting their power caps to 320W each. At time 1400 seconds, there is an unexpected spike. In the case of statically-assigned power caps, the host that was powered-off is powered back on to handle the spike, but in the CloudPowerCap case, the additional CPU capacity available on the 2 remaining hosts given their 320 W power caps is sufficient to handle this spike and the powered-off host is not needed.

	CPU Payload Ratio	Migration	Power Ratio
CPC	1.00	10	1.00
Static	0.98	19	1.36
StaticHigh	1.00	10	1.00

Table 5: CloudPowerCap reallocating standby host power

Table 5 compares the CPU payload in cycles delivered to the VMs for CloudPowerCap, Static, and StaticHigh. In this case, a number of additional vMotions are needed for Static, but the overhead of these vMotions does not significantly impact the CPU payload, because there is plenty of headroom to accomodate this overhead. However, Static consumes much more power than the other 2 cases, since powering the additional host back on and repopulating it consumes significant power. In contrast, CloudPowerCap is able to match the power efficiency of the baseline, by being able to use peak capacity of the powered-on hosts.

6 Related Work

Several research projects have considered power cap management for virtualized infrastructure [21, 18, 16, 28, 19, 14]. Among them, the research most related to our work is [19], in which authors proposed VPM tokens, an abstraction of changeable weights, to support power budgeting in virtualized environment. Like our work, VPM tokens enables shifting *power budget slack* which corresponds to *headroom* in this paper, between hosts. However the power cap management system based on VPM tokens are independent of resource management systems and may generate conflicting actions without coordination mechanisms.

In contrast, interoperating with a cloud resource management system like DRS also allows CloudPowerCap to support interesting additional features: 1) CloudPowerCap accommodates consolidation of physical servers caused by dynamic power management while previous work assumed a fixed working server set, 2) CloudPowerCap is able to handle and facilitate VM migration caused by correcting constraints imposed on physical servers and VMs, 3) CloudPowerCap can also deal with and enhance power cap management in the presence of load balancing. In most previous work, only part of these features are provided.

The authors of [21] describe managing performance and power management goals at server, enclosure, and data center level and propose handling the power cap hierarchically across multiple levels. Optimization and feedback control algorithms are employed to coordinate the power management and performance indices for entire clusters. In [28], the authors build a framework to coordinate power and performance via Model Predictive Control through DVFS (Dynamic Voltage and Frequency Scaling). To provide power cap management through the VMs management layer, [18] proposed throttling VM CPU usage to respect the power cap. In their approach, feedback control is also used to enforce the power cap while maintaining system performance. Similarly, the authors in [16] also discussed data center level power cap management by throttling VM resource allocation. Like [21], they also adopted a hierarchical approach to coordinate power cap and performance goals. In [14], authors proposed a relatively accurate model to estimate power consumption of the VM based on not only CPU utilization but also memory and disk usage. However this work has no discussion of dynamical power budget provisioning across a virtualized cluster.

While all of these techniques attempt to manage both power and performance goals, their resource models for the performance goals are incomplete in various ways. For example, none of the techniques support guaranteed SLAs (reservations) and fair share scheduling (shares). Some build a feedback model needing application-level performance metrics acquired from cooperative clients, which is rare especially in public clouds [1].

Although power management in virtualized cluster is extensively studied previously [20, 22, 15], which focus on reducing power consumption while maintaining preformance and is different to the goal of CloudPowerCap.

7 Conclusion

Many modern data centers have underutilized racks. Server vendors have recently introduced support for per-host power caps, which provide a server-enforced limit on the amount of power that the server can draw, improving rack utilization. However, this approach is tedious and inflexible because it needs involvement of human operators and does not adapt in accordance with workload variation. This paper presents CloudPowerCap to manage a cluster power budget for a virtualized infrastructure. In coordination with resource management, CloudPowerCap provides holistic and adaptive power budget management framework to support service level agreements, fairness in spare power allocation, entitlement balancing and constraint enforcement.

References

[1] BEN-YEHUDA, O. A., BEN-YEHUDA, M., SCHUSTER, A., AND TSAFRIR, D. The resource-as-a-service (RaaS) cloud. In *HotCloud'12* (2012).

[2] BERTSEKAS, D., GALLAGER, R., AND HUMBLET, P. *Data networks*. Prentice-Hall, 1992.

[3] DAVIS, J., RIVOIRE, S., AND GOLDSZMIDT, M. Star-Cap: Cluster Power Management Using Software-Only Models. Tech. rep., MSR-TR-2012-107, Microsoft Research, 2012.

[4] DELL INC. Dell Energy Smart Management.

[5] FAN, X., WEBER, W.-D., AND BARROSO, L. A. Power provisioning for a warehouse-sized computer. *SIGARCH Comput. Archit. News 35*, 2 (June 2007), 13–23.

[6] FU, Y., HOLLER, A., AND LU, C. CloudPowerCap: Integrating Power Budget and Resource Management across a Virtualized Server Cluster. `http://arxiv.org`.

[7] GARTNER RESEARCH. Shrinking Data Centers: Your Next Data Center Will Be Smaller Than You Think.

[8] GULATI, A., HOLLER, A., JI, M., SHANMUGANATHAN, G., WALDSPURGER, C., AND ZHU, X. VMware Distributed Resource Management: Design, Implementation, and Lessons Learned. *VMware Technical Journal* (Mar 2012).

[9] GULATI, A., SHANMUGANATHAN, G., HOLLER, A., AND AHMAD, I. Cloud-scale resource management: challenges and techniques. In *HotCloud* (2011).

[10] HAMILTON, J. Overall data center costs. `http://perspectives.mvdirona.com/2010/09/18/OverallDataCenterCosts.aspx`.

[11] HEWLETT-PACKARD, INTEL, MICROSOFT, PHOENIX, AND TOSHIBA. Advanced Configuration and Power Interface Specification. Tech. rep., 2011.

[12] HP INC. HP Power Capping and HP Dynamic Power Capping for Proliant Servers.

[13] IBM INC. IBM Active Energy Manager.

[14] KANSAL, A., ZHAO, F., LIU, J., KOTHARI, N., AND BHATTACHARYA, A. Virtual machine power metering and provisioning. In *Proceedings of the 1st ACM symposium on Cloud computing* (2010), ACM, pp. 39–50.

[15] KUSIC, D., KEPHART, J., HANSON, J., KANDASAMY, N., AND JIANG, G. Power and performance management of virtualized computing environments via lookahead control. *Cluster computing 12*, 1 (2009), 1–15.

[16] LIM, H., KANSAL, A., AND LIU, J. Power Budgeting for Virtualized Data Centers. In *USENIX ATC* (2011).

[17] MINAS, L., AND ELISON, B. The problem of power consumption in servers, 2009. `http://software.intel.com`.

[18] NATHUJI, R., ENGLAND, P., SHARMA, P., AND SINGH, A. Feedback driven QoS-aware power budgeting for virtualized servers. In *FeBID* (2009).

[19] NATHUJI, R., SCHWAN, K., SOMANI, A., AND JOSHI, Y. VPM tokens: virtual machine-aware power budgeting in datacenters. *Cluster computing* (2009).

[20] PINHEIRO, E., BIANCHINI, R., AND HEATH, T. *Dynamic Cluster Reconguration for Power and Performance*. Kluwer Academic, 2003.

[21] RAGHAVENDRA, R., RANGANATHAN, P., TALWAR, V., WANG, Z., AND ZHU, X. No power struggles: Coordinated multi-level power management for the data center. In *ACM SIGARCH Computer Architecture News* (2008), vol. 36, ACM, pp. 48–59.

[22] RANGANATHAN, P., LEECH, P., IRWIN, D. E., AND CHASE, J. S. Ensemble-level Power Management for Dense Blade Servers. In *ISCA* (2006), pp. 66–77.

[23] STRUNK, A. Costs of Virtual Machine Live Migration: A Survey. In *IEEE Eighth World Congress on Services* (2012).

[24] VMWARE, INC. Resource Management with VMware DRS, 2006.

[25] VMWARE, INC. VMware Distributed Power Management Concepts and Use, 2010.

[26] VMWARE, INC. VMware vSphere vMotion: Architecture, Performance, and Best Practices in VMware vSphere 5, 2011.

[27] WALDSPURGER, C. A., AND WEIHL, W. E. Lottery scheduling: flexible proportional-share resource management. In *OSDI* (1994), USENIX Association.

[28] WANG, X., AND WANG, Y. Coordinating Power Control and Performance Management for Virtualized Server Clusters. *IEEE Transactions on Parallel and Distributed Systems 22*, 2 (Feb. 2011), 245–259.

[29] WHITE, T. *Hadoop: The Definitive Guide*. O'Reilly Media, 2009.

A Comprehensive Resource Management Solution for Web-based Systems

F. Seracini, M. Menarini, and I. Krüger
Dep. of Computer Science and Engineering
UC San Diego - La Jolla (CA), 92093 USA
fseracini@ucsd.edu

L. Baresi, S. Guinea, and G. Quattrocchi
Dip. di Elettronica, Informazione e Bioingegneria
Politecnico di Milano - Milano, 20148 Italy
luciano.baresi@polimi.it

Abstract

This paper presents an autonomic resource management solution that looks at the non-functional qualities of a Web-based system, as well as at the characteristics of the infrastructural resources it uses. It exploits a detailed performance model of both these aspects to increase the efficiency of resource allocation. The solution is evaluated on an auction and shopping benchmark web site, and compared to a baseline approach and to an existing solution from literature. Results show that, by jointly taking into account the different software and hardware facets of our application, we can reduce the amount of resources allocated by up to 42.5% compared with an existing work from the literature.

1 Introduction

Modern software systems are subject to continuous change, such as changes in the stakeholder requirements, evolutions in the software or resource components, and variations in the execution context [4]. System designers need refined runtime management techniques and tools to cope with these changes, so that the systems can continue to provide the required functional and non-functional qualities.

In this paper we focus on complex Web-based systems. In these cases, change often manifests itself as significant fluctuations in the system's workload. Although they may often follow historical and seasonal patterns, multiple spikes may occur without warning (e.g., flash crowds [2]). These situations can lead to frustrated customers, and loss of online business. As a consequence, service providers tend to over-allocate resources. The result is that the average server utilization, in a typical data center, is around 30-40% [5]; some studies even estimate the utilization level as low as 18% [20]. A better utilization of resources would lead to more efficient systems. Fortunately, the diffusion of dynamic resource allocation techniques provide the flexibility needed to build systems that can adapt their configurations based on the actual workloads and acquire resources on demand.

These systems must embed fine-grained *autonomic* capabilities that cover all their facets, from their hardware resources to their software. By introducing a MAPE (Monitor, Analyze, Plan, and Execute) control loop [11] that can *estimate* the application's load and performance, and help understand how many resources should be provisioned, we can compute and apply anticipatory or corrective actions on the fly.

Many different performance models have been proposed in the past. Unfortunately, in their load estimation they tend to only take into account the volume of requests (e.g. [22, 23]). Regrettably, it has been shown that this is not enough to estimate an application's resource usage effectively [19, 21]. Some systems take a further step and claim to be workload-mix aware, meaning that they distinguish between different types of requests, based on their aggregated response times. Yet, these approaches still do not consider the hardware resource usage that these requests imply, and the resource contention that can arise. Resource contention is one of the main causes of performance degradation in systems with high loads [9, 17, 12, 16]. Knowing the resource usage profile of each of the different types of requests is vital in defining an effective allocation strategy.

Our approach provides a comprehensive, innovative solution for the *autonomic* management of complex Web-based applications. We exploit and suitably extend ECoWare [3], a monitoring and adaptation framework previously developed at Politecnico di Milano for service-based systems. Thanks to these extensions we are now able to perform fine-grained measurements of a Web-based system's behavior, by correlating the runtime data that we retrieve from its hardware and its software. To achieve this we included in ECoWare a completely new performance model that takes into account both the application's workload mix and the hardware infrastruc-

ture. It defines the resource footprint of the different application requests in terms of CPU instructions, cache, and main memory accesses. These metrics are then used to profile the resource usage of the requests, and to increase the accuracy of the performance predictions.

1.1 Running Example

To evaluate our approach, we used RUBiS [1], a well-known benchmark that simulates a common auction and shopping web site, and compared our results against a baseline approach and an existing solution from literature [19]. RUBiS was developed using HTML, Java Servlets, and SQL technology. It uses an Apache server for its static content (e.g., the home page, images, etc.), multiple JBoss Application Servers for its dynamic content (e.g., the user's bidding history, the products available in a given geographical region, etc.), and a MySQL Server for its backend data tier. A content-aware load balancer routes the requests accordingly.

In this paper, we focus on how we can use ECoWare to satisfy an SLA of the kind "the X^{th} percentile response time should be within a fixed threshold over a period of time". The percentile response time was calculated taking into account the response times seen by ECoWare over a two and a half minute window.

2 Approach Overview

The overarching goal of our ECoWare project is to empower system designers and maintainers in the development of self-adaptive systems. ECoWare is a general-purpose monitoring and adaptation framework that exploits the common *MAPE (Monitoring – Analysis – Planning – Execution) control loop* approach. System designers can tailor ECoWare to their needs by choosing appropriate technology- and application-specific sensors and actuators, and by choosing appropriate analysis and planning techniques.

Figure 1 shows how ECoWare can be applied to a complex Web-based system. The application uses multiple servers for each tier, and new servers can be added from a *Pool of Free Servers*. The application is on the left-hand side of the figure, and it interacts with ECoWare, which is on the right-hand side of the figure, through an Event Bus. Light gray components represent previous work, while white components are novel contributions of this paper.

For the **Monitoring** step, ECoWare provides two kinds of components: *Key Performance Indicator (KPI) Processors* and *Aggregators*. The former use the low-level events produced by the probes to calculate various non-functional KPIs, such as average response times, arrival rates, and throughputs. The latter correlate multiple

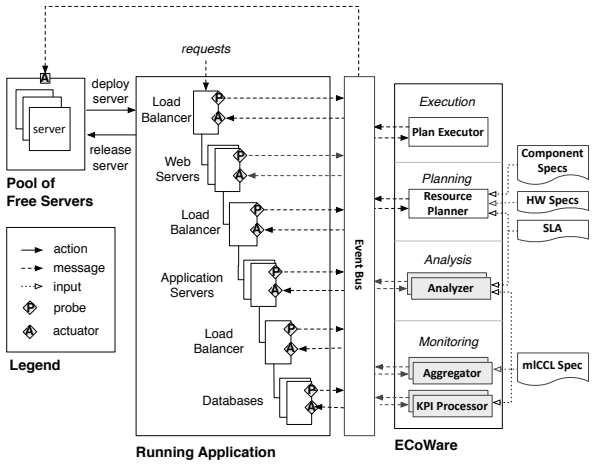


Figure 1: Example of an application with ECoWare.

low-level events and/or KPIs to produce a holistic understanding of the application's behavior. For the **Analysis** step, ECoWare's *Analyzers* listen to the bus for monitoring events (e.g., average response times) and evaluate them against certain constraints. For the **Planning** step, ECoWare provides a *Resource Planner*. This component uses a workload-mix and hardware-aware performance model to understand what changes should be made to the resource provisioning. Finally, for the **Execution** step, ECoWare provides the *Plan Executor*. This component coordinates multiple actuators to accomplish the desired adaptations.

ECoWare is extremely flexible; it can support multiple kinds of management strategies, e.g., *reactive*, *proactive*, and *periodic*. We can even mix different strategies together, to differentiate how we up-scale (e.g., reactively) from how we down-scale (e.g., periodically) our resources. This flexibility allows us to be fast to provision new resources, yet cautious when un-provisioning them; and it allows us to reduce instability.

3 The Resource Planner

The Resource Planner is responsible for identifying the correct amount of resources to add or to remove from the running system. In order to do this, it implements the *Compute Configuration Algorithm*, shown in Algorithm 1. Every decision that the algorithm makes is checked against a hardware- and workload-mix aware *Performance Model*. This allows us to avoid a trial-and-error approach in which we deploy (or remove) hardware, and then wait for the subsequent iteration of the control loop to tell us if it solved the problem.

Algorithm 1: Compute Configuration Algorithm

Input: $\overline{\lambda} = (\lambda_1, \lambda_2, ..., \lambda_n)$, R_{SLA}, and $SysConf_{Current}$
Output: $SysConf_{New}$

```
if freeServerPool = ∅ then
    return SysConf_Current
else
    select server S ∈ freeServerPool
    SysConf_New = SysConf_Current ∪ S
    R_New = AnalyzePerf(λ, SysConf_New)
    if R_New satisfies R_SLA then
        return SysConf_New
    else
        return ComputeConfiguration(λ, R_SLA, SysConf_New)
    end
end
```

3.1 Compute Configuration Algorithm

Every time the Resource Planner is notified with a SLA violation, it invokes the algorithm illustrated in Algorithm 1. The algorithm takes the vector of arrival rates per request type ($\overline{\lambda}$), the response time value defined in the SLA (R_{SLA}), and the current system configuration ($SysConf_{Current}$), and determines a new system configuration ($SysConf_{New}$) that satisfies the SLA. $\overline{\lambda}$ is provided by ECoWare's monitoring.

The algorithm takes a server S from the pool of free servers ($freeServerPool$) and adds it to the current system configuration to create $SysConf_{New}$. It then verifies the new system configuration by running *AnalyzePerf*, which solves a queuing network (QN) model, i.e., the Performance Model, for $SysConf_{New}$ and the actual workload $\overline{\lambda}$. *AnalyzePerf* returns a vector of response times $\overline{R_{New}}$. If all the values of $\overline{R_{New}}$ are smaller than R_{SLA}, the algorithm terminates returning $SysConf_{New}$; otherwise, the algorithm recursively calls itself with parameters $\overline{\lambda}$, $SysConf_{New}$, and R_{SLA}. In the worst case scenario, i.e., in which it continuously fails to satisfy the SLA, the algorithm continues until all the available resources (i.e. servers) are provisioned.

The autonomic system will also periodically check its provisioning to see whether resources can be safely removed. In this case, the Resource Planner invokes Algorithm 1 passing a base configuration of one server as $SysConf_{Current}$. The algorithm recursively adds servers till the queuing network (QN) solver validates the configuration. By doing so, ECoWare guarantees that the minimal set of servers is always allocated.

3.2 Performance Model

Algorithm 1 leverages a QN model to estimate the application's response times per request type, given a specific workload and a specific system configuration, allowing ECoWare to compare the estimated values with those required by the SLA.

In a QN model (e.g., [7]), each component is represented as a queue, called a service station. A model can be closed, open, or mixed, depending on whether the volume of requests is constant, fluctuating, or mixed. Whenever the number of incoming requests is higher than the computing capacity of a service station, the requests are queued; the time spent in queue is called waiting time. Each type of service request is defined by an arrival rate process (i.e., how the arriving requests are distributed in a unit of time) and a service demand distribution (overall time that a single request spends at each of the service stations during a complete execution). This can be expressed using Kendall's notation in the form $A/S/C - POLICY$ [7]. A describes the arrival process, S describes the distribution of service time of a job, C describes the number of servers at the node, and $POLICY$ describes the queuing discipline used at that node (e.g., first come first served, processor sharing, etc.). In Kendall's notation, M stands for Markov or memoryless, meaning that arrivals occur according to a Poisson process; and G stands for general, meaning an arbitrary probability distribution.

As the arriving traffic of the different types of requests fluctuates over time, so does the mix of requests in execution. As a consequence, resource contention changes over time. Figure 2 shows the performance model of our running RUBiS example. To simplify the presentation, the figure only shows one server (with two cores) for the application tier. The model correctly captures the following three aspects: the fact that the approach is workload mix-aware, the fact that the system is composed of multiple tiers, and the fact that the application tier relies on specific hardware resources.

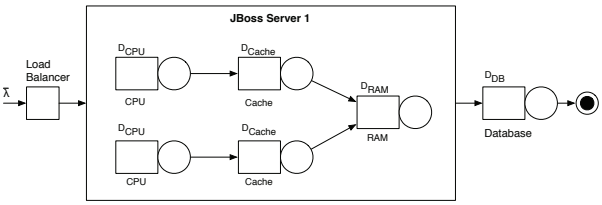


Figure 2: Performance Model in our experiments.

The intensity of the arriving flow of requests in input to the queuing network, that is the workload-mix, is described by $\overline{\lambda}$, the vector of arrival rates per request type. Regarding the fact that we have multiple tiers, the model concentrates on the application tier (JBoss) and on

the database tier (MySQL), and leaves out the web tier (Apache). The reason for this is that, in RUBiS, system performance is dominated by the application tier. The web tier only serves static content and has a negligible influence on the performance of the overall system. The database also shows a very moderate load, but we could not remove it entirely from the model since the JBoss servers query it. However, given its low utilization and resource contention, the role of the database was approximated by a single queue in the performance model.

In the application tier, i.e., in the dominating tier, we used a very fine-grained approach. We used multiple M/G/1-FCFS stations for the main hardware components of the application servers i.e., CPU, cache and main memory. (These were configured using data taken from our hardware probes through ECoWare monitoring.) This means that not only do we consider the traffic of incoming requests and their mix, we also consider the resource usage footprint of each request. On the other hand, we did not need this level of detail neither for the database tier, nor for the incoming load balancer. Therefore, we used a single M/G/1-FCFS station in both cases, and configured them using information collected by the Hibernate and Load Balancer probes, respectively. The results of our experiments, which we will discuss in the following section, show that these simplifications are correct.

By modeling the service demands for the resources, our model can estimate the waiting time of a request at each service station. Whenever a resource becomes congested, the waiting time at the corresponding service station grows. Since each type of request has a different response time and a different resource usage profile, different workload mixes (i.e., requests mixes) will use the hardware resources in a different way. As a result, the overall execution time is both mix- and volume-dependent.

Let us use a simple example to further exemplify our solution. Let us assume we have two types of requests, A and B, and that request type A is CPU-intensive and request type B is memory-intensive. If we mostly receive requests of type B, their total execution time, and the overall system's throughput, will suffer. The reason for this is that the cache will start being used by a high number of threads, and start becoming congested. When this happens, RAM will be accessed more frequently, resulting in longer waiting and execution times. A more balanced load between requests A and B may cause less contention, and the overall system's throughput would benefit.

Table 1 shows the increase in terms of response time, and number of cache and main memory accesses, for a memory intensive workload. Two different hardware architectures are compared: although they have the same CPU frequency and number of cores, HW_1 has twice the cache size per core than HW_2. The table clearly shows that the architecture with the larger cache size experiences much smaller performance degradation. Moreover, at high load levels (about 80% CPU utilization), the system with the smaller cache shows a 49% increase in time spent to access the main memory because of the increased cache conflicts. This demonstrates that reducing resource contention between running tasks is an effective way to improve a system's throughput, without sacrificing response times (this was also shown in [9, 8, 17]), and that explicitly modeling both the hardware and the resource usage profile of the requests can be an effective way to take resource contention into account, and to improve the accuracy of our performance predictions.

Table 1: Increase of response time (RT), cache accesses (CA), and memory accesses (MA) from low to high load levels for different hardware architectures.

HW Conf.	RT	CA	MA
HW_1	6%	$\sim 0\%$	19%
HW_2	26%	3%	49%

4 Experimental Evaluation

For the evaluation of this work, we focused on assessing ECoWare's frugality in allocating resources while satisfying the SLA. We adopted the queuing network model already illustrated in Figure 2. To create and solve the model we used the Java Modeling Tool JMT [6]. For our experiments we used seven servers equipped with Intel Xeon processors running at 2.66 GHz. Each processor was composed of two cores, each addressing a separate 6MB L2 cache; main memory was 32 GB on each machine.

During our initial evaluation of RUBiS, we loaded the system to measure the servlets' resource usage profiles. These were then used to calibrate the queuing network performance model. We focused on two servlets with fixed queries: `ViewUserInfo` and `ViewBidHistory`. Since the servlets' average execution times were very close, and often below 1ms, we decided to increase the load for servlet `ViewBidHistory`, to make it more representative of a real-case scenario. We also added a routine to `ViewUserInfo` that validates the content of the comments that are left about a given user (as typically done in many social web sites).

We evaluated the accuracy of ECoWare's provisioning strategy with multiple non-stationary workloads. Since provisioning decisions are applied independently at each tier, we focused on the application tier and over-provisioned the others to ensure that they did not represent bottlenecks.

To validate our approach, we compared it with a baseline solution that uses an M/M/K-FCFS queue as its performance predictor, and with the approach presented by Singh et al. in [19], where they use a closed formula approximation of a G/G/1 queue to predict server capacity for a given workload mix[1].

The first workload was designed to follow the structure of the workload used in [19]: it has a varying mix, yet a constant volume of requests. Our results showed that ECoWare used 13.75% fewer server-minutes (i.e., number of servers per allocation time) than the baseline and 42.5% less than [19] [2]. The second workload had both a varying workload mix and a varying volume of requests. The workload was executed several times and showed consistent results. For the sake of simplicity, we show here a randomly selected execution.

The Workload Figure 3 shows the workload mixes and the different time lengths of the experiment's steps. The maximum total number of requests per second at each step was limited to 22.5 in order to generate a workload that could be handled by the amount of servers at our disposal. The same randomly generated workload was used for all three solutions.

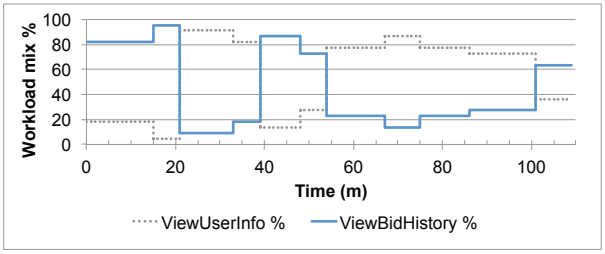


Figure 3: Request mix.

Server Provisioning and SLA. The baseline approach shows a strong instability throughout the entire workload. Figure 4 describes the provisioning decisions for the workload. The baseline approach allocates two or more servers every time the system experiences a transitory phase. These phases typically happen right after a steep variation in the workload and last for 1 or 2 minutes. After the transient phase is over, the baseline starts reducing the number of allocated servers, often incurring in SLA violations —see Figure 5. So the baseline approach has a tendency to under-allocate, and consequently experiences lots of SLA violations.

