Guaranteeing Performance SLAs of Cloud Applications under Resource Storms

In Kee Kim, *Member, IEEE,* Jinho Hwang, *Member, IEEE,* Wei Wang, *Member, IEEE,* and Marty Humphrey, *Member, IEEE*

Abstract—In modern data centers, enterprise cloud instances run not only foreground applications like web and databases, but also different background services (e.g., backup, virus/compliance scan, batch) to manage the cloud instances securely and improve the overall resource utilization. These background services often incur resource storms that suddenly consume a lot of shared resources on cloud instances. The resource storms significantly degrade the performance of foreground applications by interfering in the preemption of the shared resources, resulting in frequent SLA violations. However, stock OS schedulers are not designed to handle these situations, and prior works are insufficient to address such resource storms under highly dynamic cloud workloads. This work presents Orchestra, a cloud-specific framework for controlling multiple applications in the user space, aiming at meeting corresponding SLAs. Orchestra takes an online approach with lightweight monitoring and performance models for both applications on the fly. It optimizes the resource allocations to meet corresponding SLAs. We evaluate the performance of Orchestra on a production cloud with a diverse range of SLAs. Orchestra guarantees the foreground application's performance SLAs at all times. At the same time, Orchestra maintains the background's performance by minimizing its performance penalty with proper allocation of the shared resources.

Index Terms—Cloud Computing; Resource Storms; Resource and Application Management, Guaranteeing Performance SLA; Performance Modeling and Prediction, Enterprise Cloud Management

1 Introduction

In modern data centers, enterprise cloud instances (i.e., virtual machines) are not only serving user-facing (foreground or FG) applications, but also running diverse types of background (BG) services[1] – *backup*, *security compliance*, *virus scan*, *patching*, and *batch* tasks – in order to securely and reliably manage such instances, and improve overall resource utilization/cost efficiency. Since the BG services frequently perform very critical missions for management purposes, they have to be executed as planned in many cases [1]. This requirement incurs *resource storms* [2] that create high peaks of resource usage and surges of resource contention without being aware of the FGs' resource requirements [3–9]. Such resource storms can retard the processing time of FG applications and in turn the response time.

Figure 1(a) illustrates the performance degradation of two FGs (Olio Web application and MongoDB) when they co-run with BGs. To confirm the performance change in FGs, we use three different BGs, which are a backup, virus scanner, and batch (MapReduce) applications. The result shows that the web application's tail latency (98%tile) could be as slow as 24x with a BG application (backup), 33x with two BGs (backup and virus scanner), and 36x with three BGs (backup, virus scanner, and batch application). We also observe that MongoDB also shows a similar trend of performance degradation when co-running with different numbers of BGs. The degraded performance has a significant impact on the QoS of the FGs, resulting in frequent SLA violations and poor user experiences. For instance, Amazon has reported that every 100ms delay loses 1% of the sales profit [10], and video streaming (e.g., YouTube) users start abandoning videos after 2 seconds of buffering time [11].

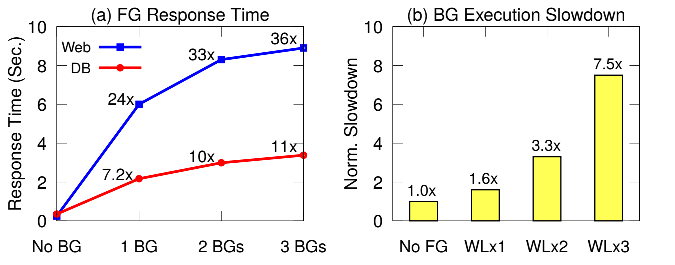


Fig. 1. Performance variation of FG and BG applications. (a) the slowdown of FGs (Web and DB) response time (98%tile) when running together with BGs; (b) the slowdown of a 10G data backup duration as the FG workloads increase.

However, the current cloud instances are not well designed to handle the resource storms. Specifically, stock operating system schedulers such as completely or weighted fair scheduler (CPU, IO) and network queueing (FIFO) mechanisms are designed without considering the resource storms [12, 13], so that SLAs of FGs suffer while parts of shared resources are consumed by BGs [14, 15]. OS

- I. K. Kim is with the Department of Computer Science, University of Georgia, Athens, GA 30602. E-mail: inkee.kim@uga.edu
- J. Hwang is with the IBM T. J. Watson Research Center, Yorktown Heights, NY 10598. E-mail: jinho@us.ibm.com
- W. Wang is with the Department of Computer Science, University of Texas at San Antonio, San Antonio, TX 78249. E-mail: wei.wang@utsa.edu
- M. Humphrey is with the Department of Computer Science, University of Virginia, Charlottesville, VA 22904. E-mail: humphrey@cs.virginia.edu