The solution from [19] exceeds the SLA limit twice at the beginning of the workload. The reason for that is the very high load that was selected by the random generator at the very beginning. Since we always begin with one allocated server, which obviously cannot satisfy the demand at the workload's first step, the two SLA violations are unavoidable and cannot be attributed to an erroneous provisioning. Indeed, the provisioning strategy completes the workload with no further violations, confirming its soundness. In terms of allocated servers, the solution from [19] appears to be more generous, requesting up to four servers for extended periods of time. Even though only three servers are actually allocated even when four are requested, the 95% percentile of the response time remains steadily well below the limit. An almost flat line for the response time throughout the workload shows that the servers are running at a very low utilization.

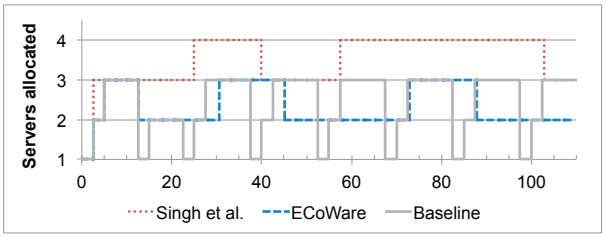


Figure 4: Number of servers allocated.

ECoWare experiences the same SLA violations at the beginning of the experiment as the previous two approaches. Similarly to [19], it does not incur in any further violation for the remaining of the workload. ECoWare correctly adapts its provisioning decisions throughout the experiment, allocating up to three servers in three different moments. ECoWare's diagram in Figure 5 shows wide fluctuations in the response time, proving that the servers have a much higher utilization, hence a longer execution and waiting time for the requests. However, ECoWare is able to maintain those fluctuations under the SLA limit, thus improving the overall utilization of the allocated servers.

Result. For this experiment, the baseline approach scored 302.5 server-minutes, but incurred in a significant amount of violations because of its tendency to under-allocate. The provisioning strategy from [19] used 385 server-minutes, but with a perfect response time, except for the two initial violations that could not be avoided. Finally, ECoWare only allocated 255 server-minutes, with no SLA violations except for the usual two at the beginning. ECoWare also showed a higher fluctuation in the response time, testifying a higher utilization of the servers. Overall, ECoWare used 16% fewer server-minutes than the baseline approach with less violations, and 33% less than [19], with the same amount of violations.

[1] In order to perform our comparisons we implemented Singh et al.'s approach according to the formulae presented in their paper.

[2] Due to lack of space, we cannot focus on this workload; a detailed presentation can be found online at `http://home.deib.polimi.it/guinea/ICAC2014/experiments.pdf`.

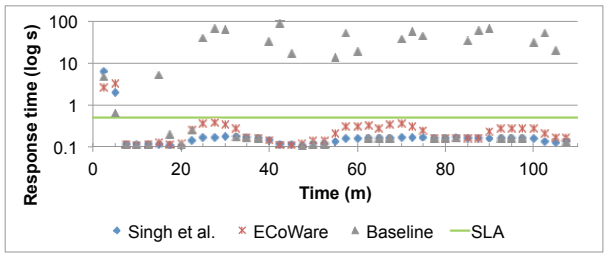


Figure 5: 95% Percentile response time.

4.1 Threats to Validity

A possible objection to our evaluation is that we only focus on two servlets. We consider this to not be an over-simplification (e.g. [19] uses three servlets). It is quite common, when there are many different kinds of requests, to use K-means clustering to group them into fewer clusters with similar service demands [19]. Clustering also allows us to mitigate a limit of queuing networks, i.e., that they do not scale well when the number of classes of jobs significantly increases.

Table 2: Average solving time for the QN model for different numbers of classes.

# of classes	2	3	5	7	10
Solving time (s)	3	5	9	15	18

To evaluate the scalability of our approach we benchmarked the QN solver with up to 10 classes (see Table 2), a realistic top limit for the number of clustered request types. The table shows that the solver still terminates within a reasonable time frame. We also bound the computational time to 20 seconds, at the cost of a possible loss of accuracy. This does not impact our results, since it takes the solver longer to terminate when the infrastructure is close to saturation. When this happens the response time will violate any given SLA, since it will grow unboundedly. Therefore, ECoWare's strategy saves time without losing any accuracy, by adding one server and re-running the solver.

5 Related Work

Most existing approaches for achieving dynamic resource provisioning are mix un-aware; they only consider the volume of arriving requests when determining their provisioning. In their QN model Almeida et al. [2] use the peak session arrival rate. Villela et al. [23] leverage QNs to model application servers and to solve an optimization problem for profit maximization; however, they only consider the aggregate number of requests generated by the clients. Kusic et al. [13] present an optimization framework for resource allocation, expressed as a sequential decision making problem under uncertainty, and solved using a limited look ahead control scheme. The approach distinguishes between different classes of clients with different SLA limits, but does not consider the resource usage profile. He et al. [10] focus on dynamic allocation for interactive systems (e.g. web search engines) where the quality of the results depends also on the allocated time; they do distinguish between interactive and batch jobs. In [25] Zhou et al. present a two-tier resource management framework that focuses on optimizing resource allocation while provisioning proportionality fairness to clients. They consider different classes of clients, associated with different SLA agreements. SPIRE [15] is an autonomous system for service provisioning driven by a utility function: optimize the average earned revenue over time, while satisfying the SLA. SPIRE also assumes a uniform service time for all the requests. As additional formulation of the dynamic allocation problem we also mention the decentralized control theory approach of [24] and the machine learning technique proposed in [14].

As we have seen in our study, different types of requests can have very different service times and resource usage profiles. Among mix-aware approaches, we mention Sharma et al. [18]. They use a network of M/G/1-PS queues and an approximate model to compute response time distribution. Their solution takes hardware configuration into consideration. However, their approach differs from ours as they focus on dealing with a multitude of hardware configurations in terms of service rates. Our approach instead focuses on modeling the hardware to achieve a higher accuracy in the performance predictions. We plan to extend our approach to consider different hardware configurations as well. Works in [17, 16] also focus on mitigating performance degradation caused by shared resource contention. Finally, Krebs et al. [12] provides metrics to quantify performance isolation in the context of shared resources.

6 Conclusions and Future Work

Allocation strategies that take into account the resource usage profiles of the different requests help increases the accuracy of our performance predictions, which in turn allows us to improve the utilization of our infrastructure. In the future we will continue to evaluate ECoWare in the context of cloud technology, and evaluate the advantages that could derive from the use of layered queuing networks for our models.

7 Acknowledgments

We thank Giuseppe Serazzi, Marco Gribaudo, and William Griswold for their excellent feedback.

References

[1] RUBiS – Rice University Bidding System – `http://rubis.ow2.org/`.

[2] ALMEIDA, V., AND MENASCE, D. Capacity Planning an Essential Tool for Managing Web Services. *IT Professional 4* (Aug. 2002), 33–38.

[3] BARESI, L., AND GUINEA, S. Event-based Multi-level Service Monitoring. In *Proceedings of the 20th IEEE International Conference on Web Services (ICWS)* (2013), pp. 83–90.

[4] BARESI, L., NITTO, E. D., AND GHEZZI, C. Toward Open-World Software: Issue and Challenges. *Computer 39*, 10 (2006), 36–43.

[5] BARROSO, L. A., AND HÖLZLE, U. The Case for Energy-Proportional Computing. *Computer 40*, 12 (Dec. 2007), 33–37.

[6] BERTOLI, M., CASALE, G., AND SERAZZI, G. JMT: Performance Engineering Tools for System Modeling. *SIGMETRICS Performance Evaluation Review 36*, 4 (2009), 10–15.

[7] BOLCH, G., GREINER, S., MEER, H. D., AND TRIVEDI, K. S. *Queueing Networks and Markov Chains: Modeling and Performance Evaluation with Computer Science Applications*, 2nd edition ed. Wiley, 2006.

[8] CASALE, G., AND SERAZZI, G. Bottlenecks Identification in Multiclass Queueing Networks using Convex Polytopes. In *Proceedingds of the 12th IEEE Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS)* (2004), pp. 223–230.

[9] DHIMAN, G., MARCHETTI, G., AND ROSING, T. vGreen: A System for Energy-Efficient Management of Virtual Machines. *ACM Transactions on Design Automation of Electronic Systems 16*, 1 (2010), 1–27.

[10] HE, Y., YE, Z., FU, Q., AND ELNIKETY, S. Budget-based Control for Interactive Services with Adaptive Execution. In *Proceedings of the 9th International Conference on Autonomic Computing (ICAC)* (2012), pp. 105–114.

[11] KEPHART, J., AND CHESS, D. The Vision of Autonomic Computing. *Computer 36*, 1 (2003), 41–50.

[12] KREBS, R., MOMM, C., AND KOUNEV, S. Metrics and Techniques for Quantifying Performance Isolation in Cloud Environments. In *Proceedings of the 8th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA)* (2012), pp. 91–100.

[13] KUSIC, D., AND KANDASAMY, N. Risk-Aware Limited Lookahead Control for Dynamic Resource Provisioning in Enterprise Computing Systems. In *Proceedings of the 3rd IEEE International Conference on Autonomic Computing (ICAC)* (2006), pp. 74–83.

[14] LAMA, P., AND ZHOU, X. AROMA: Automated Resource Allocation and Configuration of Mapreduce Environment in the Cloud. In *Proceedings of the 9th International Conference on Autonomic Computing (ICAC)* (2012), pp. 63–72.

[15] MAZZUCCO, M. Towards Autonomic Service Provisioning Systems. In *Proceedings of the 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGRID)* (2010), pp. 273–282.

[16] MENASCE, D. Two-level Iterative Queuing Modeling of Software Contention. In *Proceedings of the 10th IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunications Systems (MASCOTS)* (2002), pp. 267–276.

[17] ROYTMAN, A., KANSAL, A., GOVINDAN, S., LIU, J., AND NATH, S. PACMan: Performance Aware Virtual Machine Consolidation. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC)* (2013), pp. 83–94.

[18] SHARMA, U., SHENOY, P., AND TOWSLEY, D. F. Provisioning Multi-tier Cloud Applications using Statistical Bounds on Sojourn Time. In *Proceedings of the 9th International Conference on Autonomic Computing (ICAC)* (2012), pp. 43–52.

[19] SINGH, R., SHARMA, U., CECCHET, E., AND SHENOY, P. Autonomic Mix-aware Provisioning for Non-stationary Data Center Workloads. In *Proceedings of the 7th International Conference on Autonomic Computing (ICAC)* (2010), pp. 21–30.

[20] SNYDER, B. Server Virtualization has Stalled, Despite the Hype. *InfoWorld* (2010).

[21] SPICUGLIA, S., BJÖERKQVIST, M., CHEN, L. Y., SERAZZI, G., BINDER, W., AND SMIRNI, E. On Load Balancing: a Mix-aware Algorithm for Heterogeneous Systems. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering (ICPE)* (2013), pp. 71–76.

[22] URGAONKAR, B., AND CHANDRA, A. Dynamic Provisioning of Multi-tier Internet Applications. In *Proceedings of the 2nd International Conference on Automatic Computing (ICAC)* (2005), pp. 217–228.

[23] VILLELA, D., PRADHAN, P., AND RUBENSTEIN, D. Provisioning Servers in the Application Tier for e-commerce Systems. *ACM Transactions on Internet Technology 7*, 1 (2007).

[24] WANG, R., AND KANDASAMY, N. On the Design of Decentralized Control Architectures for Workload Consolidation in Large-scale Server Clusters. In *Proceedings of the 9th International Conference on Autonomic Computing (ICAC)* (2012), pp. 115–124.

[25] ZHOU, X., AND IPPOLITI, D. Resource Allocation Optimization for Quantitative Service Differentiation on Server Clusters. *Journal of Parallel and Distributed Computing 68*, 9 (2008), 1250–1262.

PCP: A Generalized Approach to Optimizing Performance Under Power Constraints through Resource Management

Henry Hoffmann
University of Chicago, Chicago USA

Martina Maggio
Lund University, Lund Sweden

Abstract

Many computing systems are constrained by power budgets. While they could temporarily draw more power, doing so creates unsustainable temperatures and unwanted electricity consumption. Developing systems that operate within power budgets is a constrained optimization problem: configuring the components within the system to maximize performance while maintaining sustainable power consumption. This is a challenging problem because many different components within a system affect power/performance tradeoffs and they interact in complex ways. Prior approaches address these challenges by fixing a set of components and designing a power budgeting framework that manages only that one set of components. If new components become available, then this framework must be redesigned and reimplemented. This paper presents PCP, a general solution to the power budgeting problem that works with arbitrary sets of components, even if they are not known at design time or change during runtime. To demonstrate PCP, we implement it in software and deploy it on a Linux/x86 platform.

1 Introduction

Modern computing systems are constrained by *dark silicon*, the abundance of transistors enables processors to draw more power than they can safely sustain [9, 39]. For example, the Exynos 5 processor (in the Samsung Galaxy S4 phone) has a 5.5W peak power that is nearly $2\times$ the maximum sustainable heat dissipation, limiting peak speed to less than 1 second [36]. At the other end of the spectrum, the next generation of exascale supercomputers is predicted to be constrained by an operating budget of approximately 20 MW [2]. In addition, Microsoft was recently fined for not using *enough* power and violating an agreement with a utility company [12]. Executing in these systems requires solving a constrained optimization problem: maintaining the power budget, while maximizing performance within the power constraint.

Many separate components contribute to total power consumption and various techniques have been proposed to manage individual components. For example, management systems exist for core allocation [27], dynamic voltage and frequency scaling (DVFS) [30, 42], processor idling [11], cache [1], DRAM [7, 47], and disk [25].

Of course, different applications have different needs and coordinated management of several components provides better performance within a power budget [19, 29]. Examples that coordinate components include those that handle cores and DVFS [33, 45], DVFS and memory speed [6, 10, 26], and thread scheduling and DVFS [44]. Unfortunately, prior multi-component management approaches are not general in terms of the components under control. Instead, they *fix* a specific set of components, whose interactions are known at design time and they do not permit this set of components to change. If new components become available or existing components are disabled, these power management approaches will either (1) deliver poor performance or (2) require redesign and reimplementation. Thus, *there is a need for a general approach to multi-component power management that continues to deliver maximum performance for a given power budget even as new components become available or existing components are disabled.*

Challenges

A generalized power management system must address three challenges:

Unknowns: Prior approaches rely on rigorous models for either the specific machine under control (*e.g.,* a particular smartphone [37]) or for a specific application and platform (*e.g.,* a web browser on a mobile device [38]). A generalized power management system, however, must either construct its models on the fly or compensate for inaccuracies and unknowns in general models.

Interaction: System components interact to produce a

complex (often nonlinear) effect on power and performance. If individual components are controlled separately, their interaction can lead to suboptimal behavior, even when these separate controllers are individually optimal [16]. Thus, a generalized power management system must coordinate all available components even if they are not known at design time or vary at runtime.

Optimization: A power manager must not exceed the power budget, yet must also deliver the best possible performance for a given budget. A generalized approach must not sacrifice too much performance for generality.

Contributions

This paper addresses the above challenges to produce PCP, short for Power Constraints with maximum Performance, a machine-level power management system that is general with respect to the components it manages. PCP uses feedback control to ensure the power budget is not exceeded. The controller is augmented with an *estimator*, which addresses the challenge of unknowns (see Section 3.3), and a *translator*, which addresses the challenges of interaction and optimization (see Section 3.4). The estimator dynamically tailors control to a particular application, the controller then produces a generic control signal, and the translator turns this signal into a configuration of available components that meets the power budget while providing close to maximum performance.

We implement PCP and test it on a Linux/x86 system. The results show that PCP is:

General: PCP has low overhead (as shown in Section 4.2). Furthermore, components can be added and removed at runtime and PCP automatically adapts to the new conditions. (See Section 4.3).

Accurate: PCP meets the power budgets with low errors. Across all machines and all benchmarks, PCP keeps the average power within 2% of the budget. PCP maintains good results across a range of power budgets from small to large. For small budgets (5% of maximum power), PCP's average error is slightly over 1% of the budget. For all other targets, PCP's average error is less than 1%. (See Section 4.4).

Efficient: Given a power budget, PCP achieves close to the maximum possible performance. We compare PCP to an oracle and find it achieves close to optimal performance for a range of power targets. For the smallest target, the average performance is 92% of the oracle. For all other targets, performance is greater than 95% of the oracle. (See Section 4.5).

The paper is organized as follows. Section 2 compares PCP with prior research. Section 3 discusses PCP's design. Section 4 evaluates PCP, providing experimental evidence for our claims. Finally, Section 5 concludes the paper.

2 Related Work

We describe related work building systems that solve constrained optimization problems in the power and performance dimensions.

Some approaches provide performance guarantees and minimize power consumption. Examples exist at the datacenter-level [21, 40], and machine-level. At the machine-level, techniques provide performance and minimize power by managing DVFS in the processor [45], assignment of cores to an application [27], caches [1], DRAM [47], and disks [25]. Other approaches coordinate multiple components within a machine. For example, Li et al. manage memory and processor [26], while Dubach et al. demonstrate a method for coordinating a large collection of microarchitectural features [8]. Bitirgen et al. coordinate DVFS, cache, and memory bandwidth [4]. METE is an adaptive control system which manages cores, DVFS, and off-chip bandwidth to provide performance guarantees [35]. Gu and Nahrstedt provide a general approach to guarantee performance for multimedia applications in a distributed environment [14]. Hoffmann et al. provide a general technique for coordinating components to meet performance constraints [20]. These solutions provide performance guarantees, but do not guarantee power consumption.

Other approaches guarantee power consumption while maximizing performance subject to this constraint. Such techniques are important for avoiding power overload [9, 39]. Datacenter-level solutions include those proposed by Wang et al. [41] and Raghavendra et al. [31]. These solutions are hierarchical and require some machine-level power management scheme. Machine-level systems for guaranteeing power have been developed to manage DVFS for a processor [23], per-core DVFS in a multicore [22], processor idle-time [11], and DRAM [7]. Other approaches coordinate management of multiple components including processor and DRAM [6, 10], processors and core allocation [33], and combining DVFS and thread scheduling [32, 44].

Like many of the above approaches, PCP uses control theory to manage the dynamic behavior of the machine. Control theory is a general *technique* [15], but implementations are often specific to a particular machine. Some prior approaches provide generality by implementing control systems at the middleware layer [13, 24, 46]. The middleware handles the complicated construction of the control system, but the application writer must be aware of the components on the system. PCP differs as it does not require application programmer input at all.

PCP is a novel machine-level power control approach that coordinates multiple components. PCP is distinguished by the fact that the components it manages are not known at design time and may change during run-

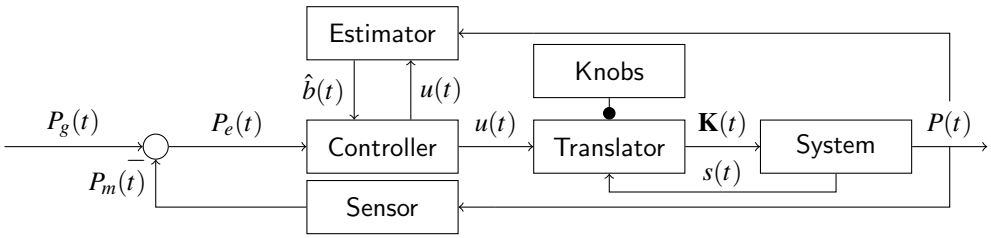


Figure 1: Block diagram representing the overall control and scheme.

time.

3 Runtime Control Framework

The PCP runtime is illustrated in Figure 1. The runtime begins with a simple model of the System under control, including the available components, or Knobs, affecting power and performance. A Controller measures power consumption and produces a generic control signal indicating available overhead in the power budget. An Estimator dynamically tunes the controller to the specific application and system under control and accounts for approximations in the initial model. A Translator turns the generic control signal into a performance-optimal component configuration that achieves the power budget. Each of these modules is described below.

3.1 System

The System block in Figure 1 describes available system components, from the controller's perspective. PCP uses a discrete time, linear model of the system where the index t measures time (in discrete intervals). The system has n separate components, or *knobs*, that affect power and performance. The system's input is a *configuration*, or vector of values $\mathbf{K}(t) \in \mathbb{R}^n$, each of which represents a setting for one of the knobs. The system has a *baseline* power consumption b representing the lowest possible power. We denote the system's power consumption as $P(\mathbf{K}(t))$, measured as a proportion of b. The speed of the system is $S(\mathbf{K}(t))$. The system's outputs are the current *power consumption* $P(t)$ and the current *performance* measure $s(t)$ (for speed). $P(t)$ is the average power consumption in the interval between $t-1$ and t:

$$P(t) = bP(\mathbf{K}(t-1)) \tag{1}$$

This simple linear model is used to derive a controller in Section 3.2. The estimator (Section 3.3) compensates for the inherent approximations in the model.

PCP requires power and performance feedback, but is agnostic about the source. Power feedback can come from a model or direct power measurements. Performance feedback can come from any number of appropriate hardware counters, the only requirement is that the metric increase with increasing speed.

3.2 Controller

The Controller in Figure 1 eliminates the power error $P_e(t) = P_g(t) - P_m(t)$, between the power budget, or goal $P_g(t)$, and the actual measured power $P_m(t)$. To do so, the controller computes a control signal. In a typical controller, this signal would directly specify settings for the available knobs $\mathbf{K}$; however, such an approach does not generalize because it ties the controller's design directly to the system under control. Instead, PCP computes a generic control signal $u(t)$, independent of the specific knobs in the system. $u(t)$ represents an allowable power consumption over the baseline b; *i.e.,* $u(t) = P(\mathbf{K}(t))$. For example, $u(t) = 1.5$ indicates that the controller allows 50% greater power consumption than baseline. The translator maps this control signal into knob settings (see Section 3.4). Therefore, given $P_e(t)$ and b at time t, the controller calculates a new signal $u(t)$. This generic signal is derived following standard practice resulting in the Integral controller:

$$u(t) = u(t-1) + P_e(t)/b \tag{2}$$

3.3 Estimator

The b parameter has a profound effect on PCP's behavior. Although PCP models it as a constant, we know that its true value varies as a function of the hardware and the application (or even phases within the application). Therefore, b represents a key *unknown* corresponding to the first challenge listed in Section 1. The Estimator in Figure 1 is responsible for overcoming this unknown by continually estimating its true value as $\hat{b}(t)$. Adding the estimator makes the control scheme *adaptive*, in the sense that it automatically adjusts, not only to the feedback, but also to varying operating points and unknowns.

PCP estimates b using a Kalman filter [43] based on the time-varying model:

$$\begin{aligned} b(t) &= b(t-1) + \delta b(t) \\ P(t) &= u(t-1)\,b(t-1) + \delta P(t) \end{aligned} \tag{3}$$

which describes the variation of b and P subject to both disturbance and noise (respectively δb and δP).

Denoting the system power variance and measurement variance as $q_b(t)$ and r_b, the Kalman filter formulation is

$$\begin{aligned} \hat{b}^-(t) &= \hat{b}(t-1) \\ e_b^-(t) &= e_b(t-1)+q_b(t) \\ k_b(t) &= \frac{e_b^-(t)\,u(t)}{[u(t)]^2\,e_b^-(t)+r_b} \\ \hat{b}(t) &= \hat{b}^-(t)+k_b(t)\,[P(t)-u(t)\hat{b}^-(t)] \\ e_b(t) &= [1-k_b(t)\,u(t-1)]\,e_b^-(t) \end{aligned} \tag{4}$$

where $k_b(t)$ is the Kalman gain for the system power consumption, $\hat{b}^-(t)$ is the *a priori* estimate of $b(t)$ and $\hat{b}(t)$ is the *a posteriori* one, and $e_b^-(t)$ is the *a priori* estimate of the error variance while $e_b(t)$ is the *a posteriori* one. PCP uses a Kalman filter because it produces a statistically optimal estimate of the system's parameters, and is provably exponentially convergent [5]. Note that the only required parameter is r_b, the measurement noise, which is a property of the feedback mechanism.

3.4 Translator

The Translator in Figure 1 converts the generic control signal $u(t)$ into a component configuration $\mathbf{K}(t)$ at time t. The control signal is a continuous signal which must be translated into discrete knob settings. This problem is challenging because any given power signal may be achieved by multiple combinations of components. The Translator must find the combination that meets the budget while maximizing performance.

PCP schedules configurations over the τ time units between now and the next power measurement. PCP assigns τ_c time units to each configuration where $\sum_c \tau_c = \tau$. The translator schedules these configurations to maximize performance while guaranteeing the power budget, specified by the control signal, is not exceeded. Denoting a configuration of knobs as $K_c(t)$ with corresponding speed $S(K_c(t))$ and power $P(K_c(t))$, then PCP solves the following optimization problem:

$$\text{maximize} \quad \sum_c \tau_c \cdot S(K_c(t)) \tag{5}$$

$$\text{subject to} \quad \sum_c \tau_c \cdot P(K_c(t))/\hat{b}(t) \leq u(t) \tag{6}$$

$$\sum_c \tau_c = \tau \tag{7}$$

$$t \geq \tau_c \geq 0, \quad \forall\, c \tag{8}$$

Equations 5–8 assign values for all τ_c to maximize performance (Equation 5) subject to the constraints that the power signal is not exceeded (Equation 6). The final two constraints (Equations 7 and 8) ensure that the time spent in all configurations is non-negative and does not exceed the time of the next measurement. This type of optimization problem is similar to others that have arisen for real-time systems which minimize energy while meeting a performance constraint. Several heuristic solutions have been developed that apply to both performance and power constraints and provide near-optimal behavior in practice [17].

We note that as components enter and leave the system, the power and performance of various configurations may change. This is especially true if different groups of components interact in non-linear ways. PCP accounts for this variability by continually estimating $S(K_c(t))$ and $P(K_c(t))$ online. Both quantities are estimated using Kalman filter formulations similar to that of Equation 4 (these formulations are omitted for space). This estimation allows PCP to adapt to changing sets of components at runtime. In practice, PCP first estimates the performance and power consumption of the previous configuration using Kalman filters, and then solves Equations 5–8 using the new estimate to determine the resource allocation for the next time step.

3.5 Summary

PCP meets power budgets while optimizing performance. Its generalized system model is independent of any particular set of hardware components. PCP's controller ensures that power budgets are met. The estimator overcomes the challenge of handling unknowns, while the translator handles the challenges of component interaction and performance optimization. PCP can incorporate new knobs as they become available at runtime, and it can be deployed on new systems without redesign.

4 Evaluation

4.1 Experimental Setup

We describe the systems and applications used in our evaluation.

Systems

To demonstrate PCP, we deploy it on a real Linux/x86 system, a single socket Intel E5-1650 with 6 cores, 1 memory controller and 12 DVFS settings from 1.2 — 3.2 GHz. It supports hyper-threading and TurboBoost. In addition, it allows suspension of the current application to enter a low-power idle state. Idling consumes 85% of the lowest measured active power (85W to 100W). The highest measured power consumption is 235W. To measure power consumption the machine supports hardware power measurement at 1 ms intervals [34].

Benchmarks

Our benchmarks consist of the 13 PARSECs [3] plus Dijkstra and STREAM [28]. PARSEC has a mix of im-

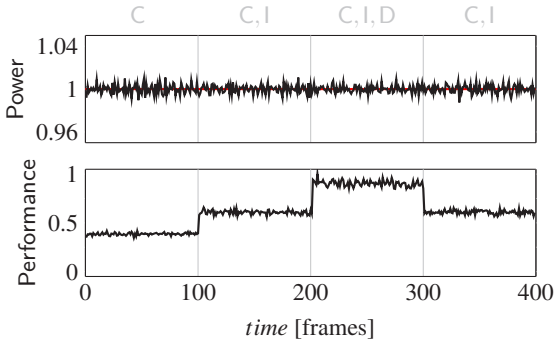


Figure 2: Experiment changing actuators.

portant, emerging multicore workloads. Dijkstra is a parallel implementation of single-source shortest paths on a large, dense graph developed for this paper. STREAM tests memory performance. These last two benchmarks test PCP's ability to handle resource limited applications. Dijkstra has some parallelism, but does not scale well. STREAM is generally memory limited, but needs sufficient compute resources to saturate the available memory bandwidth.

Although PCP supports a range of performance feedback metrics (see Section 3), in this paper we measure application performance as instructions per second and collect this data with hardware performance counters. In general, however, PCP could work with any performance metric that increases with increasing application speed and decreases as application performance decreases; *e.g.,* application specific performance metrics [18].

4.2 Overhead

We evaluate PCP's overhead by running all benchmark applications with PCP, but disabling its ability to actually change component settings. Thus, the runtime executes and consumes resources, but cannot have any positive effect. We compare execution time and power consumption to the application running without our runtime. For most applications, there is no measurable difference. The largest difference in performance is 2.2%, measured for the *fluidanimate* benchmark, while the power consumption is within the run to run variation for all benchmarks ($< 1\%$). We find this overhead acceptable. All measurements in this section include the overhead and impact of the runtime system.

4.3 Changing Actuators

To demonstrate the claim of generality, *we add and remove components at runtime and demonstrate that PCP system automatically adapts to the new conditions*. We run the raytrace benchmark for 400 time units, where each unit is a frame. During execution, the set of available knobs changes. We distinguish between four different intervals of 100 time units. During the first interval, PCP only adjusts the number of cores. During the second interval, idle time becomes available. In the third interval, DVFS is also available. In the last interval, DVFS is disabled (causing the performance loss that occurs at time 300).

Figure 2 shows the results. The power consumption is very close to the setpoint (1 in the figure); the percentage difference is less than 1%. As different actuators become available, PCP takes advantage of them to improve performance without exceeding the power budget. The figure shows performance normalized to the maximum achievable for the power budget.

4.4 Stability and Accuracy

We demonstrate PCP's stability and accuracy by running each benchmark on our target system and telling PCP to maintain a power budget. For this experiment, we set the power budget halfway between the minimum and maximum achievable power. This middling power budget is one of the toughest to meet in practice because there are many combinations of components which can achieve the same power. PCP must determine the highest performance configuration at runtime.

We quantify the stability and accuracy of PCP by measuring the power consumption at each control interval and calculating the mean absolute percentage error (MAPE):

$$MAPE = \frac{1}{n}\sum_{k=1}^{n} \begin{cases} p_m(k) > p_\ell : & \left|\frac{p_m(k)-p_\ell}{p_m(k)}\right| \cdot 100\% \\ p_m(k) \leq p_\ell : & 0 \end{cases} \quad (9)$$

Low MAPE (expressed as a percentage) indicates that the power budget is stable and at or below the desired value. Note that we do not penalize PCP for operating below the power budget, but this may result in suboptimal performance. We quantify performance in the next section. Finally, when calculating MAPE, we subtract idle power from the measured power. Doing so quantifies the contribution of the control system to accuracy and stability. Leaving the idle power in the calculation would result in lower MAPE values.

The results are illustrated in Figure 3. The figure shows each benchmark on the x-axis and the MAPE (as a percentage) on the y-axis. The measured errors are quite low in general, and all are under 8%. These results indicate that PCP is stable and accurate across a range of benchmark applications. Those benchmarks with higher MAPE tend to have multiple phases of computation and the errors occur when PCP is optimizing

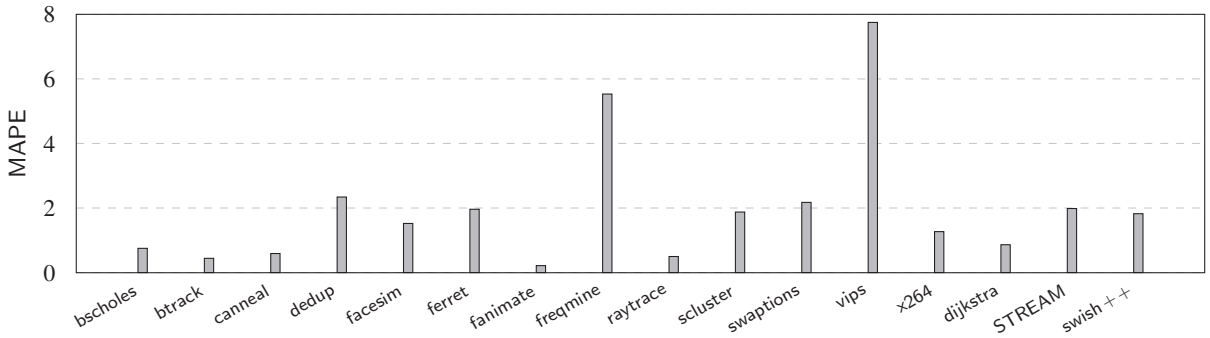


Figure 3: Mean average percentage error.

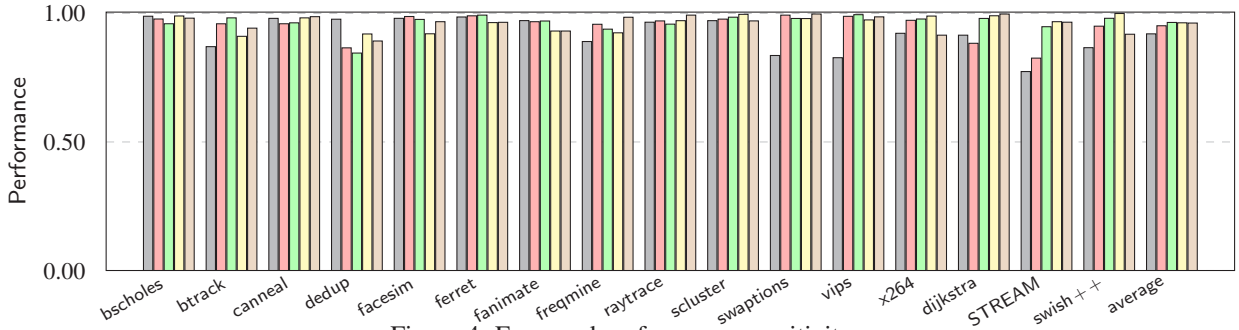


Figure 4: Error and performance sensitivity.

across phases. Even for these applications, however, PCP produces good results, which are consistent with the low observed error in Figure 2.

4.5 Error and Performance Sensitivity

We further test PCP's efficiency by measuring the sensitivity of performance to the power target. We run each of our benchmarks with 5 different power targets. Each power target is x% between the minimum and maximum achievable where $x \in \{5, 25, 50, 75, 95\}$. For each power target, we deploy the application on the machine and PCP adjusts resource usage to keep power consumption at or below the target. We measure the error and the performance for each target and report results in Figure 4.

The figure shows the normalized performance (lower plot) on the respective y-axes for each of the 5 power targets. Benchmark name is shown on the x-axis. The results demonstrate PCP's efficiency across a range of power targets. In general, the lowest performance is achieved for the the 5% target; however, the average performance of 0.92 is still close to optimal. All other targets have an average performance greater than 0.95. The worst single achieved performance comes from STREAM at the 5% target, which has a normalized performance of 0.77. This low result comes from the time taken to learn to allocate STREAM the minimum resources on this machine. The near-optimal performance across a range of benchmarks and power targets demonstrate PCP's efficiency.

Although the accuracy results are omitted for space, they are consistent with those reported in Figure 3. The highest MAPE values are achieved for the 50% target. All other targets are lower in general.

5 Conclusion

This paper presents PCP, a power budgeting system, which guarantees power consumption while maximizing performance subject to the power budget. PCP's distinguishing feature is its generality; *i.e.,* the design is independent of any particular set of components that could be used to manage power/performance tradeoffs. PCP overcomes the four challenges of handling dynamics, accounting for unknowns, managing component interaction, and optimizing performance. We have implemented PCP and deployed it on several hardware platforms and demonstrated its generality, portability, accuracy, and performance. Widespread adoption of PCP would make it easy to quickly develop power management systems as new hardware components become available.

Acknowledgements

Henry Hoffmann is grateful for support from the U.S. Dept. of Energy (under DOE DE-AC02-06CH11357).

References

[1] R. Balasubramonian et al. “Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures”. In: *MICRO*. 2000.

[2] K. Bergman et al. *ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems Peter Kogge, Editor & Study Lead*. 2008.

[3] C. Bienia et al. “The PARSEC Benchmark Suite: Characterization and Architectural Implications”. In: *PACT*. 2008.

[4] R. Bitirgen et al. “Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach”. In: *MICRO*. 2008.

[5] L. Cao and H. M. Schwartz. “Exponential convergence of the Kalman filter based parameter estimation algorithm”. In: *Intl. Journal of Adaptive Control and Signal Processing* 17.10 (2003).

[6] J. Chen and L. K. John. “Predictive coordination of multiple on-chip resources for chip multiprocessors”. In: *ICS*. 2011.

[7] B. Diniz et al. “Limiting the power consumption of main memory”. In: *ISCA*. 2007.

[8] C. Dubach et al. “A Predictive Model for Dynamic Microarchitectural Adaptivity Control”. In: *MICRO*. 2010.

[9] H. Esmaeilzadeh et al. “Dark silicon and the end of multicore scaling”. In: *ISCA*. 2011.

[10] W. Felter et al. “A performance-conserving approach for reducing peak power consumption in server systems”. In: *ICS*. 2005.

[11] A. Gandhi et al. “Power capping via forced idleness”. In: *Workshop on Energy-Efficient Design*. Austin, TX, 2009.

[12] J. Glanz. “Data Barns in a Farm Town, Gobbling Power and Flexing Muscle”. In: *The New York Times* (September 23, 2012).

[13] A. Goel et al. “SWiFT: A Feedback Control and Dynamic Reconfiguration Toolkit”. In: *2nd USENIX Windows NT Symposium*. 1998.

[14] X. Gu and K. Nahrstedt. “Dynamic QoS-aware multimedia service configuration in ubiquitous computing environments”. In: *ICDCS*. 2002.

[15] J. L. Hellerstein et al. *Feedback Control of Computing Systems*. John Wiley & Sons, 2004.

[16] J. Heo and T. Abdelzaher. “AdaptGuard: guarding adaptive systems from instability”. In: *ICAC*. 2009.

[17] H. Hoffmann. “Racing vs. Pacing to Idle: A Comparison of Heuristics for Energy-aware Resource Allocation”. In: *HotPower*. 2013.

[18] H. Hoffmann et al. “Application heartbeats: a generic interface for specifying program performance and goals in autonomous computing environments”. In: *ICAC*. 2010.

[19] H. Hoffmann et al. “Self-aware computing in the Angstrom processor”. In: *DAC*. 2012.

[20] H. Hoffmann et al. “A Generalized Software Framework for Accurate and Efficient Managment of Performance Goals”. In: *EMSOFT*. 2013.

[21] T. Horvath et al. “Dynamic Voltage Scaling in Multitier Web Servers with End-to-End Delay Control”. In: *Computers, IEEE Transactions on* 56.4 (2007).

[22] C. Isci et al. “An Analysis of Efficient Multi-Core Global Power Management Policies: Maximizing Performance for a Given Power Budget”. In: *MICRO*. 2006.

[23] C. Lefurgy et al. “Power capping: a prelude to power shifting”. In: *Cluster Computing* 11.2 (2008).

[24] B. Li and K. Nahrstedt. “A control-based middleware framework for quality-of-service adaptations”. In: *IEEE Journal on Selected Areas in Communications* 17.9 (1999).

[25] X. Li et al. “Performance directed energy management for main memory and disks”. In: *Trans. Storage* 1.3 (2005).

[26] X. Li et al. “Cross-component energy management: Joint adaptation of processor and memory”. In: *ACM Trans. Archit. Code Optim.* 4.3 (2007).

[27] M. Maggio et al. “Controlling software applications via resource allocation within the Heartbeats framework”. In: *CDC*. 2010.

[28] J. D. McCalpin. “Memory Bandwidth and Machine Balance in Current High Performance Computers”. In: *IEEE TCCA Newsletter* (1995).

[29] D. Meisner et al. “Power management of online data-intensive services”. In: *ISCA* (2011).

[30] T. Pering et al. “The simulation and evaluation of dynamic voltage scaling algorithms”. In: *ISLPED*. 1998.

[31] R. Raghavendra et al. “No ”power” struggles: coordinated multi-level power management for the data center”. In: *ASPLOS*. 2008.

[32] K. K. Rangan et al. “Thread motion: fine-grained power management for multi-core systems”. In: *ISCA*. 2009.

[33] S. Reda et al. “Adaptive Power Capping for Servers with Multithreaded Workloads”. In: *Micro, IEEE* 32.5 (2012).

[34] E. Rotem et al. “Power management architecture of the 2nd generation Intel Core microarchitecture, formerly codenamed Sandy Bridge”. In: *Hot Chips*. 2011.

[35] A. Sharifi et al. “METE: meeting end-to-end QoS in multicores through system-wide resource management”. In: *SIGMETRICS*. 2011.

[36] Y. Shin et al. “28nm High- Metal-Gate Heterogeneous Quad-Core CPUs for High-Performance and Energy-Efficient Mobile Application Processor”. In: *ISSCC*. 2013.

[37] A. Shye et al. “Into the wild: studying real user activity patterns to guide power optimizations for mobile architectures”. In: *MICRO*. 2009.

[38] N. Thiagarajan et al. “Who killed my battery?: analyzing mobile browser energy consumption”. In: *WWW*. 2012.

[39] G. Venkatesh et al. “Conservation cores: reducing the energy of mature computations”. In: *ASPLOS*. 2010.

[40] A. Verma et al. “Server workload analysis for power minimization using consolidation”. In: *USENIX Annual technical conference*. 2009.

[41] X. Wang et al. “MIMO Power Control for High-Density Servers in an Enclosure”. In: *IEEE Transactions on Parallel and Distributed Systems* 21.10 (2010).

[42] M. Weiser et al. “Scheduling for reduced CPU energy”. In: *Mobile Computing* (1996).

[43] G. Welch and G. Bishop. *An Introduction to the Kalman Filter*. Tech. rep. TR 95-041. UNC Chapel Hill, Department of Computer Science, 2006.

[44] J. A. Winter et al. “Scalable thread scheduling and global power management for heterogeneous many-core architectures”. In: *PACT*. 2010.

[45] Q. Wu et al. “Formal online methods for voltage/frequency control in multiple clock domain microprocessors”. In: *ASPLOS*. 2004.

[46] R. Zhang et al. “ControlWare: A middleware architecture for Feedback Control of Software Performance”. In: *ICDCS*. 2002.

[47] H. Zheng et al. “Mini-rank: Adaptive DRAM architecture for improving memory power efficiency”. In: *MICRO*. 2008.

Coordinating Liquid and Free Air Cooling with Workload Allocation for Data Center Power Minimization

Li Li, Wenli Zheng, Xiaodong Wang, and Xiaorui Wang
Power-Aware Computer System (PACS) Laboratory
Department of Electrical and Computer Engineering
The Ohio State University, Columbus, OH 43210
{*li.2251, zheng.691, wang.3570, wang.3596*}*@osu.edu*

Abstract

Data centers are seeking more efficient cooling techniques to reduce their operating expenses, because cooling can account for 30-40% of the power consumption of a data center. Recently, liquid cooling has emerged as a promising alternative to traditional air cooling, because it can help eliminate undesired air recirculation. Another emerging technology is free air cooling, which saves chiller power by utilizing outside cold air for cooling. Some existing data centers have already started to adopt both liquid and free air cooling techniques for significantly improved cooling efficiency and more data centers are expected to follow.

In this paper, we propose SmartCool, a power optimization scheme that effectively coordinates different cooling techniques and dynamically manages workload allocation for jointly optimized cooling and server power. In sharp contrast to the existing work that addresses different cooling techniques in an isolated manner, SmartCool systematically formulates the integration of different cooling systems as a constrained optimization problem. Furthermore, since geo-distributed data centers have different ambient temperatures, SmartCool dynamically dispatches the incoming requests among a network of data centers with heterogeneous cooling systems to best leverage the high efficiency of free cooling. A light-weight heuristic algorithm is proposed to achieve a near-optimal solution with a low run-time overhead. We evaluate SmartCool both in simulation and on a hardware testbed. The results show that SmartCool outperforms two state-of-the-art baselines by having a 38% more power savings.

1 Introduction

In recent years, high power consumption has become a serious concern in operating large-scale data centers. For example, a report from Environmental Protection Agency (EPA) estimated that the total energy consumption from data centers in the US was over 100 billion kWh in 2011. Among the total power consumed by a data center, cooling power can account for 30-40% [14][2]. As new high-density servers (e.g., blade servers) are increasingly being deployed in data centers, it is important for the cooling systems to more effectively remove the heat. However, with the high-density servers being installed, the traditional computer room air conditioner (CRAC) system might not be efficient enough, as its Power Usage Effectiveness (PUE) is around 2.0 or higher. PUE is defined as the ratio of the total energy consumption of a data center over energy consumed by the IT equipment such as servers. With a high PUE, the cooling power consumption of a data center can grow tremendously as high-density servers being deployed, which not only increases the operating cost, but also causes negative environmental impact. Therefore, data centers are in an urgent need to find higher-efficient cooling techniques to reduce PUE.

Two new cooling techniques have recently been proposed to increase the cooling efficiency and lower the PUE of a data center. The first one is liquid cooling, which conducts coolant through pipes to some heat exchange devices that are attached to the IT equipments, such that the generated heat can be directly taken away by the coolant. The second one, which is referred to as free air cooling [7], exploits the relatively cold air outside the data center for cooling and thus saves the power of chilling the hot air returned from the IT equipment. Although both of the two cooling techniques highly increase the cooling efficiency of data centers, each technique has its own limitations. The liquid cooling approach requires additional ancillary facilities (e.g., the valves and pipes) and maintenance, which can increase the capital investment when being deployed in a large scale. The free air cooling technique requires a low outside air temperature, which might not be available all the time in a year. In order to mitigate the problems, hy-

brid cooling system, which is composed with the liquid cooling, the free air cooling and the traditional CRAC air cooling, can be used to lower the cooling cost and ensure the cooling availability. Several hybrid-cooled data centers have been put into production. For example, the CERN data center located in Europe adopts a hybrid cooling system with both liquid-cooling and traditional CRAC cooling systems, in which about 9% of the servers are liquid-cooled [4].

However, efficiently operating such a hybrid cooling system is not a trivial task. Currently, existing data centers that adopt multiple cooling techniques commonly use some preset outside temperature thresholds to switch between different cooling systems, regardless of the time-varying workload. Such a simplistic solution can often lead to unnecessarily low cooling efficiencies. Although some previous studies [11][35] have proposed to intelligently distribute the workload across the servers and manage the cooling system according to the real-time workload to avoid over-cooling, they address only one certain cooling technology and thus the resulted workload distribution might not be optimal for the hybrid cooling system. To the best of our knowledge, no prior research has been done for efficiently coordinating multiple cooling techniques in a hybrid-cooled data center. In addition, for a network of data centers that are geographically distributed, there exist some works focusing on balancing the workload or reducing the server power cost [16][27], but none of them minimize the cooling power consumption, especially when the data centers have heterogeneous cooling systems. In order to minimize the power consumption of a hybrid-cooled data center, we need to face several new challenges. First, the different characteristics of these three cooling systems (liquid cooling, free air cooling and the traditional CRAC air cooling) demand for a systematic approach to coordinate them effectively. Second, workload distribution in such a hybrid-cooled data center needs to be carefully planned in order to jointly minimize the cooling and server power consumption. Third, due to the different local temperatures, a novel workload distribution and cooling management approach is needed for data centers that are geographically distributed at different locations, in order to better utilize free air cooling in the hybrid cooling system more efficiently.

In this paper, we propose *SmartCool*, a power optimization scheme to optimize the total power consumption of a hybrid-cooled data center by intelligently managing the hybrid cooling system and distributing the workload. We first formulate the power optimization problem for a single data center, which can then be solved with a widely adopted optimization technique. We then extend the power optimization scheme to fit a network of geo-distributed data centers. To reduce the computational overhead, we propose a light-weight algorithm to solve the optimization problem for the geo-distributed data centers. Specifically, this paper has the following major contributions:

- More and more data centers are on their way to adopt high-efficient cooling techniques, but many data centers heavily rely on simplistic solutions to separately manage their cooling systems, which often lead to an unnecessarily low cooling efficiency. In this paper, we propose to address an increasingly important problem: Intelligent coordination of cooling systems for jointly minimized cooling and server power in a data center.
- We formulate the cooling management in a hybrid-cooled data center with liquid cooling, free air cooling, and traditional CRAC air cooling, as a constrained power optimization problem to minimize the total power consumption. SmartCool features a novel air recirculation model developed based on computational fluid dynamics (CFD).
- To best leverage the high efficiency of free cooling in geo-distributed data centers that have different ambient temperatures, we extend our optimization formulation to dynamically dispatch the incoming requests among a network of data centers with heterogeneous cooling systems. A light-weight heuristic algorithm is proposed to achieve a near-optimal solution with a low run-time overhead.
- We evaluate SmartCool both in simulation and on a hardware testbed with real-world workload and temperature traces. The results show that SmartCool outperforms two state-of-the-art baselines by saving 38% more power consumption.

The rest of the paper is organized as follows. We review the related work in Section 2, and introduce the background of different cooling technologies in Section 3. Section 4 formulates power optimization problem for a single hybrid-cooled data center, which is extended for geo-distributed data centers in Section 5 with a light-weight algorithm. We present our simulation results in Section 6 and the hardware experiment results in Section 7. Finally, Section 8 concludes the paper.

2 Related Work

Minimizing the power consumption of data centers has recently received much attention, such as [27, 11, 18, 29, 31, 32, 13, 28, 17]. In particular, a lot of work has been done to optimize the traditional CRAC air cooling in data centers. For example, Anto et al. [22] construct

a model of a single CRAC unit which offers flexible selection in different heat exchangers and coolants. Zhou et al. [23] propose a computationally efficient multivariable model to capture the effects of CRAC fan speed and supplied air temperature on the rack inlet temperatures. Tang et al. [20] propose a workload scheduling scheme to make the inlet temperatures of all servers as even as possible. A holistic approach is proposed by Chen et al. [30] that integrates the management of IT, power and cooling infrastructures to improve the data center efficiency. In our work, we adopt the modeling process of traditional CRAC cooling system, but coordinate its work with the liquid cooling and free cooling systems as a hybrid-cooling system for the improvement of data center cooling efficiency.

Free air cooling and liquid cooling have also attracted wide research attentions. Christy et al. [9] study two primary free cooling systems, the air economizer and the water economizer. Gebrehiwot et al. [3] study the thermal performance of an air economizer for a modular data center using computational fluid dynamics. Coskun et al. [1] provide a 3D thermal model for liquid cooling, with variable fluid injection rates. Hwang et al. [8] develop an energy model for liquid-cooled data centers based on the thermo-fluid first principles. Differently, our work focuses on the power optimization of hybrid-cooled data centers, by managing the cooling modes and workload distribution.

For geo-distributed data centers, Adnan et al. [16] save the cost of load balancing by utilizing the flexibility of the Service Level Agreements. Related algorithms are developed in [33][34][12] to minimize the total cost of geo-distributed data centers and the environmental impact. Our global workload dispatching strategy minimizes the total power consumption of all the distributed data center, by leveraging the temperature differences among different locations and maximizing the usage of free air cooling.

3 Different Cooling Technologies

Figure 1 illustrates the cooling system of a hybrid-cooled data center, which includes traditional air cooling, liquid cooling and free air cooling. The liquid cooling system uses chiller and cooling tower to provide coolant. Either the CRAC system or the free cooling system can be selected for air cooling. The CRAC system also relies on chiller and cooling tower to provide the coolant, which is then used to absorb heat from the air in the data center. The free cooling system draws outside air into the data center through the Air Handling Unit (AHU) when the outside temperature can meet the cooling requirement.