1. In this paper, we use the term "application," "task," and "service" *interchangeably.*

modifications, such as changing task priority [16], designing a biased OS scheduler [17], have been proposed, but such tweaks are not feasible for ordinary cloud users due to the technical difficulties. Moreover, intuitive approaches (e.g., terminating or suspending BGs [18, 19]) to guarantee the FGs' SLA are not sufficient in practice since particular BG tasks, such as backup and security checks, may have SLAs[2] to finish the tasks due to the importance of such services [1]. As shown in Figure 1(b), the BGs' execution time is also highly affected by the amount of FG workloads. Such coarse-grained approaches – *minimizing* the resource allocation – are hard to guarantee the BG's SLAs (or completion of the tasks) and often underutilize the cloud instances by overly controlling the BGs.

The research community has performed significant work, especially when one or more FG applications – normally *latency-sensitive* applications – are running together with other BGs such as batch jobs [18]. Previous approaches mainly focus on enhancing performance isolation [12, 20–22], designing intelligent scheduling policies [5, 23–25], or determining safe co-locations [3, 4, 26]. These techniques often rely on either or both of monitoring the host machine's system/HW-level statistics (e.g., program counter and cache miss rate) and profiling the behaviors of FGs and BGs. While AWS recently started to provide PMU (Performance Monitoring Unit) capability to the dedicated instance users [27], this information is not yet accessible by the users of more general resource provisioning models like on-demand and spot instances. In general, only cloud providers (e.g., AWS, Azure, or data center operators) are allowed to leverage such information [28, 29], so we do not consider using such information in this work. The profiling-based approaches aim to create performance interference models through off/online measurements. However, the models do not provide the flexibility required in highly dynamic cloud environments and workloads [30, 31]. The models need profiling of target applications with different/diverse constraints – virtual resources (vCPU, memory, network, disk) and even HW architectures –, and different combinations of placement with other types of applications. It is apparent to imagine how much profiling effort needs with all possible combinations. As a result, the models created by the off/online measurements are more desirable in practice, yet the required computation and monitoring power are very challenging.

In this paper, we address the impact of the resource storms by creating Orchestra, a framework for controlling both FG applications and BG[3] services in the user space, aiming at meeting both SLAs. Orchestra relies on an online approach with very lightweight monitoring at runtime. With the monitoring, Orchestra estimates the response time of FGs using a multivariate polynomial model with a wide range of resource options and predicts a BG's execution time from a multivariate linear regression powered by its resource usage and application-assisted hints. It then optimizes the allocations of diverse resources on cloud instances to both FG and BGs for guaranteeing their SLAs. The resource control by Orchestra leverages the knobs provided by modern OS's improvement, such as cgroups[4]. Orchestra is complementary to widely used approaches for cloud application management. Orchestra's capabilities for performance monitoring and resource controlling offer finer-grained mechanisms than off-the-shelf monitoring/management (e.g., cloud auto-scaling[5] and CloudWatch[6]), and thus it helps cloud users accurately determine *when* to scale.

We have implemented and evaluated Orchestra with real workloads on the production clouds. Our primary workloads are a web service and a NoSQL database (MongoDB) for FG applications, and backup (AWS Sync[7]) and virus/malware scanner (ClamAV[8]) for BG services. Our evaluation shows that Orchestra can comply with various SLA targets for FG applications with 70% performance improvement of the BG services. Moreover, Orchestra has a very high overall correctness (less than 5% error), 16.5% of MAPE (Mean Absolute Percentage Error) for the FGs' response time estimation, and over 90% accuracy for the BGs' performance prediction.

As a result, Orchestra has the following contributions:

- A real trace of performance and resource usage fluctuations of two BG applications measured on real IBM production servers to demonstrate the severe impact of resource storms (Section 2).
- A user space resource control framework that provides low-overhead resource management to ensure high SLA satisfaction for multiple co-running applications (both FG and BG) on cloud instances (Section 3.1, 3.4, and 4).
- An accurate, multivariant regression-based response time estimator for various FG applications. In particular, we perform extensive characterization of the behavior of two FG applications (e.g., web and MongoDB) as well as evaluation of overhead and accuracy from the prediction model with various parameters (Section 3.2).
- Design of an accurate performance model for BG services. Specifically, we provide a detailed correlation of the execution of BG services with diverse resource factors (e.g., CPU, memory, IO, disk) as well as application-specific parameters (Section 3.3).
- A thorough evaluation of the performance of Orchestra on the production clouds with real FG and BG workloads. We evaluate the overall performance of orchestra regarding SLA satisfaction, the accuracy of performance prediction models for both FG and BGs, as well as the overhead of the framework. In particular, we employ a more realistic application deployment scenarios in public cloud environments (e.g., FG co-running with multiple BG services) to