Traditional CRAC air cooling is the most widely used cooling technology in existing data centers. This

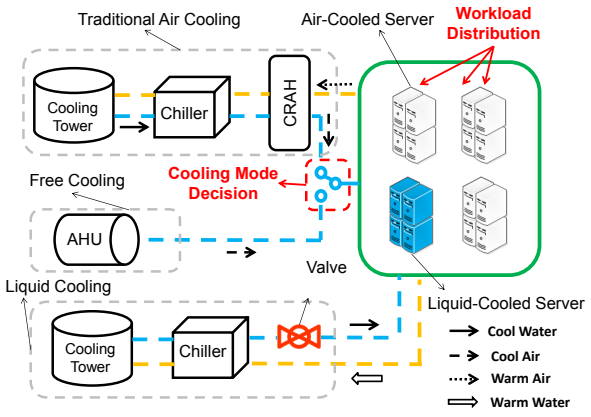


Figure 1: Cooling System of Hybrid-Cooled Data Center. The cold plates used for liquid cooling are installed inside the liquid-cooled servers. The air cooling mode is decided based on the outside temperature.

system deploys several CRAC units in the computer room to supply cold air. The cold air usually goes under the raised floor before joining in the cold aisle through perforated tiles to cool down the servers, as shown in Figure 2. The hot air from the servers is output to the hot aisle and returned to the CRAC system to be cooled and reused. The deployment of cold aisle and hot aisle is used to form isolation between cold and hot airs. However, due to the existence of seams between servers and racks, as well as the space close to the ceiling where there is no isolation, cold air and hot air are often mixed to a certain extent, which decreases the cooling efficiency. The PUE of a data center using CRAC cooling is usually around 2.0 [6].

Liquid cooling technology usually uses coolant (e.g., water) to directly absorb heat from the servers. It can be divided into three categories: *direct liquid cooling* [19], *rack-level liquid cooling* [24] and *submerge cooling* [26]. In *direct liquid cooling*, the microprocessor in a server is directly attached with a cold plate that contains the coolant to absorb heat of the microprocessor, while the other components are still cooled by chilled air flow. *Direct liquid cooling* improves the cooling efficiency by enhancing the heat exchange process. *Rack-level liquid cooling* and *submerge cooling* adopt some other heat exchange devices instead of the cold plate. In this paper, we adopt the direct liquid cooling technology as an example to demonstrate the effectiveness of our solution, due to its low cost.

Free air cooling is a highly efficient cooling approach that uses the cold air outside the data center and saves power by shutting off the chiller system. It is usually utilized within a range of outside temperature and humidity.

Within this range, the outside air can be used for cooling via an air handler fan. The traditional CRAC system is employed by these data centers as the backup cooling system.

4 Power Optimization for a Local Hybrid-cooled Data Center

In this section, we introduce the power models and formulate the total power optimization problem for a local hybrid-cooled data center. We then present how we solve the problem by using computational fluid dynamics (CFD) modeling and optimization techniques.

4.1 Power Models of a Hybrid-cooled Data Center

We use the following models to calculate the server and cooling power consumption in the data center.

- *Server Power Consumption Model*

For a server i, we adopt a widely accepted server power consumption model [11] as:

$$P_i^{server} = W_i \times P_i^{compute} + P_i^{idle} \quad (1)$$

where W_i is the workload handled by server i, in terms of CPU utilization. $P_i^{compute}$ is the maximum computing power when the workload is 100%. P_i^{idle} represents the static idle power consumed by the server. If the workload is 0, the server can be shut down to save power and thus the power consumption is 0. Therefore, the total server power consumption of a data center with N server is

$$P^{server} = \sum_{i=1}^{N} P_i^{server} \quad (2)$$

- *Air Cooling Power Model*

In a hybrid-cooled data center, the components of a liquid-cooled server except for the microprocessor, the rest server components, such as disk and memory, are cooled by the air cooling system, which contributes to the hot air coming out of the server. To characterize this relationship, we assume that in a liquid-cooled server, α percent of the power is consumed by the microprocessor. Therefore, assuming the first M servers among the total N servers are liquid-cooled, we can calculate the total power consumption of all the servers and components that are cooled by the air cooling as:

$$P_{air}^{server} = \sum_{i=M+1}^{N} P_i^{server} + \sum_{i=1}^{M} (1-\alpha) * P_i^{server} \quad (3)$$

The power consumption of the traditional CRAC air cooling system depends on the heat generation (i.e., P_{air}^{server}) and the efficiency of the CRAC system:

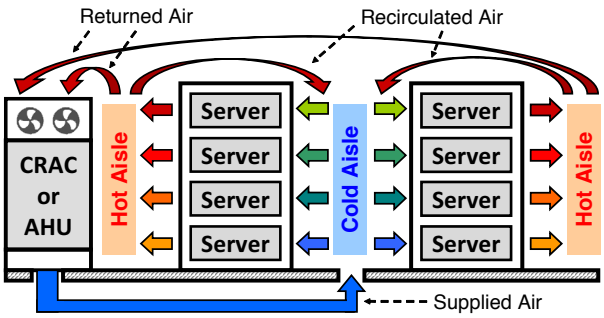


Figure 2: Air circulation in an air-cooled data center. Some hot air can be recirculated to the inlet and mixed with the cold air, degrading the cooling efficiency.

$$P_{CRAC}^{air} = \frac{P_{air}^{server}}{COP^{CRAC}} \quad (4)$$

According to [11], the COP (coefficient of performance, characterizing the cooling efficiency) of a CRAC system can be calculated according to the supplied air temperature T_{sup}:

$$COP_{air} = 0.0068 * T_{sup}^2 + 0.0008 * T_{sup} + 0.458 \quad (5)$$

To avoid overheating the servers, the inlet air temperature of an air-cooled server needs to be bounded by a threshold. Based on a temperature model from [21], we use Equation 6 to first calculate the outlet air temperature, and then get the inlet air temperature with Equation 7:

$$K_i T_{out}^i = \sum_{j=1}^{N} h_{ij} K_j T_{out}^j + (K_i - \sum_{j=1}^{N} h_{ij} K_j) T_{sup} + P_i^{air} \quad (6)$$

$$T_{in}^i = \sum_{j=1}^{N} h_{ji} * (T_{out}^j - T_{sup}) + T_{sup} \quad (7)$$

K_i represents a multiplicative item, $\rho f_i C_p$, where C_p is the specific heat of air. ρ represents the air density, and f_i is the air flow rate to server i. h describes the air recirculation. In Equation 6, the first term characterizes the impact of the air recirculation from server j to server i and the second term models the cooling effect of the supplied air. The third term is the power consumption of server i that heats up the passing cold air. Equation 7 shows that inlet server temperature is determined by the supplied air temperature and the recirculation heat. We explain how to derive h using CFD in Section 4.3.

When the free air cooling method is chosen for the hybrid-cooled data center, the air cooling power is calculated in a different way, according to [7]

$$P_{free}^{air} = (PUE_{free} - 1) * P_{air}^{server} \quad (8)$$

In our experiment, the free cooling PUE is modeled to be proportional to the ambient air temperature according

to [10]. This is because when the outside air temperature is relatively high, more air is needed to take away the heat generated by the servers, and the fan speed of AHU needs to be higher to draw more air.

In our paper, we assume that only one of the two air cooling systems can run at one time in a hybrid-cooled data center. Thus the total air cooling power consumption can be expressed as:

$$P^{air} = \beta P^{air}_{CRAC} + (1-\beta) P^{air}_{free} \quad (9)$$

where β is a binary variable indicating which air cooling system is activated.

- *Liquid Cooling Power Model*

With M liquid-cooled servers and the microprocessor consuming α percent of the server power, we have the liquid cooling power consumption as

$$P^{liquid} = \frac{\sum_{i=1}^{M} \alpha P^{server}_i}{COP^{liquid}} \quad (10)$$

where COP^{liquid} is the COP of the chiller used in the liquid cooling system. Due to the high cooling capacity of liquid medium, the changes of the liquid temperature and the flow rate hardly affect the COP value, and thus COP^{liquid} can be viewed as a constant. To derive a COP that can provide cooling guarantee for all the liquid-cooled servers, we run simple experiments with the worst-case setup by putting all servers to 100% utilization, and adjust the chiller set point and flow rate of the cold plate to ensure the microprocessor temperature is below the threshold. We then use the COP gained in this situation as a constant.

4.2 Power Minimization

We now formulate the power minimization problem of the hybrid-cooled data center. N servers are deployed in the data center and M of them are liquid-cooled. Assuming that the total workload is W_{total}, we minimize the total power consumption as:

$$\min\{P^{server} + P^{air} + P^{liquid}\} \quad (11)$$

Subject to:

$$\sum_{i=1}^{N} W_i = W_{total} \quad (12)$$

$$T^{mp}_i < T^{mp}_{th} \quad 1 \le i \le M \quad (13)$$

$$T^{in}_i < T^{in}_{th} \quad M+1 \le i \le N \quad (14)$$

Equation 12 guarantees that all the workload W_{total} is handled by the servers. Equation 13 enforces that the microprocessors' temperatures of these M liquid-cooled servers are below the required threshold T^{mp}_{th}. Equation 14 enforces that the inlet temperatures of the $(N-M)$ air-cooled servers are below the required threshold T^{in}_{th}.

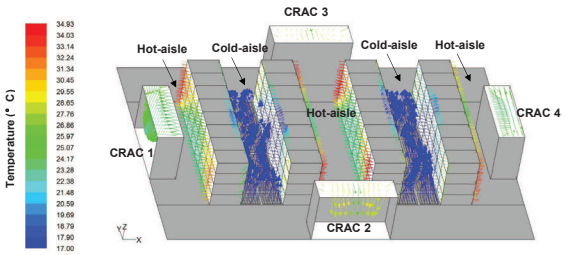


Figure 3: Data center model used in evaluation

4.3 CFD-based Recirculation Matrix and Optimization Solution

We now explain how to get the CFD-based recirculation matrix H. In Equation 6, h_{ij} is an element of the matrix H, indicating the percentage of heat flow recirculated from server i to server j. To simulate the thermal environment of the data center, we use Fluent [5], which is a CFD software package. Figure 3 shows both the layout of the data center model used in this paper and an example of the thermal environment when all the servers are air-cooled. We set the CRAC supply temperature in CFD and use it to get the outlet temperature of each server, in different workload distribution scenarios. After getting the power consumption (P^{air}_i), the outlet temperature of each server (T^i_{out},T^j_{out}) and the CRAC supply temperature (T_{sup}) in all the scenarios, we use them to solve the linear equation shown in Equation 6 and get the recirculation matrix H.

To solve the optimization problems, we use LINGO [15], a comprehensive optimization tool. LINGO employs branch-and-cut methods to break a non-linear programming model down into a list of sub problems to enhance the computation efficiency. It is important to note that our scheme performs offline optimization to determine workload distribution, server on/off and the cooling mode of the data center at different outside temperatures. To dynamically determine those configurations, our scheme can conduct the optimization for different loading levels in an offline fashion and then apply the results online based on the current loading and the current outside temperature.

5 Power Optimization for Geo-Distributed Hybrid-cooled Data Centers

Some big IT companies may have multiple data centers around the world. Although the power minimization for a single hybrid-cooled data center is helpful, it might not be efficient enough for geo-distributed data centers. It is because data centers at different locations have different outside temperatures which lead to different cooling efficiencies. Therefore, it is important to manage the geo-

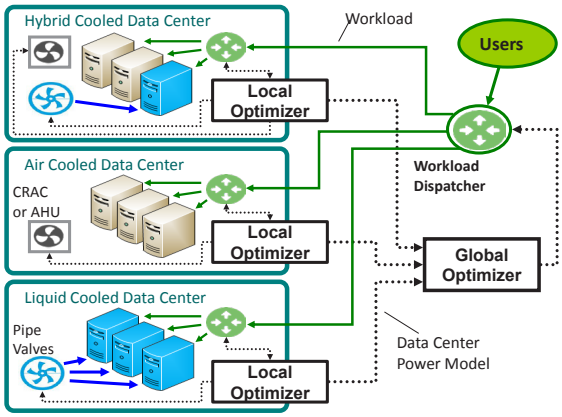


Figure 4: System diagram for geo-distributed data centers. The local optimizer optimizes the cooling configuration and workload distribution locally. The global optimizer optimizes the global workload distribution based on the data center power models.

distributed data centers together. In this section, we extend the total power optimization problem for a single data center to fit geo-distributed data centers. We develop a two-layer light-weight optimization algorithm to lower the computation time.

5.1 Global Optimization

At first, we formulate a global optimization problem for minimizing the total power consumption of geo-distributed data centers, which is very similar to the optimization problem for a single data center. To simplify the notations, we assume that each data center has N servers, which is not a hard requirement for the formulation. We minimize the total power consumption of the data center system that contains K data centers to handle W^{geo}_{total} workload as:

$$\min \sum_{j=1}^{K} P^{DC}_{j} \tag{15}$$

Subject to:

$$\sum_{j=1}^{K}\sum_{i=1}^{N} W_{i,j} = W^{geo}_{total} \tag{16}$$

$$T^{mp}_{i,j} < T^{mp}_{th} \quad 1 \le i \le M \quad 1 \le j \le K \tag{17}$$

$$T^{in}_{i,j} < T^{in}_{th} \quad M+1 \le i \le N \quad 1 \le j \le K \tag{18}$$

P^{DC}_{j} is the total power consumption of data center j. $T^{mp}_{i,j}$ is the microprocessor temperature of liquid-cooled server and $T^{mp}_{i,j}$ represents the inlet temperature of air-cooled server in a data center. $W_{i,j}$ is the workload distributed to server i in data center j. Equation 16 guarantees that all the workload for geo-distributed data centers can be handled. Equation 17 and Equation 18 are the temperature constraints of liquid-cooled and air-cooled servers.

5.2 Two-layer Light-weight Optimization

To solve the global optimization problem for geo-distributed hybrid-cooled data centers, a straightforward solution is to use LINGO directly, as the solution for the single data center power optimization in Section 4. However, as LINGO utilizes the branch-and-bound technique to solve the problem, the computational complexity increases significantly when LINGO solves the problem with geo-distributed data centers. Therefore, we design a two-layer light-weight optimization algorithm to lower the computation complexity.

We first define cPUE for a single data center as

$$cPUE = \frac{P^{server} + P^{cooling}}{P^{compute}(W)} \tag{19}$$

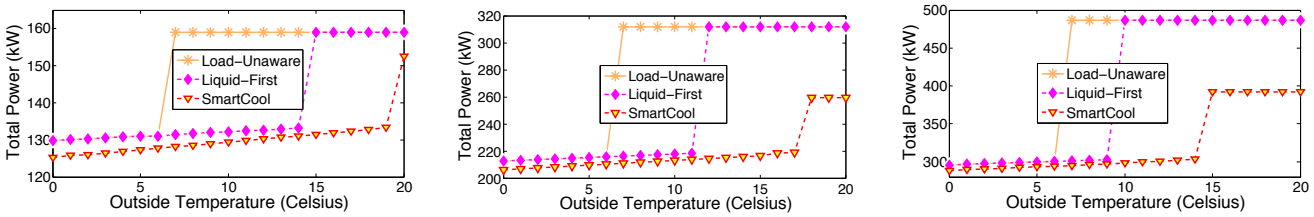
where $P^{compute}$ is the total dynamic computing power consumed by the servers to handle a given workload W. As shown in Figure 4, the local optimizer uses the power optimization process of a single data center (discussed in Section 4) to derive an optimal cPUE for a data center with a given workload W and an outside temperature $T_{outside}$ as $cPUE_{optimal}(T_{outside}, W)$,

$$cPUE_{optimal} = f(T_{outside}, W) \tag{20}$$

In fact, with different amounts of workload and different outside temperatures, $cPUE_{optimal}$ has different values. To get the $cPUE_{optimal}$ model for each data center, we need to obtain a set of sample values of the optimal power consumption with different workloads and outside temperatures. We change the workload from 0% to 100% with a 10% increment step each time, and also change the outside temperature from 0 °C to 20 °C with an increment of 1 °C. We get the power optimization solution of each single data center with different workload and outside temperature combinations. The obtained results are a set of sampled triplets as $(cPUE_{optimal}, W, T_{outside})$. We then use the *Levenberg Marquardt* (LM) algorithm to conduct the linear fitting to find out the $cPUE_{optimal}$ model:

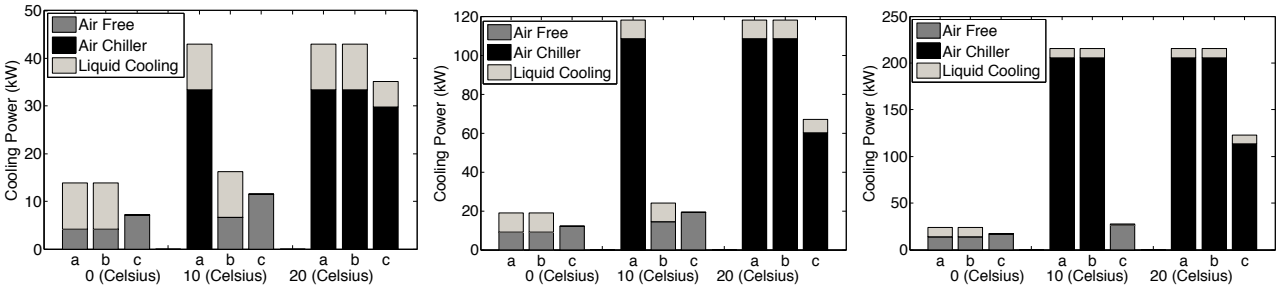
$$cPUE_{optimal} = a * T_{outside} + b * W + c \tag{21}$$

The coefficients *a, b, c* are determined by the data center cooling configuration (e.g., the number of liquid-cooled servers). We choose to do linear model fitting due to the consideration of calculation complexity. Its accuracy is adequate within the acceptable range as we will discuss in Section 6.4. With the $cPUE_{optimal}$ model, we can model the optimal power consumption of a single data center by combining Equations 19 and 21 as:



(a) Total Power at 30% Loading (b) Total Power at 50% Loading (c) Total Power at 70% Loading

Figure 5: Total power consumption with *Load-Unaware*, *Liquid-First* and *SmartCool* at different loadings



(a) Power break at 30% Loading (b) Power break at 50% Loading (c) Power break at 70% Loading

Figure 6: Cooling power breakdown for different schemes (x-axis: a is *Load-Unaware*, b is *Liquid-First*, c is *SmartCool*)

$$P^{DC} = P^{compute}(W) * cPUE_{optimal}(T_{outside}, W) \quad (22)$$

Given a specific moment, the air temperature outside a data center is a constant, and thus P^{DC} only depends on W. As shown in Figure 4, the local optimizer sends the $P^{DC}(W)$ model to the global optimizer, which optimizes the total power consumption by manipulating the workload assigned to each data center. This optimization problem is also solved using LINGO.

Our algorithm successfully decouples the global power optimization problem to a global workload distribution problem and a power optimization problem of each local data center, and thus reduces the optimization overhead significantly.

5.3 Maintaining Response Time

When the workload dispatching is managed by the global optimizer in Figure 4, the request response time needs to be maintained below a threshold. We consider two components of response time: the queuing delay within the data center, and the network delay outside the data center.

A data center can be modeled as a GI/G/m queue [11]. Using the Allen-Cullen approximation for the GI/G/m model, the queuing delay and the number of servers needed to satisfy a given workload demand are related as follows:

$$R = \frac{1}{\mu} + \frac{P_m}{\mu(1-\rho)}\left(\frac{C_A^2 + C_B^2}{2m}\right) \quad (23)$$

where R is the average queuing delay. $\frac{1}{\mu}$ is the average processing time of a request. ρ is the average server utilization. m is the number of servers. $P_m = \rho^{\frac{m+1}{2}}$ for $\rho < 0.7$. $P_m = \frac{\rho^m + \rho}{2}$ for $\rho > 0.7$ and C_A^2 and C_B^2 represent the squared coefficients of the variation of request inter-arrival times and request sizes, respectively. The network delay d_{ij} between the source i and the data center j is taken to be proportional to the geographical distances between them.

When dispatching workload among data centers, we have the constraint that

$$W + d_{ij} < T_{ij} \quad (24)$$

where T_{ij} is the response time threshold of the requests dispatched from source i to data center j.

6 Simulation Results

In this section, we present our evaluation results from the simulation.

6.1 Evaluation Setup

To evaluate different power optimization schemes in a single hybrid-cooled data center, we use a data center model that employs the standard configuration of alternating hot and cold aisles, which is consistent with those used in the previous studies [11]. Figure 3 shows both the data center layout and a thermal environment example when all the servers are air-cooled. The data center consists of four rows of servers, where the first row is composed of liquid-cooled servers. Each row has eight racks, where each rack has 40 servers, adding up to 1,280

servers in the entire data center. The server in our data center model has a 100W idle power consumption and a 300W maximum power consumption when fully utilized. The volumetric flow rate of the intake air of each server is $0.0068 m^3/s$. Each of the four CRAC units in the data center pushes chilled air into a raised floor plenum at a rate of $9000 ft^3/min$ [11]. There also exists a free cooling economizer system that uses outside air when suitable to meet the cooling requirements.

For the liquid-cooled servers we use the rack CDU (coolant distribution unit), in which the CPU of every server is cooled by cold plate and the other components are cooled by the chilled air. We have two chiller systems, of which one is to supply cold water for the cold plates and the other one is to supply the coolant to the CRAC units.

To evaluate different power optimization schemes among different data centers, we consider about three data centers with different cooling configuration, a air-cooled data center (all the four rows of servers are air-cooled), a hybrid-cooled data center (one row of servers are liquid-cooled as discussed in the previous paragraphes) and a liquid-cooled data center (all the servers are liquid-cooled). The only difference between these three data centers are the number of liquid-cooled servers. Other settings of the data centers are the same.

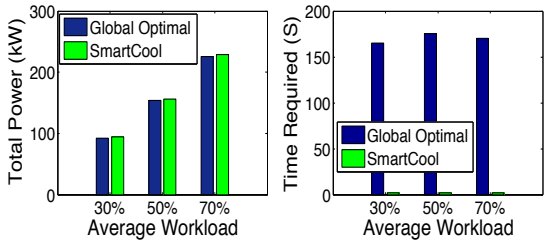
6.2 Comparison of Cooling Schemes

In this section, we compare our *SmartCool* scheme with two baselines: *Load-Unaware* and *Liquid-First*.

Load-Unaware determines the cooling mode by comparing the outside temperature to a fixed temperature threshold, which is equal to the highest CRAC supply temperature that can safely cool the servers when they are all fully utilized. When the outside temperature is below the threshold free air cooling is used, otherwise the traditional air cooling system with chillers and pumps is selected. *Load-Unaware* prefers to distribute the workload to the liquid-cooled servers. If they are fully utilized, the remaining workload is then distributed to the air-cooled servers. The servers in the middle of each row and at the bottom of each rack are prior, as servers located at those places have less recirculation impact and lower inlet temperature [11].

In contrast, *Liquid-First* dynamically adjusts the temperature threshold for free air cooling, based on the real-time workload. It first distributes workload to the servers in the same way as *Load-Unaware*, and then uses the highest CRAC supply temperature that can safely cool the servers as the temperature threshold.

Figure 5 shows the total power consumption of the three different schemes at different loadings with different outside temperature. We can see from the results that



(a) Power Comparison (b) Time Comparison

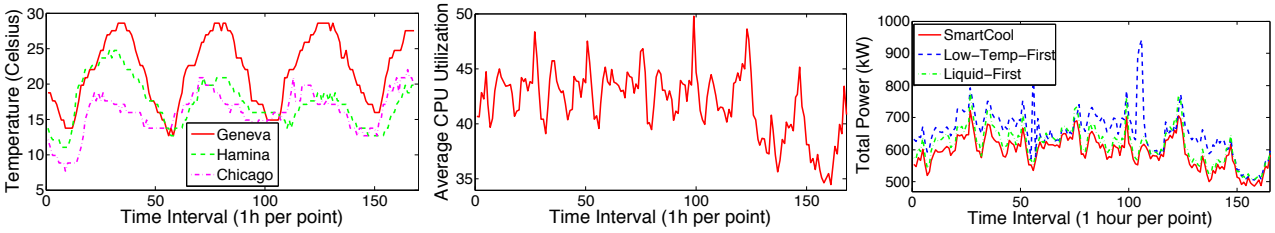
Figure 7: Power consumption and required time comparison between *Global Optimal* and *SmartCool*

all the three cooling schemes achieve a low power consumption when the outside temperature is low, because all of them can use free air cooling. Compared with *Load-Unaware* and *Liquid-First*, *SmartCool* shows the lowest power consumption because it considers the heat recirculation among air-cooled servers when distributing workload.

When the outside temperature increases, we can see that *Load-Unaware* is the first to have a jump in the power consumption curve among the three schemes. This is because *Load-Unaware* uses a fixed temperature threshold to decide whether to use free air cooling or not. The temperature threshold of *Load-Unaware* is determined with the data center running a 100% percent workload. Therefore, it is unnecessarily low for less workload such as 30%. *Liquid-First* is the second one to have a power consumption jump due to switching from free air cooling to CRAC cooling. It can use free air cooling more than *Load-Unaware* when the outside temperature is higher, because its temperature threshold is determined based on the real-time workload (e.g., 30%, 50% or 70%) rather than the maximum workload (100%). Hence *Liquid-First* saves power compared with *Load-Unaware*. *SmartCool* scheme is the last one to have the power consumption jump, because *SmartCool* optimizes the workload distribution among liquid-cooled and air-cooled servers, while the two baselines concentrate the workload on a small number of air-cooled servers and result in some hot spots when air cooling is necessary. Those hot spots require a lower temperature of the supplied air for cooling and thus increase the power consumption. Therefore, *SmartCool* is the most power efficient scheme.

6.3 Power Breakdown of Cooling Schemes

In Figure 6, we break the cooling power consumption of the three schemes (including *Load-Unaware*,*Liquid-First* and *SmartCool*) into *Air Free* (free cooling power consumption), *Air Chiller* (traditional air cooling power consumption) and *Liquid Cooling* (liquid cooling power



(a) Temperature Trace (b) IBM trace file (c) Total Power Consumption Comparison

Figure 8: Total Power Consumption comparison of different workload dispatching schemes

consumption). We choose three outside temperature points, 0 °C, 10 °C and 20 °C to discuss the impact of outside temperature.

Figure 6 (a) shows the cooling power under different outside temperatures at 30% loading. We can see that when the outside temperature is 0 (Celsius), *Load-Unaware* and *Liquid-First* consume the same amount of cooling power. *SmartCool* consumes less cooling power than the two baselines because it distributes all the workload to the air-cooled servers, which are cooled by the more efficient free air, while the baselines prefer to use liquid-cooled servers.

When the outside temperature raises to 10 °C, *Load-Unaware* switches to the traditional air cooling mode, which starts to use the chiller system and leads to the increase of the cooling power consumption. Differently, the other two schemes show similar results as those at 0 °C. When the outside temperature is 20 °C, all the three schemes switch to traditional air cooling mode. However, *SmartCool* still consumes less cooling power because it considers the impact of air recirculation and optimizes the workload distribution.

Figures 6 (b) and 6 (c) show the breakdown of cooling power consumption when the data center is at 50% and 70% loading. They show the same trends as 6 (a) though the total cooling power increases due to the increase of the workload.

6.4 Comparison between SmartCool and Global Optimal Solution

In this section, we evaluate our two-layer power optimization algorithm in the geo-distributed data center settings and compare it with the global optimization scheme in terms of optimization performance, including the optimized total power consumption and the time overhead. The global optimal scheme solves the geo-distributed power optimization problem as a whole including deciding the global and local workload distribution as well as the cooling mode management of each data center. *SmartCool* uses a two-layer optimization algorithm as discussed in Section 5. Due to the long computation time of the global optimization scheme, we use three smaller scale data centers in this set of experiments. Each data center has two rows of racks. Each row contains 4 racks and each rack contains four blocks. There exists 10 servers in each block.

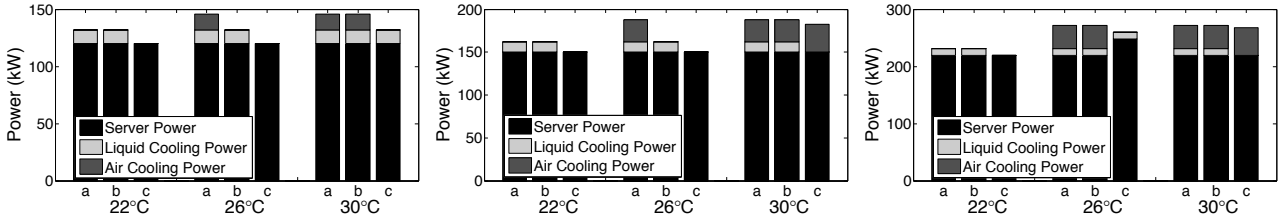
Figure 7(a) shows the total power consumption of the two schemes at different loadings. We can see that *SmartCool* has very close optimization result to the global optimal solution. The performance difference is due to the model fitting error introduced from the cPUE modeling process as discussed in Section 5. However, our *SmartCool* consumes much less time than the *Global Optimal* solution according to Figure 7(b).

6.5 Results with Real Workload and Temperature Traces

We now evaluate different power management schemes on three geo-distributed data centers with real workload and temperature traces. Each data center contains 1,280 servers. To show the diversity of different data centers, we configure them with different cooling system, which are air cooling, liquid cooling and hybrid cooling, respectively. The outside temperatures are shown in Figure 8 (a) which are one week temperature traces of three different locations, Geneva, Hamina and Chicago [25].

We compare the total power consumption under three workload dispatching schemes: *Liquid-First*, *Low-Temp-First* and *SmartCool*. *Liquid-First* dispatches workload to the three data centers according to their cooling efficiencies, which are ranked from high to low as the liquid-cooled data center, the hybrid-cooled data center and the air- cooled data center. Thus workload is first dispatched to the liquid-cooled data center and if it can not handle all the workload, the rest part is dispatched to the hybrid-cooled data center and then the air-cooled data center. For *Low-Temp-First*, workload is first distributed to the data center with the lowest outside temperature since it has the highest possibility to use free cooling system which will consume the least cooling power. For *SmartCool*, workload is distributed according to the approach discussed in Section 5.

Figure 8 (b) shows a one week trace of the average CPU utilization from an IBM production data center [29]. We use this trace to generate the total workload in our experiment. Figure 8 (c) shows the power consumption of the three different schemes. We can see that our *SmartCool* consumes the least power because it considers the impacts of both the outside tempera-



(a) Total power at 30% Loading (b) Total Power at 50% Loading (c) Total Power at 70% Loading

Figure 9: Hardware results under different ambient temperatures at different loadings (x-axis: a is *Load-Unaware*, b is *Liquid-First*, c is *SmartCool*)

ture and the workload on the data center PUE. *Liquid First* consumes more power than *SmartCool* but less than the *Low-Temp-First* solution, because it first concentrates workload on liquid-cooled data center, which has relatively high cooling efficiency. *Low-Temp-First* consumes the most power among the three schemes because when the outside temperature is not low enough or the workload is relatively high, *Low-Temp-First* first dispatches all the workload to the data center with the lowest outside temperature and then traditional air cooling system with chiller and pump must be used to cool down the servers, which will cause high cooling power.

7 Hardware Experiment

In addition to the simulations, we also conduct experiments on our hardware testbed to evaluate *SmartCool* by comparing it with the baselines, i.e., *Load-Unaware* and *Liquid-First*. The testbed includes one liquid-cooled server and three air-cooled servers. A heater is used to set the ambient temperature to be 22°C, 26°C and 30°C.

To compare the total power consumptions of the three schemes, we use power meters to measure the power consumed by the servers and the cold plate used for liquid cooling. For the reason that we do not have an air handler and just use the ambient air to take away heat generated by the server, we assume that the air cooling power is zero under free cooling mode and use Equation 4 and 5 to estimate the power consumption of traditional air cooling.

Figure 9 shows the power consumption of different schemes with different ambient temperatures and loadings. We can see that when the ambient temperature is 22°C, the air cooling power of all the three schemes are zero at different loadings, because they can all adopt free air cooling. *SmartCool* consumes less cooling power than *Load-Unaware* and *Liquid-First*, for it does not consume liquid cooling power when the ambient temperature is at 22°C as all the workload is distributed to the air-cooled servers. In contrast, the two baselines prefer to distribute workload to the liquid-cooled servers, no matter how cold the ambient air is. When the ambient temperature is 22°C and the workload is at 30% or 50%, *Load-Unaware* consumes the most cooling power, because it begins to use traditional air cooling since the ambient temperature exceeds its fixed temperature threshold for cooling mode decision, and thus cause more cooling power. *Liquid-First* still uses free cooling at 30% and 50% loadings, and begins to use traditional air cooling when the workload is 70%. At 26°C *SmartCool* still consumes less cooling power than the other two schemes. The results show the same trend when the outside temperature is 30°C.

8 Conclusion

In this paper, we have presented SmartCool, a power optimization scheme that effectively coordinates different cooling techniques and dynamically manages workload allocation for jointly optimized cooling and server power. In sharp contrast to the existing work that addresses different cooling techniques in an isolated manner, SmartCool systematically formulates the integration of different cooling systems as a constrained optimization problem. Furthermore, since geo-distributed data centers have different ambient temperatures, SmartCool dynamically dispatches the incoming requests among a network of data centers with heterogeneous cooling systems to best leverage the high efficiency of free cooling. A light-weight heuristic algorithm is proposed to achieve a near-optimal solution with a low time overhead. SmartCool has been evaluated both in simulation and on a hardware testbed with real-world workload and temperature traces. The results show that SmartCool outperforms two state-of-the-art baselines by having a 38% more power savings.

References

[1] A.COSKUN, J.L.AYALA, D.ATIENZA, AND T.S.ROSING. Modeling and dynamic management of 3d multicore systems with liquid cooling. In *Proceedings of International Conference on Very Large Scale Integration (VLSI-SoC)* (2009).

[2] A.GREENBERG, J.HAMILTON, A.MALTZ, AND P.PATEL. The cost of a cloud: research problems in data center networks. In *ACM SIGCOMM CCR* (2009).

[3] B.GEBREHIWOT, K.AURANGABADKAR, N.KANNAN, AND D.AGONAFER. CFD analysis of free cooling of modular data centers. In *Proceedings of Semiconductor Thermal Measurement and Management Symposium (SEMI-THERM)* (2012).

[4] CERN . Data Centre. http://home.web.cern.ch/about/computing.

[5] COMPUTATIONAL FLUID DYNAMICS (CFD) SOFTWARE BY ANSYS INC. . Fluent. http://www.caeai.com/cfd-software.php.

[6] D.CHEMICOFF. The uptime institute 2012 data center survey. http://symposium.uptimeinstitute.com.

[7] D.C.SUJATHA, AND S.ABIMANNAN. Energy efficient free cooling system for data centers. In *Proceedings of IEEE Third International Conference on Could Computing Technology and Science (CloudCom)* (2011).

[8] D.HWANG, V.P.MANNO, M.HODES, AND G.J.CHAN. Energy savings achievable through liquid cooling: A rack level case study. In *Proceedings of Intersociety Conference on Thermal and Thermomechanical Phenomena in Electronic Systems (ITherm)* (2010).

[9] D.SUJATHA, AND S.ABIMANNAN. Energy efficient free cooling system for data centers. In *Proceedings of International Conference on Cloud Computing Technology and Science (CloudCom)* (2011).

[10] EMERSON. Liebert DSE precision cooling system sales brochure . http://www.emersonnetworkpower.com.

[11] F.AHMAD, AND T.N.VIJAYKUMAR. Joint optimization of idle and cooling power in data centers while maintaining response time. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2010).

[12] I.GOIRI, W.KATSAK, K.LE, T.NGUYEN, AND R.BIANCHINI. Parasol and greenswitch: Managing datacenters powered by renewable energy. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2013).

[13] K.ZHENG, X.WANG, L.LI, AND X.WANG. Joint power optimization of data center network and servers with correlation analysis. In *Proceedings of the 33rd IEEE International Conference on Computer Communications (INFOCOM)* (2014).

[14] L.A.BARROSO, J.CLIDARAS, AND U.HOLZLE. The datacenter as a computer: An introduction to the design of warehouse-scale machines. In *Morgan and Claypool Publishers* (2009).

[15] LINDO SYSTEMS INC. LINDO software products. http://www.lindo.com.

[16] M.A.ADNAN, R.SUGIHARA, AND R.GUPTA. Energy efficient geographical load balancing via dynamic deferral of workload. In *Proceedings of 5th IEEE conference on cloud computing (CLOUD)* (2012).

[17] M.BROCANELLI, S.LI, X.WANG, AND W.ZHANG. Joint management of data centers and electric vehicles for maximized regulation profits. In *Proceedings of the fourth IEEE International Greeen Computing Conference (IGCC)* (2013).

[18] M.IYENGAR, M.DAVID, P.PARIDA, AND V.KAMATH. Server liquid cooling with chiller-less data center design to enable significant energy savings. In *Proceedings of Semiconductor Thermal Measurement and Management Symposium (SEMI-THERM)* (2012).

[19] M.IYENGAR, M.DAVID, P.PARIDA, AND V.KAMATH. Server liquid cooling with chiller-less data center design to enable significant energy savings. In *Proceedings of Semiconductor Thermal Measurement and Management Symposium (SEMI-THERM)* (2012).

[20] Q.TANG, S.K.S.GUPTA, AND G.VARSAMOPOULOS. Thermal-aware task scheduling for data centers through minimizing heat recirculation. In *Proceedings of IEEE International Conference on Cluster Computing* (2007).

[21] Q.TANG, T.MUKHERJEE, S.GUPTA, AND P.CAYTON. Sensor-based fast thermal evaluation model for energy efficient high-performance data centers. In *Proceedings of International Conference on Intelligent Systems and Image Processing (ICISIP)* (2006).

[22] R.ANTON, H.JONSSON, AND B.PALM. Modeling of air conditioning systems for cooling of data centers. In *Proceedings of Intersociety Conference on Thermal and Thermomechanical Phenomena in Electronic Systems (ITHERM)* (2002).

[23] R.ZHOU, Z.WANG, C.E.BASH, AND A.MCREYNOLDS. Data center cooling management and analysis-a model based approach. In *Proceedings of Semiconductor Thermal Measurement and Management Symposium (SEMI-THERM)* (2012).

[24] S.MOVOTNY. Data center rack level cooling utilizing water-cooled, passive rear door heat exchangers as a cost effective alternative to crash air cooling.

[25] TIMEANDDATE. Weather around the World. www.timeanddate.com.

[26] TREEHUGGER INC. . Data Center Cooling Energy Reduction Thanks to Fluid Submerged Servers. http://www.treehugger.com.

[27] W.HUANG, M.ALLEN-WARE, J.B.CARTER, AND E.ELNOZAHY. TAPO: Thermal-aware power optimization techniques for servers and data centers. In *Proceedings of IEEE International Greeen Computing Conference (IGCC)* (2011).

[28] W.ZHENG, K.MA, AND X.WANG. Exploiting thermal energy storage to reduce data center capital and operating expenses. In *Proceedings of the 20th International Symposium on High Performance Computer Architecture (HPCA)* (2014).

[29] X.WANG, M.CHEN, C.LEFURGY, AND T.W.KELLER. SHIP: Scalable hierarchical power control for large-scale data centers. In *Proceedings of International Conference on Parallel and Distributed Systems (PACT)* (2009).

[30] Y.CHEN, D.GMACH, C.HYSER, Z.WANG, C.BASH, C.HOOVER, AND S.SINGHAL. Integrated management of application performance, power and cooling in data centers. In *Proceedings of Network Operations and Management Symposium (NOMS)* (2010).

[31] Y.ZHANG, Y.WANG, AND X.WANG. Electricity bill capping for cloud-scale data centers that impact the power markets. In *Proceedings of the International Conference on Parallel Processing* (2012).

[32] Y.ZHANG, Y.WANG, AND X.WANG. TEStore: Exploiting thermal and energy storage to cut the electricity bill for datacenter cooling. In *Proceedings of the 8th International Conference on Network and Service Management (CNSM)* (2012).

[33] Z.LIU, M.LIN, AND L.ANDREW. Greening geographical load balancing. In *Proceedings of the ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems* (2011).

[34] Z.LIU, Y.CHEN, C.BASH, A.WIERMAN, D.GMACH, Z.WANG, M.MARWAH, AND C.HYSER. Renewable and cooling aware workload management for sustainable data centers. In *Proceedings of the ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems* (2012).

[35] Z.WANG, C.BASH, C.HOOVER, AND A.MCREYNOLDS. Integrated management of cooling resources in air-cooled data centers. In *Proceedings of IEEE Conference on Automation Science and Engineering (CASE)* (2010).

Managing Green Datacenters Powered by Hybrid Renewable Energy Systems

Chao Li[1], Rui Wang[2], Tao Li[1], Depei Qian[2], and Jingling Yuan[3]
[1]*University of Florida* [2]*Beihang University* [3]*Wuhan University of Technology*

Abstract

The rapidly growing server energy expenditure and the warning of climate change have forced the IT industry to look at datacenters powered by renewable energy. Existing proposals on this issue yield sub-optimal performance as they typically assume certain specific type of renewable energy sources and overlook the benefits of cross-source coordination. This paper takes the first step toward exploring green datacenters powered by hybrid renewable energy systems that include baseload power supply, intermittent power supply, and backup energy storage. We propose *GreenWorks*, a power management framework for green high-performance computing datacenters powered by renewable energy mix. Specifically, GreenWorks features a hierarchical power coordination scheme tailored to the timing and capacity of different renewable energy sources. Using real datacenter workload traces and renewable power generation data, we show that our scheme allows green datacenters to achieve better design trade-offs.

1. Introduction

The global server power demand reaches approximately 30 gigawatts in total [1], which account for over 250 million metric tons of CO_2 emissions per year [2]. Faced with a growing concern about the projected rise in both server power demand and carbon emissions, academia and industry alike are now focusing more attention than ever on non-conventional power provisioning solutions. For instance, recently there have been vigorous discussions on renewable energy driven computer system design with respect to carbon-aware scheduling [3-5], renewable power control [6-9], and cost optimization strategies [10-11]. In addition, Microsoft, eBay, HP, and Apple have announced projects that use green energy sources like solar/wind power, fuel cells, and bio-gas turbines to minimize their reliance on conventional utility power [12-15]. It has been estimated that these eco-friendly IT solutions could reduce almost 15% global CO_2 emissions by 2020, leading to around $900 billion of cost savings [16].

The expected growth in renewable power generation poses new challenges for datacenter operational resilience. A number of the renewable energy sources are *intermittent power supply*, such as wind turbine and solar array. They are free sources of energy but incur power variability problems. Several emerging green power supplies, such as fuel cells and bio-fuel based generators, are typically used as *baseload power supply*. They are stable and controllable power sources, but not fast enough to respond instantaneously to quick changes in server power demand. In case the intermittent power supply drops suddenly or the baseload power supply cannot follow an unexpected power demand surge, *backup power supply* (e.g., batteries, super-capacitors) must be used to handle the power shortfall.

As we move toward a smarter grid, datacenters are expected to be powered by hybrid renewable energy systems that combine multiple power generation mechanisms [17]. With an integrated mix of complementary power provisioning methods, one can overcome the limitations of each single type of power supply, thereby achieving better energy reliability and efficiency.

However, a common limitation of prior proposals is that they mainly focus on certain specific type of green power supplies. We classify existing schemes into three broad categories: 1) *load shedding*, which focuses on utilizing intermittent power [6, 9], typically reduces load when renewable power drops; 2) *load boosting*, which uses both intermittent and backup power [8, 18], takes advantage of the stored energy to maintain desired performance when the current green power generation is inadequate; and 3) *load following*, which assumes both baseload and backup power [19], leverages tunable generators to track datacenter load demand. Since prior proposals lack the capability of managing renewable energy mix, they can hardly gain the maximum benefits from hybrid renewable energy systems, and consequently yield sub-optimal design tradeoffs.

In this study we explore diversified multi-source power provisioning for green high-performance datacenters today and in the future. We propose GreenWorks, a framework for managing datacenter power across several layers from datacenter server to onsite renewable energy mix. GreenWorks comprises two key elements: the *green workers*, which are multiple platform-specific power optimization modules that use different supply/load control strategies for different types of renewable energy systems; and *green manager*, a hierarchical coordination scheme for green workers.

GreenWorks tackles the challenges of integrating and coordinating heterogeneous power supplies with a three-tiered hierarchical coordination scheme. Each layer of the hierarchy is tailored to the specific timing and utilization requirements of the associated energy sources. In addition, power management modules in different layers of the hierarchy can also interact with

each other within the framework. This allows us to further improve the power management effectiveness of hybrid renewable energy systems.

GreenWorks emphasize a multi-objective power management. It jointly manages green energy utilization, backup energy availability, and workload performance. Specifically, we define three types of green workers: 1) *baseload laborer*, which adjusts the output of the baseload power to track the coarse-grained changes in load power demand; 2) *energy keeper*, which regulates the use of the stored renewable energy to achieve satisfactory workload performance while maintaining desired battery life; and 3) *load broker*, which could opportunistically increase the server processing speed to take advantage of the excess energy generation of the intermittent power supply. All the three modules are able to distill crucial runtime power profiling data and identify appropriate control strategies for different types of renewable generation.

To our knowledge, this paper is the first to design a hierarchal power management and coordination framework for multi-source powered green datacenters.

This paper makes three main contributions:

- We propose GreenWorks, a hierarchical power management framework for green datacenters powered by renewable energy mix. It enables cross-source power management coordination, thereby greatly facilitating supply-load power matching.
- We propose a multi-source driven multi-objective power management that takes advantage of our hierarchical power management framework. Our technique enables GreenWorks to maximize the benefits of the hybrid renewable energy systems without heavily relying on any single type of power supply.
- We evaluate GreenWorks using real-world workload traces and green energy data. We show that GreenWorks could achieve less than 3% job runtime increase, extend battery lifetime by 23%, increase UPS backup time by 12%, and maintains the same energy efficiency as the state-of-the-art design.

The rest of this paper is organized as follows. Section 2 introduces background. Section 3 proposes the GreenWorks framework. Section 4 proposes multi-objective power management scheme. Section 5 describes evaluation methodologies. Section 6 presents our results. Section 7 discusses related work and Section 8 concludes this paper.

2. Background

Today's energy crisis and environmental problems force the IT industry to look at datacenter power provisioning in a different way. In this section, we introduce green datacenters powered by hybrid renewable energy systems and discuss their design challenges.

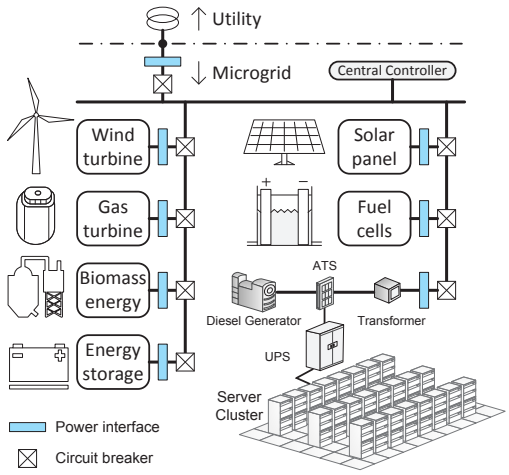


Figure 1: A datacenter powered by renewable power mix.

2.1 Hybrid Renewable Energy Systems

There are three types of renewable power supplies that we can leverage to power a datacenter. Some green power supplies, such as solar panels and wind turbines, are affected by the availability of ambient natural resources (i.e., solar irradiance or wind speed). They are referred to as *intermittent power supply* since their outputs are time-varying. Several emerging green power supplies – including fuel cells, bio-fuel based gas turbines, and bio-mass generators – can offer controllable green energy by burning various green fuels. We refer to them as *baseload power supply* since they can be used to provide stable renewable power to meet the basic datacenter power demand (e.g., idle power). In addition, energy storage devices such as batteries and super-capacitors are also critical components that provide *backup power supply*. They can be used to temporarily store green energy or improve power quality.

Looking ahead, datacenters in the smart grid era are expected to be powered by *hybrid renewable energy systems* that combine all the three types of power supplies, as shown in Figure 1. Different power supplies are typically implemented as small, modular electric generators (called micro-sources) near the point of use. To manage such an integrated renewable energy mix, micro-grid is proposed as a coordinated cluster/network of supply and load [20]. Although the micro-grid allows its customer to import power from the utility, we focus our attention on minimizing the reliance on utility power due to sustainability and cost concerns.

Energy source management and datacenter load management are largely decoupled in prior studies. Existing micro-grid control strategies often focus on power supply scheduling [21]. Recent proposals on power-aware datacenter mainly emphasize demand response control [22, 23]. In contrast, we propose load/supply cooperative power management across several layers from servers to hybrid renewable energy systems.

Micro-sources	Response Speed	Startup Time
Batteries	Immediate	N/A
Flywheel	Immediate	N/A
Fuel cells	30 sec ~ 5 min	20 ~ 50 min
Gas turbine	10s of seconds	2 ~ 10 min

Table 1: Response speed of different power supplies [24-28].

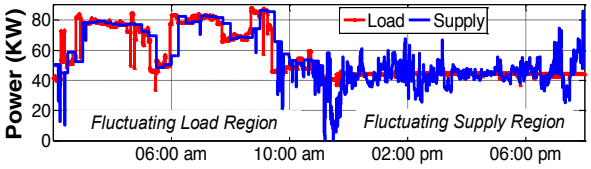


Figure 2: The demand-supply mismatch scenario.

2.3 Energy Balance Challenge

Many system-level events can cause power demand fluctuations, such as dynamic power tuning via DVFS, on/off server power cycles, and random user request. Unexpected variations in intermittent power supplies, unfavorably combined with datacenter workload fluctuation, could make the power mismatch problem even worse. Therefore, matching datacenter load to the variable power budget is often the crux of eliminating power disruptions in a green datacenter.

Managing multi-source powered system can be a great undertaking. As shown in Table 1, micro-sources often have different characteristics and operating timeframes. Most baseload green power systems cannot meet the needs of fast supply-load power matching. For example, both fuel cells and gas turbines need time to be committed and dispatched to a desired output level. They provide a slow energy balance service called *load following*, which typically occurs every tens of minutes to a few hours [28].

Figure 2 illustrates the load matching effectiveness using real-world datacenter traces and renewable energy datasets. The power supply trace shown in the figure combines the outputs of baseload power units and intermittent wind power supplies. As can be seen, load following alone cannot eliminate fine-grained power mismatch. When the wind power is stable, fluctuating load can be the main cause of power mismatch; when wind power output varies, it can significantly increase mismatch events. Although increasing the baseload power output can reduce the chance of brownout, it will significantly increase the operational expenditure.

Note that we cannot heavily rely on utility power grid and energy backup to manage the demand-supply power mismatches. First, it requires additional standby power capacity, which is economically unfavorable. Energy backup services are typically much more costly than the load following services [29]. Second, grid-inverter and battery incur round-trip energy loss, which degrade overall system efficiency. Third, heavy reliance on backup power supply can be risky. As recent survey indicates, datacenters in the US experiences 3.5 times of utility power loss per year with an average duration over 1.5 hours [30]. It also shows that UPS battery failure and capacity exceeded are the top root causes of unplanned outages. Without appropriate coordination, the demand-supply power mismatch can cause frequent battery discharging activities, which not only decrease the battery lifetime but also frequently deplete the stored energy that is crucial for handling emergencies.

In this study we explore a holistic approach for eliminating the supply-load mismatch problems in green datacenters. Specifically, we look at how cross-source power management and coordination will help to improve energy balance and datacenter resilience.

3. The GreenWorks Framework

GreenWorks is a hierarchical power management scheme that is tailored to the specific timing and utilization requirements of different energy sources. It provides coordinated power management across intermittent renewable power supplies, controllable baseload generators, onsite batteries, and datacenter servers.

The intention of this work is to provide an initial power management framework for datacenters powered by renewable energy mix. In the smart grid era, datacenters must increase their awareness of the attributes of power supplies to achieve the best design trade-offs.