2. These SLAs are often very relaxed as compared to the SLAs for the FGs. i.e., once in a day.
3. In this work, we consider that BG is running on Virtual Machines (VMs) and managed by public cloud users. The users have control over both FG applications and BG services. Thus the users can obtain application-level and system-level statistics regarding the execution of FGs and BGs.

4. https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt
5. https://aws.amazon.com/autoscaling/
6. https://aws.amazon.com/cloudwatch/
7. http://docs.aws.amazon.com/cli/latest/reference/s3/sync.html
8. https://www.clamav.net/

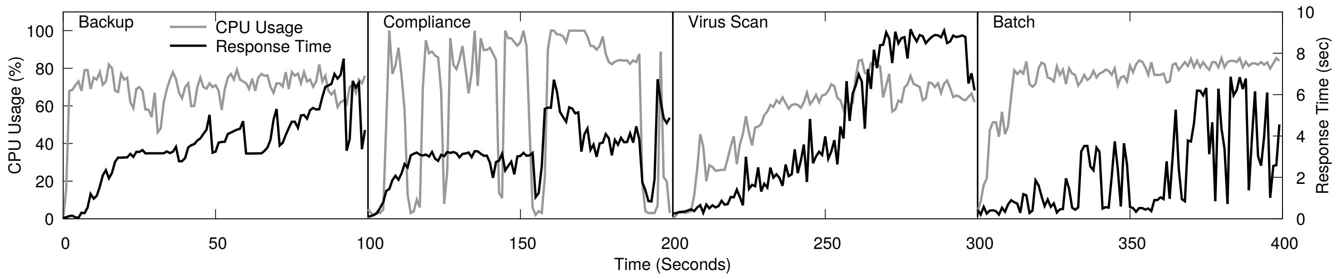


Fig. 2. Resource and performance metrics of a managed cloud instance. Two BGs (Backup and Virus Scan) are measured on VMs in AWS. The other two BGs (Compliance and Batch) are measured on production IBM servers. The four measurements on real cloud VMs depict the fluctuation of response time in FG application regarding the change of co-running BG services' resource consumption.

compare the performance of Orchestra against state-of-the-art approaches (Section 5).

A preliminary version of this work appeared in ISPDC 2018 conference [2], but the first version only focused on simplified deployment scenarios of cloud applications. For example, the previous version only considered deployment of a FG application (e.g., only hosting Web or MongoDB) and a BG service (e.g., only executing AWS Sync or ClamAV) on the same VM. In this version, we take a step further in presenting an improved version of Orchestra with adopting a more realistic deployment scenario of cloud applications by executing and controlling multiple BG services co-running with a FG application. We further provide a thorough investigation of the resource storms and the resource management solution under them. Specifically, in this paper, we provide an in-depth analysis of the resources storms inside a production data center (IBM) and public clouds (Amazon EC2). Moreover, we perform a more comprehensive analysis of the behavior of both FG applications and BG services and identify the most critical factors that can change the execution of such applications and services. We also provide a comprehensive evaluation of Orchestra to study its performance in terms of accuracy and overhead. These improvements aim at obtaining a complete understanding of resource storms and their impacts in real clouds, as well as a better understanding of the effectiveness of our solution.

We structure the rest of the paper as follows. In Section 2, we describe the background of this work. We present the framework details and implementation of Orchestra in Section 3 and 4. In Section 5, we evaluate the performance of Orchestra with real-world cloud workloads. Section 6 describes discussion and future work of Orchestra; In Section 7, we summarize related work. Finally, Section 8 concludes this paper.

2 Background

2.1 Enterprise Cloud Instances

Enterprise or managed cloud instances require high standards for infrastructure service management such as backup, application/system monitoring, compliance, and patching. The services are typically built by a diverse set of providers. Service vendors often have agent processes running in the background, and they run commands from a central manager or report data to the central location. Due to such behaviors of the BG services, there has been significant research focused on resource contention, especially batch jobs. Although the research community has been mainly focusing on the cloud resource contention problem among VMs as to resource over-provisioning, in practice, it is unlikely as cloud providers have reported that the average utilization of cloud instances is about 20% – 30% [31–35]. Therefore, over-provisioning resources do not solve the resource contention for cloud applications.

However, each managed cloud instance often runs 5 – 10 BG service agents, and their operations are not coordinated as they are from different vendors [36]. Thus, it is highly likely that agents run simultaneously and saturate resources abruptly without being aware of FGs' resource requirements and in turn, result in *resource storms* [2]. In each cloud instance, resource storms generated from BG management services have a significant impact on the performance of FG applications. Figure 2 demonstrates the fluctuation of the response time of a FG application (webserver) on a managed cloud when co-running with multiple management and BG services such as backup, compliance check, virus scan, and batch jobs. The impacts from the compliance and batch are measured on the production IBM servers, and the other BGs (backup and virus scan) are measured on VMs from Amazon EC2. As shown in Figure 2, the FG's varies significantly as the BGs preempt more shared resources (e.g., CPU) on the same cloud instance. Therefore, it is apparent to imagine that simultaneous executions of BG services could substantially worsen the FGs performance.

2.2 User Space Resource Control

The resource controls in user space for each resource type has been developed separately thus far. A CPU control, *nice* directly maps to a kernel system call to manually adjust the task's priority level of a process, and yet another CPU resource control, *cpulimit*[9] repeatedly curbs the CPU usage of a process by stopping the process at different intervals to keep it under the defined ceiling. A disk resource control, *ionice* gets and sets program IO scheduling class and priority to adjust the disk usage of processes. There are network traffic shaping tools such as *trickle* [37], *force_bind*[10], and *damper*[11] that throttle the traffic bandwidth of processes.

9. http://cpulimit.sourceforge.net/
10. http://kernel.embedromix.ro/us/
11. https://github.com/vmxdev/damper

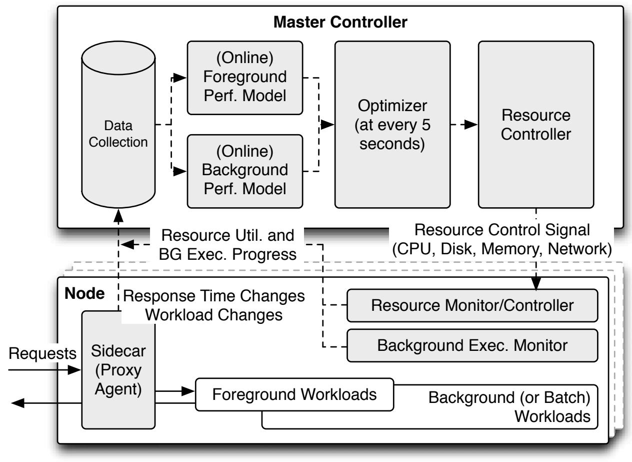

Master Controller
Data Collection
(Online) Foreground Perf. Model
(Online) Background Perf. Model
Optimizer (at every 5 seconds)
Resource Controller
Resource Util. and BG Exec. Progress
Resource Control Signal (CPU, Disk, Memory, Network)
Node
Response Time Changes Workload Changes
Resource Monitor/Controller
Background Exec. Monitor
Requests
Sidecar (Proxy Agent)
Foreground Workloads
Background (or Batch) Workloads

Fig. 3. Overall architecture of Orchestra.

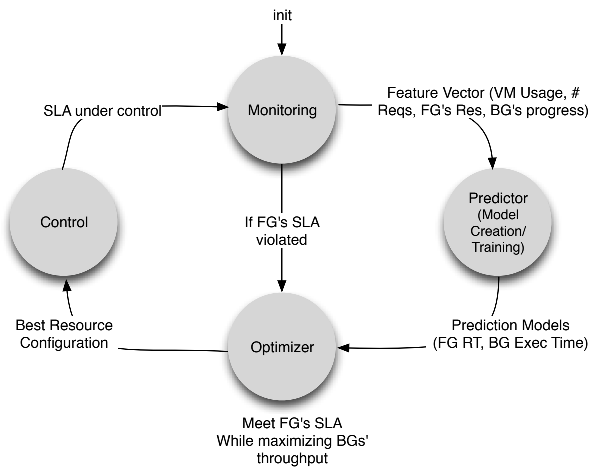

init
SLA under control
Monitoring
Feature Vector (VM Usage, # Reqs, FG's Res, BG's progress)
Control
If FG's SLA violated
Predictor (Model Creation/ Training)
Best Resource Configuration
Optimizer
Prediction Models (FG RT, BG Exec Time)
Meet FG's SLA While maximizing BGs' throughput

Fig. 4. Orchestra state diagram.