3.1 System Overview

Figure 3 depicts the architecture of a green datacenter powered by renewable energy mix. We adopt typical micro-grid power distribution scheme for managing various renewable energy resources. Various renewable energy systems are connected to the power feeder through circuit breakers and appropriate interfaces.

GreenWorks is a middleware that resides between front-end computing facilities and back-end distributed generators. It manages various onsite energy sources through a micro-grid central controller, which is a typical power management module in the micro-grid system. The controller is able to adjust onsite power generation through communication with the dedicated power interface connected to each distributed generators. GreenWorks also communicates with the UPS battery rack, the cluster-lever power meters, and the server-level power control module. It cooperatively adjusts power supplies and workload performance levels, and thereby eliminates demand-supply power mismatch.

As shown in Figure 3, GreenWorks comprises two key elements: the *green workers* and the *green manager*. The former are platform-specific power management modules for managing different types of micro-sources and the later coordinates these modules. In this study we define three types of green workers: *baseload laborer* (B), *energy keeper* (E), and the *load broker* (L).

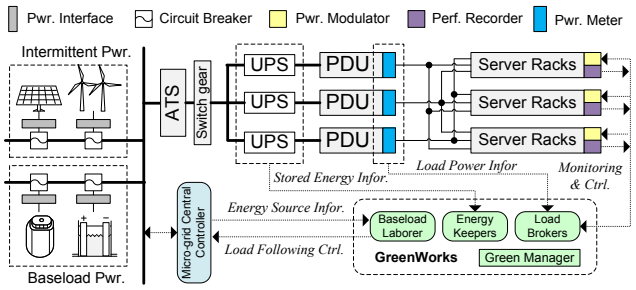


Figure 3: High-level system architecture for GreenWorks.

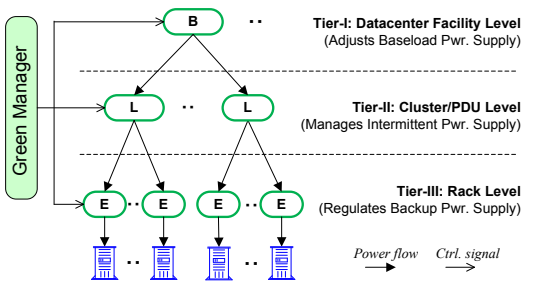


Figure 4: GreenWorks power management hierarchy.

The *baseload laborer* controls the output of distributed generators such as fuel cells and bio-fuel generators. It is responsible for providing a specific amount of baseload power to satisfy the basic power needs (i.e., datacenter idle power). It can also provide load following services [28] at each coarse-grained time interval.

The *energy keeper* is able to provide necessary power support if intermittent power supply drops suddenly or load surge happens. It also monitors the capacity utilization and the health status of the battery packs. In Figure 3, we use distributed battery architecture (at server cluster level) since it has better energy efficiency, reliability, and scalability [31].

The *load broker* is responsible for managing the fine-grained power mismatch between the fluctuating datacenter load and the intermittent power supply. We leverage the performance scaling capability (via CPU frequency scaling) of server system to match load power demand to time-varying green energy budget.

3.2 Power Management Hierarchy

Although the hybrid renewable energy systems are often centrally installed at the datacenter facility level, improving the overall efficiency requires a multi-level, cooperative power management strategy.

GreenWorks uses a three-tier control hierarchy for power management coordination. It organizes different types of green workers in the power management hierarchy based on their design goals.

As Figure 4 shows, in the top tier of the hierarchy is the baseload laborer. We put the load laborer at the datacenter facility level since it is where the baseload power generator is integrated. Managing baseload power budget at datacenter level facilitates load following control, thereby minimizing over-/under- generation of the baseload renewable energy.

GreenWorks manages the intermittent renewable power supply at the cluster level, or PDU (power distribution unit) level. At this level, dynamic voltage and frequency scaling (DVFS) shows impressive peak power management capabilities [32] and could be leveraged to manage the supply-load power mismatch. During runtime, the load broker calculates the total renewable power generation based on the baseload power budget and the assigned renewable power. When the total renewable power generation is not enough, the load broker will decrease server processing speed evenly or request stored energy (from the energy keeper), depending on whichever yields the best design tradeoff.

The energy keeper resides in the third tier of the hierarchy. This allows us to provide backup power directly to server racks if local demand surge happens or power budget drops. Such distributed battery architecture [31] has many advantages such as high efficiency and reliability. In this study, we leverage it for managing fine-grained supply-load power mismatches.

The main advantage of our multi-level cooperative power management scheme is that it facilitates cross-source power optimization. For example, GreenWorks allows datacenters to schedule additional baseload generation to release the burden of the energy backup when the capacity utilization of onsite batteries is high. It also allows them to request additional stored renewable energy to boost server performance if necessary.

4. Multi-Objective Power Management

In this section, we propose multi-source driven multi-objective power management for GreenWorks. The basic idea is to take advantages of the cross-source coordination capability of GreenWorks to balance the usage of different types of energy sources. To achieve this goal, we develop a novel three-stage coordination scheme that synergistically combines battery-aware power management, workload-aware power management, and variability-aware power management to achieve the best design trade-offs.

4.1 Stage I: Adequate Power Supply Budget

The green manager enters power management Stage-I (as shown in Figure 5), when the renewable power generation is unable to ensure the rated speed on all the active servers. In this stage, the excess renewable energy generated will be stored in UPS batteries if there is still enough room. In addition, the green manager also monitors the actual charging current and the maximal power capacity of batteries. The remaining excess renewable power will be send to the utility grid via grid-tie inverter, which is a power inverter that synchronizes onsite power generation with a utility line.

Requires: ***The percentage of job execution time increase: T TimeTable[T][index], a N×2 lookup table for N jobs***
Initialize: ***TimeTable is sorted based on T (descending order)***

```
1:  PowerHeadroom= TotalSupply – PeakLoadDemand;
2:  for each job j in the TimeTable
3:      if job j has enough thermal headroom then
4:          while (The frequency of j < maxFreq)
5:              Increase the node frequency for job j;
6:              Re-evaluate PowerHeadroom
7:              if PowerHeadroom = 0 then break;
```

Figure 5: Load adaptation pseudo code for Stage-I.

During runtime, the load broker dynamically monitors each job's progress and calculates an *execution time increase* (ETI). Assuming that a job *j* has *n* execution phases: {*1, 2, 3, ···, i, ···, n*}. For a given execution phase *i* that spends t_a seconds under actual processing frequency f_{actual}, it would spend t_r seconds under rated processing frequency f_{rated}. If we scale down the frequency (i.e., $f_{actual} \le f_{rated}$), we expect to increase the execution time (i.e., $t_a \ge t_r$). As frequency scaling mainly changes CPU time and has little impact on non-CPU time (i.e., I/O waiting time and memory access time), the job's ETI in phase *i* is given by:

$$T_{ij} = t_a - t_r = \mu(1 - \frac{f_{actual}}{f_{rated}})t_a, \quad \mu = \frac{CPUtime}{Runtime}, \qquad (1)$$

where μ is the monitored actual CPU utilization (under scaled processing frequency) in execution phase *i*. ideally, without performance scaling, the total execution time E_r of previous *i* execution phases is:

$$E_r = \sum t_r = E_a - \sum_i T_{ij}, \qquad (2)$$

where E_a is the actual total execution time of previous *i* execution phases monitored by load brokers. Thus, we can compute the percentage increase of execution time at the end of execution phase *i* as:

$$T_{\%(i,j)} = \left(\sum_i T_{ij}\right) \Big/ E_r \qquad (2)$$

In Figure 5, the green manager dynamically updates the job execution time information and maintains a sorted lookup table for each running job. When allocating additional renewable power budget across server nodes, the green manager will always give priority to jobs that have higher job execution time increase. Specifically, our green manager uses a job acceleration scheme which opportunistically boosts the processing speed/frequency (i.e., over-clocking) to take advantage the additional renewable power budget. This can help mitigate unnecessary energy loss due to power feedback and improve workload performance. It allows a processor to enter a performance state higher than the specified frequency if there is enough thermal/power headroom and if it is enabled by the power management software. Through execution time monitoring and power allocation balancing in the Stage I, we can greatly improve average workload performance.

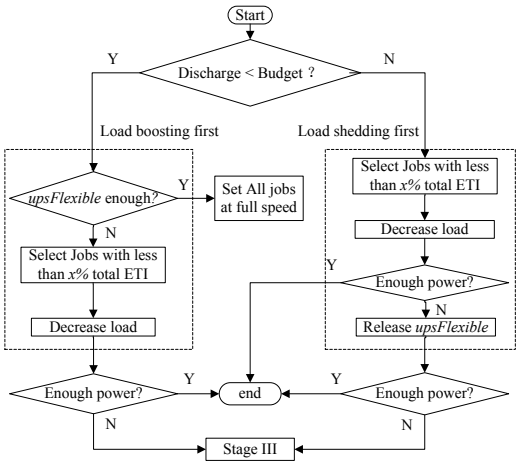


Figure 6: The power management pseudo code for Stage-II.

4.2 Stage II: Moderate Power Supply Drop

Our system enters Stage-II when it senses inadequate power supply. Unlike prior designs which heavily rely on either load shedding or backup power, we use a balanced power management, as shown in Figure 6.

Battery Discharge Control:

Battery lifetime is an important design consideration. To maximize the benefits of the stored energy without compromising reliability, we dynamically monitor the discharge events of the UPS system and calculate a *discharge budget* based on the aggregated discharge throughput (Amp-hours) of the batteries, the overall runtime of the battery, and the rated cycle life.

We use an Ah-Throughput Model [33] to evaluate the battery cycle life and a kinetic battery model (KiBaM) [34] to analyze the battery charging/discharging behaviors. The Ah-Throughput model states that there is a fixed amount of charges that can be cycled through a battery before it requires replacement. The KiBaM model uses a chemical kinetics process as its basis and describes the charge movement inside the battery, as shown in Figure 7. Both models provide reasonable evaluations of battery systems and have been used in professional power system simulation software developed by the National Renewable Energy Lab [35].

We use two different power control schemes in this stage. If the required UPS energy is within the *discharge budget*, the green manager will give priority to using stored energy to maintain high performance (load boosting). Otherwise, it will first decrease the server speed (load shedding) and then use stored energy if necessary. In Figure 7, we assume a maximal UPS discharge amount of 40% of the total installed capacity, which we refer to as *flexible UPS energy* (*upsFlexible*, 0~40% of the total capacity). We also define a *reserved UPS energy* (40%~80% of the total capacity), which is used to handle significant power drop in the Stage-III.

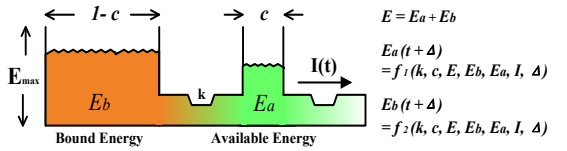


Figure 7: The KiBaM battery model [34]. The stored charge is distributed over two pools: An available-energy pool supplies current directly to the load.

Load Shedding Control:

The load brokers of GreenWorks use performance statistics to make load shedding decisions. GreenWorks allows the datacenter operator to specify a limit (not a hard limit) on job ETI to achieve different performance goals. Our system allows performance scaling only on jobs that have less than *x*% (default value is 10%) increase of execution time. We refer to this as *x% load shedding mechanism*. If there is still a demand-supply mismatch after the x% load shedding and the system has run out of flexible UPS power, the green manager will enter to power management Stage-III.

4.3 Stage III: Significant Power Supply Drop

Our system enters Stage-III when it realizes that moderate load tuning in Stage-II cannot handle the significant power mismatch. The Stage-III is an emergency state since in this scenario the green manager might put the load into minimum power state and use reserved UPS capacity to avoid server shutdown.

Saving UPS Reserved Capacity:

Maintaining an appropriate level of stored energy is important to ensure service availability. In this stage we trade off performance for higher reserved UPS capacity. We first decrease load power demand, and then use stored energy to bridge the remaining power gap.

Deadline-Aware Load Shedding:

GreenWorks uses a deadline-aware load shedding to achieve a better tradeoff between UPS capacity and job execution time increase. Figure 8 shows the algorithm for our deadline-aware load shedding.

The green manager first checks the current ETI values of all the jobs for load shedding opportunities. It calculates a *Time Budget* which evaluates if a job could meet its deadline in the future with frequency boosting techniques. For example, if the monitored CPU utilization μ is 50% (i.e., CPU time is 50% of the job runtime), a 20% frequency increase in the future is expected to reduce 50% × (1-1/1.2) = 0.08s execution time for one second frequency boost.

To estimate the total *Time Budget*, one must know the chances (%) of enabling boosted processing speed. In this study we use historical renewable power traces to estimate the changes of receiving additional renewable power. To further improve accuracy, one can combine our estimation with weather forecasting.

Requires: ***The value of power shortfall after Stage II: Shortfall***
Initialize: ***The mean percentage of CPU time (i.e., utilization): μ***
The duty ratio of performing turbo boost: D
The likelihood of receiving adequate renewable power: P

```
1:   // 1st step of Stage III: decrease load power demand
2:   for each job j in the TimeTable
3:        Saving = μ ×(1 – 1/FreqSpeedup) ;
4:        TimeBudget = RemainingRuntime × D× P × Saving;
5:        if the execution time increase of job j < TimeBudget then
6:             if ( Freq. of j > MinFreq ) & (Shortfall >0) then
7:                  Lower the node frequency for job j;
8:                  Re-evaluate Shortfall;
9:             if Shortfall < 0 then break;
10:  // 2nd step of Stage III: use reserved UPS energy if have to
11:  if Shortfall > 0 then decrease load in round-robin fashion
12:  Re-evaluate Shortfall;
13:  if Shortfall > 0 then
14:       if Shortfall < upsReserve then
15:            release UPS power;
16:       else shut down servers
```

Figure 8: Power management pseudo code for Stage-III.

Assuming that the given job has 1 hour remaining execution time and the chance of receiving adequate green power is 60%, the anticipated time of being in Stage I is 3600s × 60% = 2160s. However, the actual turbo boost duration is far less than this value. In Figure 5, a duty ratio *D* is defined as the percentage of one period in which the CPU is over-clocked. The value of *D* is hardware-specific and is used to control the thermal headroom of processors. If the duty ratio is 30%, the anticipated turbo boost duration is 3600s × 60% × 30% = 648s. Therefore the total *Time Budget* is 648s × 0.08s/s = 52s. This means that the given job can tolerate up to 52s ETI at the current timestamp.

If the given job has enough *Time Budget,* our controller will incrementally reduce its CPU frequency (Δf=0.1GHz) until it reaches its lowest speed (*MinFreq* = 1.6GHz). It will put server nodes into low power states in a round-robin fashion if the demand-supply discrepancy still exists. Finally, we release the reserved UPS energy if necessary. In this study we assume that each node runs independent data-processing task. Parallel workloads are often not accelerated as much as calculated since the accelerated threads or processes have to wait for others. Exploring workloads with high communication to computation ratio is our future work.

Note that we assume that a job's runtime is known a priori. Typically, HPC users are required to submit their job runtime estimations to enable backfilling, which can help maximize cluster utilization. In this study we leverage it to determine job deadline.

4.4 Managing Baseload at Coarse-Grained Interval

At each fine-grained timestamp (e.g., every 1 second), the green manager adjusts the load processing speed and manages the stored energy. The objective is to mitigate power mismatch caused by the variability issue in the intermittent power supply and server load.

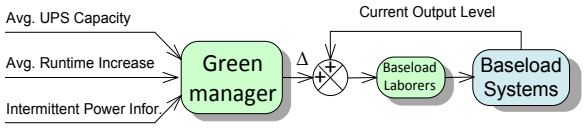


Figure 9: Feedback control for managing baseload power.

At each coarse-grained timestamp (15 minutes), it adjusts the baseload power generation level through the baseload laborers, as shown in Figure 9. The green manager collects the monitored information at the end of each coarse-grained control period; it then adjusts the output of the baseload power supply based on the average power supply shortfall in the last control period. The green manager can also incrementally add additional baseload power (10% of the current output level) if the monitored UPS capacity is low (*upsFlexible* = 0), or the workload performance is low (e.g., 80% of the jobs would be delayed), or the anticipated wind power availability is low (e.g., P > 80% in Figure 8).

5. Evaluation Methodologies

We develop a simulation framework for datacenters powered by renewable generation mix. As shown in Figure 10, this framework is configured into three layers for modeling the entire system from the job dispatching behavior to the power system specifics. It uses discrete-event simulation to process a chronological sequence of job submissions. It also simulates the power behavior of renewable energy system on per-second time scale which is in tune with our datacenter job scheduler. This three-layer framework provides us the flexibility in analyzing various design spaces.

We adopt renewable energy system model from HOMER [35]. Table 2 shows the parameters we used. All the values are carefully selected based on manufacturer's specifications, government publications and industry datasheet. The maximum baseload power output in our simulator equals to the average power demand of the evaluated datacenter workload. The default load following interval is 15 minutes. The capacity of our simulated battery cell is 24Ah at a 20-hour rate (1.2A discharge current). Its capacity is 10Ah at a 15-minute rate (40A discharge current). We determine the total battery capacity in such a way that the backup power system can ensure 15 minutes power output in emergency. We maintain detailed log of each discharging event to calculate battery life using methods in [33, 34].

We use wind turbine as our evaluated intermittent power source since it is widely used to provide abundant and affordable green energy for large-scale facilities. We collect minute-by-minute wind speed data from the National Wind Technology Center [36] during the month of March, 2012, as shown in Figure 11. We calculate wind power based on the wind speed data and the wind turbine output curve.

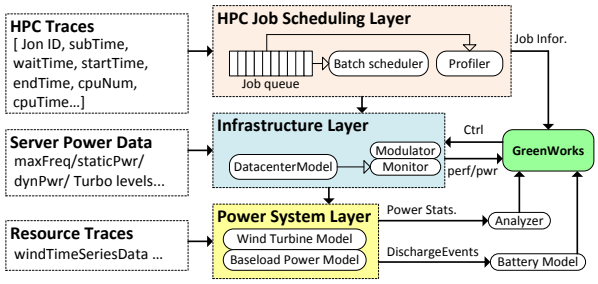


Figure 10: Details of our three-layer simulation platform.

Inputs	Typical Value	Value Used
Load Following Interval	5 min ~ 1 hour	15 min
Battery Life Cycle	5,000 ~ 20,000	10,000 times
Rated Depth of Discharge	0.8	0.8
Battery Efficiency	75% ~ 85%	80%
Max Charging Current	N/A	8 Ah
Peukert Coefficient	1.0~ 1.3	1.2
UPS Installed Backup	10~20 min	15 min

Table 2: Key parameters used in the simulation [21-25].

We use a queueing-based model that takes real workload traces as input. It uses a first come first serve (FCFS) policy and puts each job request into a queue and waits to grant allocation if computing nodes are available. Each job request in the trace has exclusive access to its granted nodes for a bounded duration. Such trace-driven simulation has been adopted by several prior studies on datacenter behaviors and facility-level design effectiveness [8, 19, 37, 38].

We use real-world workload traces from a well-established online repository [39]. As shown in Table 3, these workload activity logs are collected from state-of-the-art HPC systems in production use around the world. We select five key task parameters in each trace file: job arrival time, job start time, job completion time, requested duration, and job size in number of granted CPU nodes. As shown in Table 3, we select eight 1-week workload traces that have different mean utilization level and mean job runtime.

Our datacenter infrastructure is based on the IBM System x3650 M2 (2.93G Intel Xeon X5570 processor) high-performance server which supports Intel Turbo Boost technology. While the number of performance states (P-states) is processor specific, we assume 12 P-states as indicated in [40]. The minimum frequency is 1.6GHz and the normal frequency is 2.9GHz. In Turbo Boost mode, the processor could increase the frequency by 14%. We increase the frequency moderately (i.e., 10%) when the Turbo Boost mode is enabled. Our power model uses CPU utilization as the main signal of machine-level activity. Prior work has shown that CPU utilization traces can provide fairly accurate server-level power prediction [41]. According to the published SPEC power data, the modeled system consumes 244 Watts at full utilization and 76 Watts when idle [42].

Traces	Descriptions	Load	Mean Inter-arrival		Avg. Job Run Time	
Thunder	Lawrence Livermore Lab's 4096-CPU capacity cluster called Thunder	61%	1.8 min	Short	0.58h	Short
DataStar	San Diego Supercomputer Center's 184-node cluster DataStar	56%	3.5 min		1.41h	
Atlas	Lawrence Livermore Lab's 9216-CPU capability cluster called Atlas	33%	11 min	Long	0.61 h	
BlueGene	A 40-rack large Blue Gene/P system at Argonne National Lab	26%	8.4 min		1.4h	
RICC	A massively parallel Japanese cluster of cluster with 1024 nodes	49%	0.9 min	Short	16.6 h	Long
MetaC	Czech national grid infrastructure called MetaCentrum	67%	2.1 min		11.8 h	
Seth	A 120-node European production system named Seth	80%	21 min	Long	6.2 h	
iDataPlex	320-node IBM iDataPlex cluster for Climate Impact Research	18%	50 min		3.7h	

Table 3: The evaluated real-world workload traces in representative HPC datacenters [39].

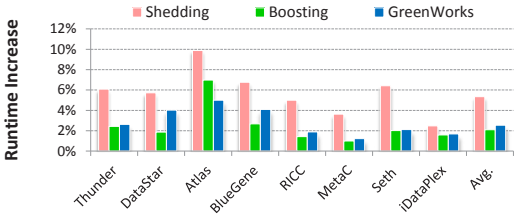


Figure 12: Average increase of job turnaround time (i.e., the average ETI for all the processed jobs).

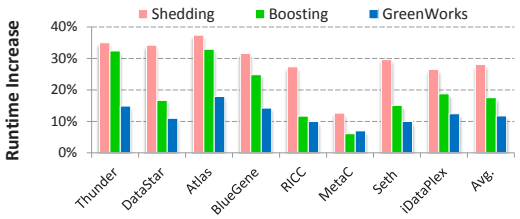


Figure 13: Maximum increase of job turnaround time (i.e., the average ETI for the worst 5% delayed jobs).

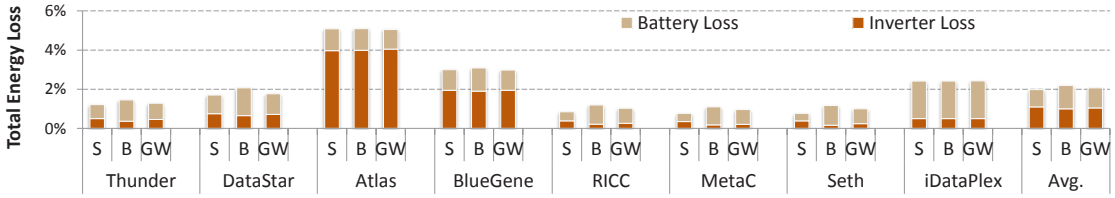


Figure 14: GreenWorks (GW) maintains almost the same green energy utilization efficiency as *Shedding* (S) and *Boosting* (B).

6. Results

In this section we evaluate the benefits of applying GreenWorks to datacenters powered by hybrid onsite green power supplies. We compare GreenWorks to two state-of-the-art baselines: *Shedding* and *Boosting*. *Shedding* is a widely used load management schemes for emerging renewable energy powered datacenters [43, 44]; *Boosting* represents recent datacenter power management approaches that emphasis the role of energy storage devices [45, 46]. Both baselines use UPS and server load scaling to manage fine-grained power shortfall and adjust baseload output level at each end of the control period. The only difference between the two is that *Shedding* gives priority to load scaling, while *Boosting* gives priority to UPS stored energy.

6.1 Execution Time

We evaluate datacenter performance in terms of average job turnaround time increase compared to an oracle (which always ensures full processing speed with zero service downtime). Figure 12 shows the average job execution time increase. On average, the job execution time increase of *Shedding*, *Boosting* and GreenWorks are 5.4%, 2.1%, and 2.4%, respectively. Compared to *Shedding*, *Boosting* shows less execution time increase since it trades off UPS capacity for performance. As GreenWorks seeks a balanced power management across different power supplies, it yields slightly higher ETI compared to *Boosting*.

The performance of the worst 5% jobs could significantly affect the service-level agreements (SLA) of datacenters. Figure 13 shows the maximum increase of job turnaround time which is calculated as the average execution time increase of the 5% worst cases. The worst-case result of *Shedding* is 28%. Surprisingly, GreenWorks (12%) reduces the maximum job execution time increase by 33%, compared to *Boosting* (18%). The improvement is due to the *x% shedding mechanism* (detailed in Section 4.2). By modifying the value of the *x*, one can easily adjust the performance goal of GreenWorks (detailed in Section 6.5).

6.2 Energy Efficiency

The main sources of inefficiency in green datacenters are the battery round-trip power loss and the power conversion loss in the grid-tied inverter. We assume a typical battery system of 80% round-trip energy efficiency and a power inverter of 92% energy efficiency.

GreenWorks could maintain the same energy efficiency as *Shedding* and *Boosting*. In Figure 14 we show the total energy loss due to the battery round-trip energy loss and the inverter's power conversion loss. The overall efficiencies of the three evaluated schemes are very close to each other. The differences are less than 0.5%. Compared to the other two, *Boosting* shows relatively lower inverter loss because it can maximally leverage the power smoothing effect of UPS battery to reduce the amount of power feedback.

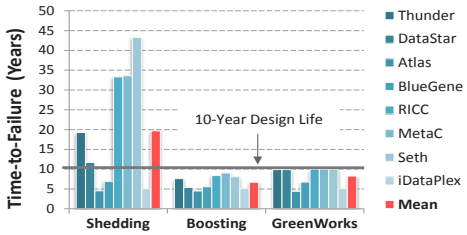


Figure 15: The estimated battery lifetime based on detailed battery charging/discharging statistics.

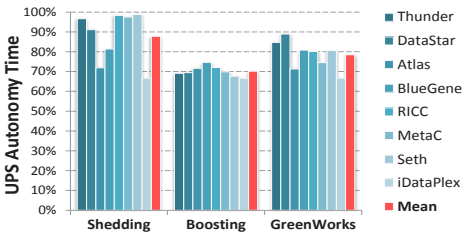


Figure 16: The normalized backup time throughout the evaluated operation duration (normalized to rated backup time).

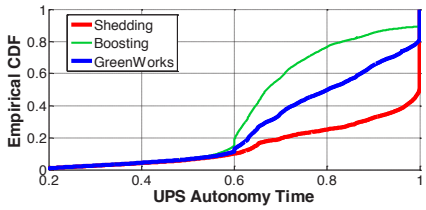


Figure 17: Cumulative distribution function (CDF) for the normalized UPS autonomy time.

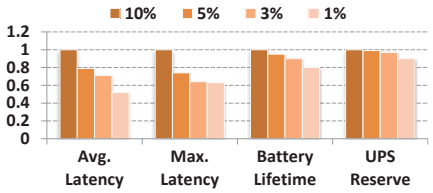


Figure 18: Sensitivity to various performance capping requirements. The default performance threshold is 10%.

6.3 Battery Lifetime

Typically the rated lifetime of a valve-regulated lead–acid battery (VLRA) is 3 years to 10 years [47]. In Figure 15, GreenWorks shows a near-threshold battery life (8.3 years). It means our multi-source multi-objective power management can maximally leverage batteries without degrading their life significantly. In contrast, *Boosting* shows a mean lifetime of 6.7 years; and *Shedding* shows a mean lifetime of 19.7 years. Typically, the battery lifetime is not likely to exceed 10 years [47]. The reason *Shedding* over-estimates battery life is that the system underutilizes batteries. Since batteries may fail due to various aging problems and self-discharging issues, it is better to fully utilize it.

6.4 UPS Backup Time

Another advantage of GreenWorks is that it can optimize the mean UPS autonomy time. The autonomy time is also known as backup time. It is a measure of the time for which the UPS system will support the critical load during an unexpected power failure. Figure 16 shows the mean normalized UPS autonomy time throughout the operation duration for various datacenter traces and different power management schemes. On average, the mean autonomy time is: *Shedding* (88%), *Boosting* (70%), and GreenWorks (78%).

In Figure 17 we plot the cumulative distribution function (CDF) for the normalized UPS autonomy time. Our results show that the CDF curve of GreenWorks lays nicely between our two baselines: *Shedding* and *Boosting*. GreenWorks could ensure rated backup time (the discharge time of a fully charged UPS) for 20% of the time. *Shedding* maintains its rated backup time for 50% of the time and the number for *Boosting* is only 10%. This is because *Boosting* uses UPS battery much more aggressively than *Shedding*.

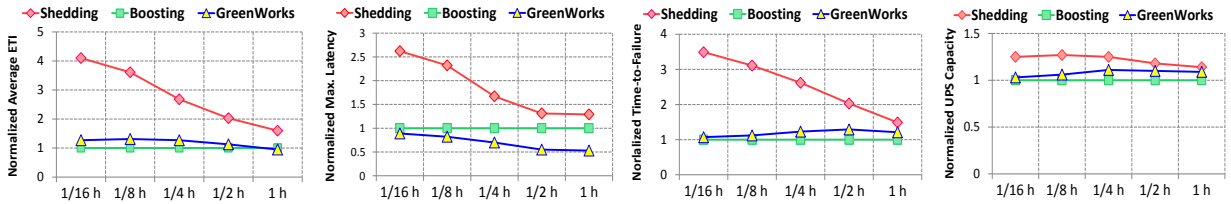
Energy storage devices should be always taken care of. A lower autonomy time can pose significant risk as the backup generator may not be ready to pick up the load. Without appropriate power management and coordination, datacenters have to increase their installed UPS capacity, which is both costly and not sustainable.

6.5 Control Sensitivity

We also evaluate the control sensitivity of our system by varying the value of several key parameters.

In Figure 18 we first show the impact of the *x% shedding mechanism* (detailed in Section 4.2) on various performance metrics of GreenWorks. The default value of the performance limit in our study is 10% and we evaluate the performance impact when the user lowers the threshold. As can be seen, the *x% shedding mechanism* has a much larger impact on the average latency, other than the battery lifetime and UPS capacity. Decreasing the threshold (i.e., the *x*) can reduce the job execution time and increase the reliance on energy storage elements, which will lower the battery lifetime and backup capacity to some extent.

In Figure 19 we further evaluate the impact of the control intervals (load following intervals of the baseload power supply) on the performance of our multi-source driven multi-objective control. Our default interval of adjusting the baseload power is 15 minutes. All the results are normalized to that of *Boosting*. They show that the job latency drops as the control interval becomes larger. The battery lifetime and UPS capacity of GreenWorks both rise as we increase the length of the control interval. Note that although the relative latency may decrease as load following interval increases, the actual value of latency increases. A longer interval often degrades load following effectiveness, and therefore increases the chance of power mismatch.



(a) Job ETI (b) Worst-case ETI (c) Battery time-to-failure (d) UPS reserved capacity

Figure 19: Sensitivity to various load following intervals of the baseload power supply.

7. Related Work

Managing computer systems on green energy has been done at various levels. However, existing designs mainly focus on certain specific type of green energy sources (i.e., intermittent power or baseload generators) and overlook the benefits of cross-source coordination.

Managing Intermittent Power Source: Prior studies on this issue mainly focus on load adaptation schemes which can be broadly categorized into three types: load shedding, load deferring, and load migration. For example, SolarCore [6] is a load shedding based design. It temporarily lowers server power demand using per-core power gating when solar power drops. [48] and [49] investigate server power adaptation under intermittent power budget. Load deferring, also known as load shifting, leverages the flexibility of job scheduling [9, 18, 50]. It re-schedules load by shifting user requests to a future time horizon if renewable power is currently not available. Load migration based design focuses on re-allocating application to another datacenter that has reserved capacity [8]. With intelligent workload packing and virtual machine placement [51], one could further minimize resource wastage and power consumption in green datacenters.

Managing Baseload Power Generation: Several recent proposals have explored baseload power supply in datacenters. The most similar studies are [52] and [19]. In [52], the authors propose design methodology for sustainable datacenters powered by onsite generation. However, they mainly focus on high-level datacenter infrastructure management policies. In [19], we investigate the benefits of load following mechanism in distributed generation powered datacenters. However, [19] does not consider the power variability issue of intermittent green power integration.

Managing Backup Power / UPS Systems: There have been several studies exploring the use of backup power systems for energy-efficient datacenters. For example [31, 45, 46, 53] investigate the use of energy storage (particularly the UPS system) to manage the datacenter peak power. For example, [31] explore the TCO of the distributed UPS system in datacenters and propose using local distributed UPS to shave the datacenter peak power. Govindan et.al, [53] use UPS as the major tuning knob for minimizing power cost in aggressively under-provisioned datacenter infrastructure.

Cost-Aware Green Energy Scheduling: The system cost-effectiveness also receives many attentions in renewable energy powered datacenter. For example, [10] proposes algorithms that minimize fossil fuel-based energy consumptions; [11] discusses load balancing on distributed datacenters. Recent work in capacity planning for datacenters also looks at the cost issue of green energy purchases [3].

In contrast to prior work, this paper explores hierarchical, cross-layer power management for datacenters powered by hybrid renewable energy systems. We consider an integrated mix of complementary power provisioning methods that include intermittent power supply, baseload power generation, and energy storage devices.

8. Conclusions

Although emerging green power systems are often centrally installed at the datacenter level, maximizing the overall efficiency requires a multi-level, cooperative power management strategy. We propose GreenWorks, a novel framework that could greatly facilitate multi-source based green datacenter design. GreenWorks enables datacenters to make informed power management decisions based on the available baseload power output, renewable power variability, battery capacity, and job performance. We show that GreenWorks could achieve less than 3% job runtime increase, extend battery life by 23%, increase UPS backup time by 12%, and still maintain desired energy utilization efficiency.

Acknowledgement

We would like to thank our shepherd Ming Zhao and the anonymous reviewers for their insightful comments and feedbacks.

This work is supported in part by NSF grants 1320100, 1117261, 0937869, 0916384, 0845721 (CAREER), 0834288, 0811611, 0720476, by SRC grants 2008-HJ-1798, 2007-RJ-1651G, by Microsoft Research Trustworthy Computing, Safe and Scalable Multi-core Computing Awards, by NASA/Florida Space Grant Consortium FSREGP Award 16296041-Y4, by three IBM Faculty Awards, and by NSFC grant 61128004. Chao Li is also supported by University of Florida Graduate Fellowship, Yahoo! Key Scientific Challenges Program Award, and Facebook Fellowship. Jingling Yuan is supported by NSFC grant 61303029.

References

[1] DCD Industry Census 2011: Forecasting Energy Demand, 2011, www.dcd-intelligence.com

[2] Data Center Carbon Calculator, http://www.apcmedia.com/salestools/WTOL-7DJLN9_ R0_EN.swf

[3] C. Ren, D. Wang, B. Urgaonkar, and A. Sivasubramaniam, Carbon-Aware Energy Capacity Planning for Datacenters, *IEEE Int'l Symp. on Modeling, Analysis & Simulation of Computer and Telecommunication Systems*, 2012

[4] N. Deng, C. Stewart, D. Gmach, M. Arlitt, and J. Kelley, Adaptive Green Hosting, *Int'l Conf. on Autonomic Computing*, 2012

[5] M. Haque, K. Le, I. Goiri, R. Bianchini, and T. Nguyen, Providing Green SLAs in High Performance Computing clouds. *Int'l Green Computing Conference* , 2013

[6] C. Li, W. Zhang, C. Cho, and T. Li, SolarCore: Solar Energy Driven Multi-core Architecture Power Management, *IEEE Int'l Symp. on High-Performance Computer Architecture* , 2011

[7] N. Deng, C. Stewart, J. Kelley, D. Gmach, and M. Arlitt, Adaptive Green Hosting, *Int'l Conf. on Autonomic Computing* , 2012

[8] C. Li, A. Qouneh, and T. Li, iSwitch: Coordinating and Optimizing Renewable Energy Powered Server Clusters, *Int'l Symp. on Computer Architecture* , 2012

[9] I. Goiri, K. Le, T. Nguyen, J. Guitart, J. Torres, and R. Bianchini, GreenHadoop: Leveraging Green Energy in Data-Processing Frameworks, *ACM EuroSys*, 2012

[10] K. Le, R. Bianchini, M. Martonosi, and T. D. Nguyen, Capping the Brown Energy Consumption of Internet Services at Low Cost," *Int'l Green Computing Conference*, 2010

[11] Z. Liu, M. Lin, A. Wierman, S. Low and L. Andrew, Greening geographical load balancing, *ACM Int'l Conf.e on Modeling and Measurement of Computer Systems*, 2011

[12] http://biomassmagazine.com/articles/8351/microsoft-data-center-to-install-biogas-fuel-cell-power-plant

[13] http://www.datacenterknowledge.com/archives/2012/05/30/hp-developing-net-zero-data-center-concept/

[14] http://green.ebay.com/greenteam/ebay/blog/Building-a-Greener-Company/26

[15] http://www.apple.com/environment/renewable-energy/

[16] Enabling the Low Carbon Economy in the Information Age, http://www.smart2020.org

[17] G. Burch, Hybrid Renewable Energy Systems, Natural Gas / Renewable Energy Workshops, U.S. Department of Energy, 2001

[18] I. Goiri, W. Katsak, K. Le, T.D. Nguyen, and R. Bianchini, Parasol and GreenSwitch: Managing Datacenters Powered by Renewable Energy, *Int'l Conf. on Architectural Support for Programming Languages and Operating System*s, 2013

[19] C. Li, R. Zhou, and T. Li, Enabling Distributed Generation Powered Sustainable High-Performance Data Center, *IEEE Int'l Symp. on High-Performance Computer Architecture* , 2013

[20] R. Lasseter and P. Piagi, Microgrid: A Conceptual Solution, *IEEE Annual Power Electronics Specialists Conference* , 2004

[21] D. Salomonsson, L. Soder, and A. Sannino, An Adaptive Control System for a DC Microgrid for Data Centers, *IEEE Transactions on Industry Applications*, pp. 1910 – 1917, Volume:44, Issue:6

[22] R. Wang, N. Kandasamy, C. Nwankpa, and D. Kaeli, Datacenters as Controllable Load Resources in the Electricity Market, *Int'l Conf. on Distributed Computing Systems*, 2013

[23] W. Deng, F. Liu, H. Jin, and C. Wu, SmartDPSS: Cost-Minimizing Multi-source Power Supply for Datacenters with Arbitrary Demand, *Int'l Conf. on Distributed Computing Systems*, 2013

[24] B. Kirby and E. Hirst, Customer-Specific Metrics For the Regulation and Load-following Ancillary Services, Technical report, ORNL, 2000

[25] The Role of Distributed Generation and Combined Heat and Power (CHP) Systems in Data Centers, Technical Report, US EPA, 2007

[26] S. Chowdhury and P. Crossley, Microgrid and active distribution networks, The Institute of Engineering and Technology, 2009

[27] Fuel Cell Technologies Program Multi-year Research, Development and Demonstration Plan, Technical Report, US Department of Energy

[28] H. Zareipour, K. Bhattacharya, and C. Canizares, Distributed generation: current status and challenges, *the 36th Annual North American Power Symposium*, 2004

[29] The Importance of Flexible Electricity Supply, Solar Integration Series, Technical Report, U.S. Department of Energy, 2011

[30] National Survey on Data Center Outages, Ponemon Institute, White Paper, 2010

[31] V. Kontorinis, L. Zhang, B. Aksanli, J. Sampson, H. Homayoun, E. Pettis, T. Rosing, and D. Tullsen, Managing Distributed UPS Energy for Effective Power Capping in Data Centers, *Int'l Symp. on Computer Architecture* , 2012

[32] X. Fan, W. Weber, and L. Barroso, Power Provisioning for a Warehouse-Sized Computer, *Int'l Symp. on Computer Architecture* , 2007

[33] H. Bindner, T. Cronin, P. Lundsager, J. Manwell, U. Abdulwahid, and I. Gould, Lifetime Modelling of Lead Acid Batteries, Technical Report, Risø National Laboratory, 2005

[34] M. Jongerden and B. Haverkort, Which Battery Model to Use?, the 24th UK Performance Engineering Workshop, 2008

[35] Getting started guide for HOMER version 2.1, National Renewable Energy Laboratory, 2005

[36] National Wind Technology Center (NWTC), http://www.nrel.gov/wind/

[37] S. Pelley, D. Meisner, P. Zandevakili, T. Wenisch and J. Underwood, Power Routing: Dynamic Power Provisioning in the Data Center, *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, 2010

[38] F. Ahmad and T. Vijaykumar, Joint Optimization of Idle and Cooling Power in Data Centers While Maintaining Response Time, *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, 2010

[39] Logs of Real Parallel Workloads, http://www.cs.huji.ac.il/labs/parallel/workload

[40] Host Power Management in VMware vSphere 5, Technical Report, VMware, 2010

[41] P. Ranganathan, P. Leech, D. Irwin, and J. Chase, Ensemble-level Power Management for Dense Blade Servers, *Int'l Symp. on Computer Architecture*, 2006

[42] SPEC power_ssj2008 http://www.spec.org/power_ssj2008/

[43] G. Ghatikar, V. Ganti, N. Matson, M. Piette, Demand Response Opportunities and Enabling Technologies for Data Centers: Findings from Field Studies, Technical Report, Lawrence Berkeley National Laboratory, 2012

[44] H. Xu, U. Topcu, S. Low, C. Clarke, and K. Chandy, Load-shedding Probabilities with Hybrid Renewable Power Generation and Energy Storage, *the 48th Annual Allerton Conference on Communication, Control, and Computing*, 2010

[45] D. Wang, C. Ren, A. Sivasubramaniam, B. Urgaonkar, and H. Fathy, Energy Storage in Datacenters: What, Where, and How much?, *ACM Int'l Conf. on Modeling and Measurement of Computer Systems*, 2012

[46] S Govindan A. Sivasubramaniam and B. Urgaonkar , Benefits and Limitations of Tapping into Stored Energy for Datacenters, *Int'l Symp. on Computer Architecture*, 2011

[47] S. McCluer, Battery Technology for Data Centers and Network Rooms: Lead-Acid Battery Options, APC White Paper #30

[48] C. Li, R. Wang, N. Goswami, X. Li, T. Li, and D. Qian, Chameleon: Adapting Throughput Server to Time-Varying Green Power Budget Using Online Learning, *Int'l Symp. on Low Power Electronics and Design*, 2013

[49] C. Li, Y. Hu, R. Zhou, M. Liu, L. Liu, J. Yuan, and T. Li, Enabling Datacenter Servers to Scale Out Economically and Sustainably, *Int'l Symp. on Microarchitecture*, 2013

[50] I. Goiri, R. Beauchea, K. Le, T.D. Nguyen, M. Haque, J. Gui-tart, J.Torres, and R. Bianchini, GreenSlot: Scheduling Energy Consumption in Green Datacenters, *Int'l Conf. for High Performance Computing, Networking, Storage and Analysis*, 2011

[51] J. Xu, and J. Fortes, A Multi-Objective Approach to Virtual Machine Management in Datacenters, *Int'l Conf. on Autonomic Computing*, 2011

[52] P. Banerjee, C. Patel, C. Bash, and P. Ranganathan, Sustainable Data Centers: Enabled by Supply and Demand Side Management, *Design Automation Conference* , 2009

[53] S. Govindan，D. Wang， A. Sivasubramaniam, and B. Urgaonkar, Leveraging Stored Energy for Handling Power Emergencies in Aggressively Provisioned Datacenters, Battery Emergency, *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, 2012

WattValet: Heterogenous Energy Storage Management in Data Centers for Improved Power Capping

Shen Li, Shaohan Hu, Shiguang Wang, Siyu Gu, Chenji Pan, Tarek Abdelzaher
University of Illinois at Urbana-Champaign
{shenli3, shu17, swang83, siyugu2, cpan8, zaher}@illinois.edu

Abstract

This paper presents WattValet, an efficient solution to reduce data center peak power consumption by using heterogeneous energy storage. We henceforth call an energy storage device, a *battery*, with the understanding that the discussion applies to other devices as well such as pumped hydraulic and thermal systems. Previous work on energy storage management in data centers often ignores or underestimates their degree of heterogeneity. Even if batteries used in a data center are of the same model and purchased at the same time, differences in storing temperature and humidity, as well as discharging cycles and depth, gradually drive their characteristics apart. We show that differences in battery characteristics, such as discharge rates, if not fully accounted for, can lead to significantly suboptimal power caps. A new algorithm, called WattValet, is described that reduces peak power consumption by efficiently exploiting heterogeneity. Evaluation using Wikipedia traces shows that the power cap generated by WattValet is within 2% of the optimal solution, whereas WattValet finishes the computation orders of magnitude faster than the optimal solution even in small-scale experiments.

1 Introduction

With increased datacenter power consumption and growing concerns for sustainability of large-scale computing systems, reducing datacenter power becomes an increasingly important problem [9, 5, 17, 18, 4, 6]. Much recent work in both industry and academia addressed the challenge of shaving off power peaks using energy storage devices (*e.g.*, batteries) [7, 10, 16, 14]. Early work considered smarter charging and discharging strategies for energy storage systems [14, 16, 7, 2] to achieve improved power capping. However, they took into account only homogeneous environments, where all batteries are the same [14], or allowed for a very limited heterogeneity [7, 2]. In reality, even in homogeneous systems, battery characteristics gradually deviate from each other due to different storing temperatures, charging/discharging cycles, aging, and other factors. Hence, heterogeneity has to be considered explicitly.

Our work leverages the observation that load and power demand in data centers often follow clearly repeated patterns [3, 14, 15]. That is to say, by extracting patterns from historical traces, future power demand can be predicted. In principle, given a power demand pattern and battery characteristics as input parameters, the optimal power capping problem can be formulated as a linear programming problem and solved using an LP solver. However, if the system contains hundreds of batteries, and thousands of time slots, LP solutions cannot guarantee to generate optimal results in a reasonable amount of time. For example, Wikipedia's trace shows a weekly pattern. If it is cut into 10-minute time slots, there will be more than 1000 time slots in the power demand pattern. Hence, clever approximations must be developed that significantly reduce computational overhead without tangibly degrading solution quality, which motivates this work.

The paper describes WattValet, an efficient solution to reduce datacenter peak power consumption using heterogeneous energy storage. WattValet takes a predicted power demand pattern and a set of heterogeneous batteries as input. It searches for a battery charging/discharging schedule that minimizes the power consumption peak. The search space is very large as it is a function of all variables indicating the amount of energy charged into/discharged from each battery at each time slot, and the starting time slot in the cyclic power demand pattern. The contribution of the paper lies in developing efficient heuristics that take into account not only battery capacity and efficiency but also maximum charge and discharge rates, leading to an improved power cap compared to prior art.Evaluation results show that the power cap achieved by WattValet is only 2% higher than the optimal value in the worst case, which significantly outperforms state-of-the-art solutions.

The remainder of this paper is organized as follows. The system model is described in Section 2. Section 3 elaborates the design of WattValet. Our solution is compared to the optimal solution as well as two others from recent literature in Section 4. Section 5 briefly summarizes related work. We conclude this paper in Section 6.

2 System Model

Datacenter workload often follows a clear periodic pattern. As an example, Figure 1 plots the English Wikipedia's workload in 2008 [15], presenting an obvious weekly pattern. Therefore, future workload can be predicted with accuracy from historical traces. Let P denote the power demand pattern. Hence, P can be represented by a repeating time series of period, T, composed

This work was sponsored in part by the National Science Foundation under grants CNS 13-20209, CNS 13-02563, CNS 10-35736 and CNS 09-58314.

of successive slots, such that the average demand in slot k is denoted by $P[k]$, where $k = 1, 2, \ldots, T$.

Without an energy storage capability, the power cap P^c needs to be greater than the peak power consumption in the demand time series (*i.e.*, $P^c \geq \max\{P[k]|k = 1, 2, \ldots, T\}$). The presence of batteries can lower the power cap by charging during power demand vallies and discharging during power demand spikes.

Assume the datacenter is equipped with a centralized pool of B batteries [2] that serves the entire datacenter. We model each battery b with four parameters: maximum charging rate r_b^c, maximum discharging rate r_b^d, energy storage capacity c_b, and efficiency η_b. The maximum charging (discharging) rate represents the maximum amount of energy the battery can charge (discharge) in one time slot. The parameter c_b denotes the maximum amount of energy that battery b can store at any time. For each unit of energy spent on charging battery b, only η_b goes into the battery, which represents the battery efficiency. Please note, even if all batteries are of the same model, with the passage of time, their characteristics will gradually deviate due to differences in the environment (*e.g.*, temperature, humidity, etc.) and usage (*e.g.*, charging/discharging cycles and depth) [8].

We further assume that the batteries are equipped with on/off switches to connect or disconnect them for charging or for discharging purposes [2]. When a battery is connected it can charge or discharge at a rate no larger than its maximum charge or discharge rate. When a battery is disconnected it does not participate in charging or discharging.

The objective of this paper is to determine how much each battery should charge or discharge in each time slot such that the power cap is minimized, while being able to meet the power demand in every time slot. Let $u_b[k]$ represent the amount of energy that battery b is supplied (or, if negative, discharged) in time slot k. Hence, the solution sought in the paper is to compute $u_b[k]$ for all b and k, such that the demand is met and the power cap, P^c, is minimized. This can be modeled as a linear programming problem as shown below:

$$
\begin{array}{lrll}
\textbf{\textit{min}} & & P^c & \\
\textbf{\textit{s.t.}} & \forall k: -r_b^d \leq u_b[k] & & \leq r_b^c \\
& \forall k: x_b[k] + u_b[k] & & \geq 0 \\
& \forall k: \sum_{b=1}^{B} u_b[k] - P^c + P[k] & & \leq 0 \\
& x_b[0] = 0, \quad \forall k: x_b[k] & & \leq c_b \\
& \forall k, P[k] > P^c: x_b[k] + \frac{u_b[k]}{\eta_b} & & = x_b[k+1] \\
& \forall k, P[k] \leq P^c: x_b[k] + u_b[k] & & = x_b[k+1]
\end{array}
\tag{1}
$$

where $x_b[k]$ is the amount of energy stored in battery b at time slot k. The first constraint guarantees that charging and discharging rates are not violated. The second constraint states that no battery can discharge more energy than it stores. The power balance equation at a given time slot is described with the third constraint. The fourth constraint enforces battery capacity. Batteries are initially empty ($x_b[0] = 0$). The last two constraints describe how the amount of energy stored in a battery is updated when the battery charges or discharges, respectively.[1]

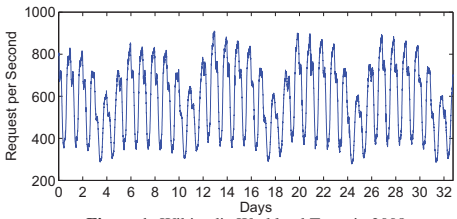


Figure 1: Wikipedia Workload Trace in 2008

The above linear programming model contains a large number of variables and constraints. For example, if the data center is equipped with 1000 batteries and the weekly pattern is divided into 10-minute slots, this model will generate more than 2 million variables and more than 4 million constraints. LP solutions cannot guarantee to finish the computation in a reasonable amount of time. Hence, we seek an approximation with low worst-case asymptotic complexity.

3 System Design

The original problem of minimizing the power cap can be transformed into a sequence of feasibility check problems. Each feasibility check needs to determine whether the predicted power demand time-series can be met for a given power cap P^c. A binary search can then find the minimum feasible power cap. Section 3.1 describes a feasibility check algorithm for a specified power demand sequence. It depends on which slot is considered to be the beginning of the sequence. Since the demand is cyclic, of period T, there are T candidate starting slots. Section 3.2 describes how to efficiently consider all possible start slots to find the best power cap overall.

3.1 Feasibility Check for a Given Power Demand Sequence

Let us call a time slot, a *charging* time slot if its average power demand is smaller than the power cap P^c. Otherwise, it is a *discharging* time slot. A set of consecutive charging (discharging) time slots is defined as a charging (discharging) *phase*. Hence, a power cap naturally divides a power demand sequence into alternating charging and discharging phases. We can refer to the j^{th} charging phase as set, CP_j, and to the ith discharging phase as set, DP_i. As graphed in Figure 2, solid green curves represent the charging phases and dashed red curves represent the discharging phases. Figure 2 also suggests that

[1] Note that, in principle, battery inefficiency affects both charging and discharging. Without loss of generality, we attribute all loss to charging, while modeling discharge as lossless. This does not change the result as long as capacity is reduced by discharge efficiency.

datacenter power demand does not change dramatically within small time intervals (*e.g.*, 5 minutes). Therefore, using the average power demand in a time slot, in lieu of the instantaneous demand curve is a good approximation.

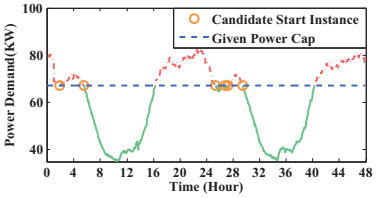


Figure 2: Candidates

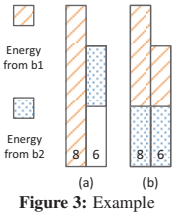


Figure 3: Example

Our algorithm alternates charging and discharging. Two main intuitions guide the design:

- *Intuition 1: Minimize lost energy:* Lost energy is energy that is wasted due to battery inefficiency. To minimize lost energy, charging should *exploit more efficient batteries first*. By exploiting them, we mean (i) charge them first and (ii) discharge them first, in order to incur minimum loss. Other less efficient batteries should be used only when the more efficient ones do not suffice.

- *Intuition 2: Minimize locked energy:* Locked energy is energy that is charged but not discharged within a given cycle. Specifically, the discharge rate of a battery limits the total amount of energy that it can discharge in different discharge phases. There is never a need to charge a battery beyond the maximum amount it can discharge, as such energy will in effect be "locked" and not utilized.

Note that, simple greedy algorithms that exploit the most efficient batteries first can result in some amount of locked energy, for example, if the maximum discharge rate of these batteries was low. In contrast, by putting a limit on how much each battery is allowed to charge (based on the amount it can ever discharge), we are able to outperform state of the greedy solutions. Below, we describe the algorithm in more detail.

3.1.1 Spatial Energy Allocation According to *Intuition 1*, we order batteries in decreasing order of efficiency, such that we consider the most efficient battery first. Let us number them $1, ..., B$, such that $\eta_b \geq \eta_{b+1}$. For each battery, we need to determine how much to charge or discharge it in each time slot.

According to *Intuition 2*, we need to limit battery charging to the maximum amount that a battery can discharge. To compute the latter, it is first useful to define the notion of energy shortfall in slots, k, where demand $P[k]$ exceeds the power cap, P^c. The initial energy shortfall in a discharging slot k, denoted $\mathcal{P}^0[k]$, is given by the difference between the demand and the power cap. After batteries $1, ..., b \leq B$ have been allocated, the remaining shortfall becomes $\mathcal{P}^b[k]$. Hence:

$$\begin{aligned} \mathcal{P}^b[k] &= \mathcal{P}^{b-1}[k] + u_b[k] \\ \mathcal{P}^0[k] &= P[k] - P^c \end{aligned} \tag{2}$$

Let us now compute the maximum amount of energy a battery can discharge in a set of time slots, DP_i, belonging to the ith discharging phase. Let md_b[i] be the maximum amount of energy that battery b may discharge in that phase, when the battery starts full. The amount can be computed as:

$$md[i] = \min\left\{c_b, \sum_{k \in DP_i} \min\left(\mathcal{P}^b[k], r_b^d\right)\right\} \tag{3}$$

The equation states that the maximum energy discharged in each time slot by battery b is the minimum of the remaining shortfall in that time slot, $\mathcal{P}^b[k]$, and the maximum discharge rate, r_b^d. The maximum discharge over the entire phase is then the minimum of the discharge sum over all time slots, and battery capacity, c_b.

Next, we extend the result to multiple discharge phases. First, we define $mc_b[i][j]$ as the maximum amount of energy that battery b may carry from charging phase j (CP_j) to discharging phase i (DP_i), $j \leq i$, which is bounded by the summation of the amount of energy it may charge in each time slot and the remaining capacity. For mc_b[i][i], the remaining capacity is the battery capacity c_b, as battery b has not been used in charging phase i before and there is no phases between charging and discharging phase i:

$$mc_b[i][i] = \min\left\{c_b, \sum_{k \in CP_i} \min\left(-\mathcal{P}^b[k], r_b^c\right)\right\} \tag{4}$$

where we extend the definition of shortfall $\mathcal{P}^b[k]$ to the charging phases to denote the surplus energy. Hence, $me_b[i][i] = \min\left(md_b[i], mc_b[i][i]\right)$ is the amount of energy battery b may carry from CP_i to DP_i. It becomes more complex if the charging phase j and discharging phase i are not consecutive (*i.e.*, $i > j$). Because, 1) battery b has been used to carry energy from $CP_{j'}$ to $DP_{i'}$, for all i' and j' such that $j \leq j' \leq i' \leq i$, occupying a portion of its capacity, 2) as battery b has been used in CP_j before when exploiting $DP_{i'}, j \leq i' < i$, we need to deduct $u_b[k]$ from its charging rate r_b^c in each time slot $k \in CP_j$, resulting in:

$$c_b' = c_b - \max\left\{\max_{j \leq i' \leq i} \sum_{j'=1}^{i'} me_b[i'][j']\right\} \tag{5}$$

$$mc_b[i][j] = \min\left\{c_b', \sum_{k \in CP_j} \min\left(-\mathcal{P}^d[k], r_b^c - u_b[k]\right)\right\} \tag{6}$$

where $\sum_{j'=1}^{i'} me_b[i'][j']$ in Equation (5) is the total energy battery b has discharged in $DP_{i'}$. Although this energy has been depleted in $DP_{j'}$, only $c_b - \sum_{j'=1}^{i'} me_b[i'][j']$ capacity can be used to carry energy

from charging phases before j' to discharging phases after j'. To calculate the remaining capacity for CP_j and DP_i, all discharging phases between j and i have to be accounted, leading to the remaining capacity c'_b.

The above analysis is the foundation of our algorithm, called the Discharge-bounded Highest Efficiency First (DHEF). The pseudo code is shown in Algorithm 1. The loop from Line 3 to 18 iterates over all discharging phases, where $|DP|$ denoting the total number of discharging phases. For each discharging phase, the algorithm iterates over all batteries according to the descending order of efficiency to carry energy from charging phases to the discharging phase. After calculating $me_b[i][j]$, it calls a function, EALLOC, to allocate the energy into each time slot in charging phase j. It computes $u_b[k]$ such that shortfall is updated correctly, using Equation (2). If the i^{th} discharging phase cannot be satisfied, line 15 exits the execution with return value null. Otherwise, it proceeds to meet all discharging requirement and return the battery scheduling plan in line 19.

Algorithm 1 Discharge-Bounded Highest Efficiency First

Require: Power demand sequence $\mathcal{P}$, battery set $\mathcal{B}$
Ensure: Battery scheduling plan $\mathcal{U}$
1: **procedure** DHEF($\mathcal{P}$, $\mathcal{B}$)
2: $\mathcal{U} \leftarrow$ 2D array of size $|\mathcal{B}| \times |\mathcal{P}|$
3: **for** $i \leftarrow 1 \sim |DP|$ **do**
4: **for** $b \leftarrow 1 \sim |\mathcal{B}|$ **do**
5: $\Sigma \leftarrow 0$
6: calculate md_b[i]
7: **for** $j \leftarrow i \sim 1$ **do**
8: calculate mc_b[i][j]
9: me_b[i][j]$\leftarrow \min(md_b$[i] - Σ, mc_b[i][j])
10: $\Sigma \leftarrow \Sigma + me_b$[i][j]
11: $\mathcal{P}, \mathcal{U} \leftarrow$EALLOC($\mathcal{P}, CP_j, \mathcal{B}$[b], me_b[i][j], $\mathcal{U}$)
12: **end for**
13: $\mathcal{P}, \mathcal{U} \leftarrow$EALLOC($\mathcal{P}$, DP_i, $\mathcal{B}$[b], Σ, $\mathcal{U}$)
14: **end for**
15: **if** DP_i is not satisfied **then**
16: **return** null
17: **end if**
18: **end for**
19: **return** $\mathcal{U}$
20: **end procedure**

Algorithm Analysis: The first 2 level loops of DHEF enumerate over all discharging phases and all batteries which contribute $\mathcal{O}(T)$ and $\mathcal{O}(B)$ computational complexity in the worst case. For each discharging phase, DHEF checks all charging phases prior to it, which induces at most $\mathcal{O}(T)$ computational complexity. Hence, the worst case computational complexity is $\mathcal{O}(T^2B)$.

It remains to show how to compute $u_b[k]$ such that shortfall is updated correctly, using Equation (2), as we consider each battery. This is described below.

3.1.2 EALLOC: Allocating One Battery The problem solved in function EALLOC is the following: Given an initial amount of available energy, $x_b[k]$ in battery, b, and given a discharge phase, compute the energy allocation $u_b[k]$ for battery b for each slot k in the discharge phase.

To appreciate why some allocations are better than others, consider Figure 3. Assume that two batteries, b_1 and b_2, carry 8 and 6 units of energy respectively. Their discharge rates are 8 and 3. The batteries are discharged into two time slots, with a shorfall of 8 and 6 units of energy respectively. In Figure 3 (a), b_1 is used exclusively in time slot 1. Hence, it is depleted and cannot contribute to time slot 2. The discharge rate constraint of b_2 (namely, 3) falls short of supplying the needed power in slot 2 (namely, 6). Consequently, the schedule fails even though there is enough battery capacity to cover the shortfall. In contrast, Figure 3(b) is a solution where the shortfall is covered in all time slots. Specifically, in the first time slot, both batteries contribute (5 and 3 units of energy), leaving 3 units of energy in each battery. This is enough to cover the remaining shortfall in the second time slot.

The example demonstrates that one needs to be mindful of not only capacity but also the discharge constraints of batteries, as such constraints, if exceeded, will prevent covering the shortfall. Note that, these rate constraints apply independently in each time slot. Hence, given a set of undepleted batteries, the maximum shortfall slot determines the feasibility of meeting discharge rate constraints. If the shortfall in that slot is higher than the sum of the rate constraints, the allocation is infeasible.

The above observation suggests a simple solution to the problem of battery energy allocation; namely, allocate the energy across time slots such that the *maximum remaining shortfall is minimized*, hence maximally reducing the odds that rate constraints of remaining batteries will preclude filling the shortfall. The algorithm maintains a variable bar (the level to which to reduce the remaining shortfall). As bar is reduced, slots that reach the maximum discharge rate are "closed". Shortfall in the remaining slots continues to be reduced to the same level (bar) until all capacity of battery, b, is exhausted. The resulting allocation value, $u_b[k]$, is then returned for each slot, k, as as well as the updated shortfall, $\mathcal{P}^b[k]$.

3.2 Sequence Selection

Section 3.1 introduced algorithms to check feasibility under a given power demand sequence. Given that the power demand pattern is cyclic, one has to decide on a start time for the precending algorithm to be applied. Naively checking all possible start slots multiplies the computational complexity by another $\mathcal{O}(T)$ term, resulting in an$\mathcal{O}(T^3B)$ overall computational complexity, which is undesirable. Below, we describe how to describe a more efficient way of considering all possible start times.

The curve in Figure 2 shows a power demand pattern generated using Wikipedia's workload trace. The dashed horizontal line represents an attempting power cap P^c. As the power demand pattern is cyclic, fixing a starting instance is equivalent to selecting a sequence from the pattern. Let $\mathcal{D}_s^l = (P_s, P_{s+1}, ..., P_{s+l-1})$ denote the power demand sequence starting from time instance s with length l. As batteries are all empty in the very beginning, feasible sequence does not start with discharging time slots. We can also exclude time slot k if $k-1$ is a charging time slot. Because, if P^c is feasible for $\mathcal{D}_s^l$, the same set of batteries is also able to satisfy $\mathcal{D}_{s-1}^l$ under P^c. Therefore, remaining power sequence candidates start with intersection points of the power cap P^c and the power demand pattern. At last, we remove intersection points with positive derivatives on the power pattern curve, as no energy can be discharged from empty battery to meet the shortfall in discharging slot P_{s+1}. Hence, the feasibility checker only needs to try the intersection points whose derivatives are negative on the power demand curve, which are highlighted with circles in Figure 2.

Let $\mathcal{C}$ denote the power demand sequence candidate set. Although $|\mathcal{C}|$ is much smaller than $|P|$, it is still not efficient enough. As shown in Figure 2, a two day trace generates 6 candidates with the given power cap. A one week trace may result in several tens of intersections. Before diving into refinements, we first discuss the composability of feasibility. Suppose there are two power demand sequences, $\mathcal{D}_{s_1}^{l_1}$ and $\mathcal{D}_{s_2}^{l_2}$. Define the sequence composition operation $|$ as:

$$\mathcal{D}_{s_1}^{l_1}|\mathcal{D}_{s_2}^{l_2} = (P_{s_1}, ..., P_{s_1+l_1-1}, P_{s_2}, ..., P_{s_2+l_2-1}) \quad (7)$$

Please note that the operation $|$ is not commutative (*i.e.*, $\mathcal{D}_{s_1}^{l_1}|\mathcal{D}_{s_2}^{l_2} \neq \mathcal{D}_{s_2}^{l_2}|\mathcal{D}_{s_1}^{l_1}$). Given feasibility check results of $\mathcal{D}_{s_1}^{l_1}$ and $\mathcal{D}_{s_2}^{l_2}$ under the same power cap P^c, can we tell whether P^c is feasible for $\mathcal{D}_{s_1}^{l_1}|\mathcal{D}_{s_2}^{l_2}$? If P^c is infeasible on $\mathcal{D}_{s_1}^{l_1}$, neither will it be feasible for $\mathcal{D}_{s_1}^{l_1}|\mathcal{D}_{s_2}^{l_2}$, as the first l_1 time slots in $\mathcal{D}_{s_1}^{l_1}|\mathcal{D}_{s_2}^{l_2}$ violate P^c anyway. If P^c is feasible for both $\mathcal{D}_{s_1}^{l_1}$ and $\mathcal{D}_{s_2}^{l_2}$, it will also be feasible for $\mathcal{D}_{s_1}^{l_1}|\mathcal{D}_{s_2}^{l_2}$. Because the first l_1 time slots in $\mathcal{D}_{s_1}^{l_1}|\mathcal{D}_{s_2}^{l_2}$ experience exactly the same situation as $\mathcal{D}_{s_1}^{l_1}$, and the last l_2 time slots in $\mathcal{D}_{s_1}^{l_1}|\mathcal{D}_{s_2}^{l_2}$ are no worse than $\mathcal{D}_{s_2}^{l_2}$ as it may or may not enjoy some leftover energy in batteries from the first l_1 time slots. For the last combination where P^c is feasible for $\mathcal{D}_{s_1}^{l_1}$ and infeasible for $\mathcal{D}_{s_2}^{l_2}$, we cannot tell its feasibility without invoking the feasibility checking algorithm (DHEF). The reason is that there might be some leftover energy after the first l_1 time slots which may help to meet shortfalls in the last l_2 time slots. Table 1 summaries the results, which we call the feasibility composition law.

The candidate set $\mathcal{C}$ naturally divides the cyclic power demand pattern into an array of smaller pieces. Please note, the array is still cyclic, as we do not know the optimal starting time slot yet. Let 1 denote feasible, and 0 denote infeasible. With a given power cap P^c, the feasibility checker tests each piece separately and generates a cyclic 0/1 series. According to feasibility composition law, compose two feasible or two infeasible pieces does not change the feasibility. Hence, we merge consecutive 0s into a single 0, and merge consecutive 1s into a single 1. Now, we have a new series with alternating 0 and 1. Again, based on the feasibility composition law, compose infeasible piece in front of feasible piece results in a larger infeasible piece, and the only unknown combination is (1, 0).

$\mathcal{D}_{s_1}^{l_1}$	feasible	feasible	infeasible	infeasible
$\mathcal{D}_{s_2}^{l_2}$	feasible	infeasible	feasible	infeasible
$\mathcal{D}_{s_1}^{l_1}\|\mathcal{D}_{s_2}^{l_2}$	feasible	**unknown**	infeasible	infeasible

Table 1: Feasibility Composition Law

As the cost of evaluating a (1, 0) piece grows quadratically with its length, it is more efficient to avoid long (1, 0) pieces when possible. Therefore, we propose a greedy algorithm that only reduces the smallest candidate power demand piece in each iteration instead of reducing all of them. An example is shown in Figure 4.

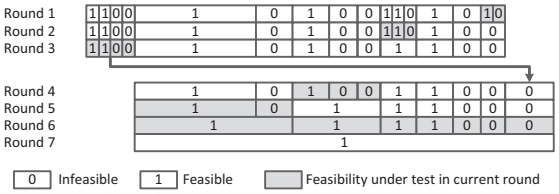


Figure 4: Accelerated Feasibility Check Example

Suppose we have n_k candidates in round k. Each round, the algorithm checks the (1, 0) subsequence with the smallest number of time slots. If the answer is feasible, this candidate merges with its right-hand side candidate, as all candidate subsequences start with a feasible piece. Otherwise, it merges with its left-hand side candidate, as all candidate subsequences end with a infeasible piece. Hence, we have $|\mathcal{C}| > n_k > n_{k+1} > 1$. Let W denote the length of the entire power demand pattern. Then, the length of the smallest candidate piece in round k is shorter than $\frac{W}{n_k}$. According to the computational complexity of the DHEF algorithm, it takes at most $\frac{1}{n_k^2}\mathcal{O}(T^2B)$ to check the feasibility in k^{th} round. Therefore, the worst case computational complexity is:

$$\mathcal{O}(T^2B)\sum_k \frac{1}{n_k^2} \leq \mathcal{O}(T^2B)\sum_{i=1}^{|\mathcal{C}|}\frac{1}{i^2} \leq \frac{\pi^2\mathcal{O}(T^2B)}{6} = \mathcal{O}(T^2B).$$

4 Evaluation

This section evaluates how WattValet compares to the optimal solution as well as state-of-the-art solutions in terms of approximating Optimality and handling Heterogeneity.

Battery configurations in our experiments are based on APC 3U UPS [1] devices. The designed UPS capacity is

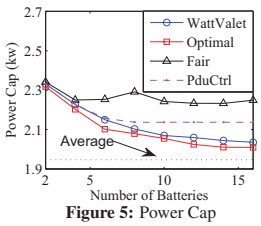


Figure 5: Power Cap

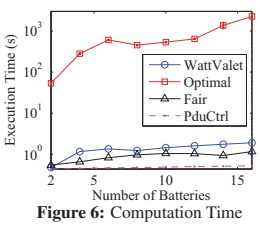


Figure 6: Computation Time

0.5 $KW \cdot H$, and its maximum charging and discharging rate are 240W and 2100W respectively. The efficiency is about 85% when serving more than 25% load. Aged batteries suffer degradations in their performance. We apply random degradations on batteries to generate a synthetic battery set. Each battery parameter is scaled with a random factor γ:

$$\gamma = random() \times \theta + 1 - \theta, \tag{8}$$

where $random()$ returns a random fractional number between 0 and 1 whenever called, and θ is an input parameter that controls the randomness bound. Wikipedia hosts its service on about 300 servers [13]. Wikipedia's workload is translated into power consumption traces using the linear power model of our Dell D620 servers [11, 12].

WattValet trades optimality for efficiency. However, we need to make sure it stays within a reasonable range from the optimal solution. Evaluations compare WattValet to the optimal solution as well as two heuristics from recent literatures[2, 10]. The *fair* battery scheduling algorithm [2] deducts equal current from all batteries, whereas the *pduCtrl* algorithm [10] charges/discharges batteries one after another. Both solutions examine time slots chronologically. As the LP solver takes excessively long to converge, optimal solutions are only calculated for small scale evaluations. We scale down wikipedia's workload to fit into 30 servers, and use at most 16 UPS devices. The other three-week trace in Figure 1 is used to generate the predicted power demand pattern by taking average in each time slot. The LP problem is solved with Matlab using *linprog* method. As plotted in Figure 5, the power cap achieved by WattValet is only 2% higher than the optimal solution in the worst case, while the other two heuristics lead to 7% and 12% degradations respectively. With the *fair* scheduler, the power cap increases when using 8 UPS devices. It is because the 8^{th} battery suffers much lower efficiency compared to other batteries. The computation times of the four solutions are shown in Figure 6. The LP solver takes 1600 times longer than WattValet when using 16 batteries. The average power demand is plotted with the dotted line. There is still a gap between the settled optimal solution and average. The reason is that batteries' efficiency is at most 85%. Some energy losses when carried from charging phases to discharging phases.

5 Related Work

Recently, shaving off data center power peaks using energy storage devices was introduced by Govindan *et al.* [7]. They gave a complete overview of the datacenter power hierarchy, and implemented a simple heuristic to reduce datacenter operational expenses. Later, Kontorinis *et al.* [10] adapted this idea to a Google's datacenter, where each server is equipped with its own dedicated battery. Wang *et al.* [16] further investigated and evaluated broader types of energy storage methods, and compared their advantages and limitations. Other literature [14] explicitly achieved minimum power capping by modeling the problem as one of linear programming. However, all of above work considers only homogeneous batteries, or environments with very limited heterogeneity. In real-world systems, even if all batteries are identical initially, different storage temperature, humidity, and charging/discharging cycles will cause them to diverge over time.

This paper shares a similar infrastructure setup to Aksanli *et al.* [2]. Together with the utility power, a pool of batteries provide centralized support to the entire computing side. The major difference is that Aksanli [2] aims at maintaining homogeneity of all batteries which may not be possible in real world systems, whereas WattValet takes explicit advantage of battery heterogeneity.

Finally, the paper suggests that energy storage allocation in data centers is a QoS mechanism of growing importance at an age where sustainability of computation becomes a dimension of quality. The minimum power cap is presented as a metric of projected increasing interest. Solutions that lead to smaller power caps are more sustainable, because smaller power caps are indicative of smaller power consumption and better alignment between maximum and average power, achieved via more judicious energy storage management. The paper addresses the latter subproblem, where the cap is minimized for a given power demand profile.

6 Conclusion

This paper presents WattValet that reduces datacenter peak power consumption by using batteries. WattValet explicitly considers heterogeneities between batteries when generating the battery charging and discharging plan. It breaks down the power capping problem into two smaller parts, namely, generating battery charging/discharging plans for a given power demand sequence, and searching for the power demand sequence that leads to the minimum power cap. The efficiency of WattValet allows it to scale to datacenter-size problems, whereas the power capping result achieved by WattValet on Wikipedia's data is within 2% of the optimal solution. WattValet considerably outperforms state-of-the-art solution, and the advantage increases as the heterogeneity grows.

References

[1] Apc smart-ups on-line model SURTA3000XLTW. http://www.apc.com, Jan 2014.

[2] B. Aksanli, E. Pettis, and T. Rosing. Architecting efficient peak power shaving using batteries in data centers. MASCOTS, 2013.

[3] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel. Finding a needle in haystack: facebook's photo storage. USENIX OSDI, 2010.

[4] A. A. Bhattacharya, D. Culler, A. Kansal, S. Govindan, and S. Sankar. The need for speed and stability in data center power capping. IGCC, June 2012.

[5] Gartner. Gartner says data center power, cooling and space issues are set to increase rapidly as a result of new high-density infrastructure deployments, May 2010. http://www.gartner.com/it/page.jsp?id=1368614.

[6] D. Gmach, J. Rolia, C. Bash, Y. Chen, T. Christian, A. Shah, R. Sharma, and Z. Wang. Capacity planning and power management to exploit sustainable energy. CNSM, October 2010.

[7] S. Govindan, A. Sivasubramaniam, and B. Urgaonkar. Benefits and limitations of tapping into stored energy for datacenters. ACM/IEEE ISCA, 2011.

[8] S. Govindan, D. Wang, A. Sivasubramaniam, and B. Urgaonkar. Leveraging stored energy for handling power emergencies in aggressively provisioned datacenters. ACM ASPLOS, 2012.

[9] J. Hamilton. Cost of power in large-scale data centers, November 2008. http://perspectives.mvdirona.com/2010/09/18/OverallDataCenterCosts.aspx.

[10] V. Kontorinis, L. E. Zhang, B. Aksanli, J. Sampson, H. Homayoun, E. Pettis, D. M. Tullsen, and T. S. Rosing. Managing distributed ups energy for effective power capping in data centers. ISCA, 2012.

[11] S. Li, T. Abdelzaher, and M. Yuan. TAPA: Temperature aware power allocation in data center with map-reduce. In *IEEE*, IGCC, 2011.

[12] S. Li, H. Le, N. Pham, J. Heo, and T. Abdelzaher. Joint optimization of computing and cooling energy: Analytic model and a machine room case study. In *IEEE ICDCS*, 2012.

[13] R. Miller. Google gift means more servers for wikipedia. http://www.datacenterknowledge.com/, Jan 2014.

[14] D. S. Palasamudram, R. K. Sitaraman, B. Urgaonkar, and R. Urgaonkar. Using batteries to reduce the power costs of internet-scale distributed networks. ACM SoCC, 2012.

[15] G. Urdaneta, G. Pierre, and M. van Steen. Wikipedia workload analysis for decentralized hosting. *Elsevier Computer Networks*, 53(11):1830–1845, July 2009. http://www.globule.org/publi/WWADH_comnet2009.html.

[16] D. Wang, C. Ren, A. Sivasubramaniam, B. Urgaonkar, and H. Fathy. Energy storage in datacenters: what, where, and how much? ACM SIGMETRICS, 2012.

[17] I. Widjaja, A. Walid, Y. Luo, Y. Xu, and H. J. Chao. Small versus large: Switch sizing in topology design of energy-efficient data centers. IEEE IWQoS, June 2013.

[18] D. X, X. Liu, and B. Fan. Minimizing energy cost for internet-scale datacenters with dynamic traffic. IEEE IWQoS, June 2011.