With the emergence of cloud computing, there have been demands for the user space resource control mechanisms. In particular, the control groups (cgroups) have gained large attention recently under container virtualization dominance. The Linux's built-in cgroups is a mechanism that lets OS schedulers to limit the amount of resources (e.g., CPU time, system memory, disk IO, network bandwidth, or combinations of these resources) available to processes from user space, and this allows users to specify how the kernel should allocate specific resources to a group of processes. Specifically, reconfiguring resources dynamically on a running system is amenable to user space resource control with fine-grained control over allocating, prioritizing, denying, managing, and monitoring system resources. The advantage of cgroups over prior implementations is that the limits are applied to a set of processes, rather than to just one. Orchestra adopts the user space resource control mechanisms.

3 Orchestra FRAMEWORK

3.1 Orchestra Overview

3.1.1 Overall Architecture

We design Orchestra with a two-layer, distributed architecture, of *managed nodes* and a *master controller*. Figure 3 illustrates the design of Orchestra.

A **managed node** (VM instance) has two components in Orchestra– *sidecar* and *node agent*. The sidecar – *a traffic forwarder used as an online performance monitor* – is designed to monitor performance variation of the FGs, i.e., a response time of web requests and DB transactions. It measures the FG's processing time by capturing the ingress and egress time of user requests. Moreover, the sidecar can monitor a diverse set of FG workloads as long as they use general purpose protocols (e.g., HTTP, TCP) to communicate with end-users. The measured FG's performance is reported to the data collector in the master controller.

The node agents are used for 1) monitoring the resource usage of the target applications, 2) monitoring the progress of BG's execution, and 3) reconfiguring resources allocation to both FG and BGs. The monitoring of resource usage focuses on collecting general system statistics, i.e., vCPU, memory, disk, and network IO. The BG's execution progress is relying on probing the application-assisted hints, such as retrieving log files. All the collected statistics – resource utilization and application progress – are reported to the data collector in the master controller. The resource reconfiguration is to manage subsystems of control knobs (e.g., cgroups) with the decision made by the master controller.

The **master controller** plays the most important role in Orchestra by determining the adjusted resource allocations to both FG and BGs with the goal of satisfying FG's SLA requirement and maximize BG's executions. To this end, with various statistics – response time, resource utilization, and application progress – from the node agents, the master controller creates a *response time estimator* (Section 3.2) and *performance model* (Section 3.3) for both applications on the fly. With these models, the master controller optimizes the resource allocations to achieve the management goal. The detailed mechanisms will be explained in Section 3.4.

3.1.2 Orchestra Workflow

The workflow of Orchestra is illustrated in Figure 4. Orchestra starts with monitoring diverse statistics of the FG and BG applications' performance and the instance's resource utilization. And it creates feature vectors with the statistics, and these feature vectors are used to train and create two predictive models that forecast the FG's response time and BG's execution duration. If a response time of the FG application is close to or violates a SLA target (defined by Orchestra operator), Orchestra adjusts the resource allocations to both applications through optimization with two predictive models. Once the proper resource allocation is determined, the decision is sent to a node agent that will change resource usage to both applications. If the performance of FG becomes stable (meeting the SLA targets), then Orchestra stops controlling the resource allocation and comes back to the steady monitoring state that collects the essential statistics to tune two predictive models.

3.2 Response Time Estimator for FG Applications

A key component of the master controller is the RT (Response Time) estimator that predicts (the near future) web

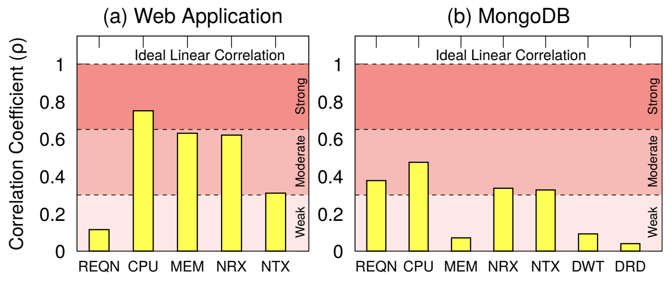


Fig. 5. Correlation coefficient of factors that could affect FG's response time. (NRX: Network RX Bytes/sec. NTX: Network TX Bytes/sec., DWT: Disk Write Bytes/sec., DRD: Disk Read Byte/sec.)

response time or DB transaction time with a broad range of resource utilization. Since Orchestra's decision on the resource control relies upon this estimation model and the resource control should be made at runtime, high accuracy with low overhead is an essential prerequisite.

3.2.1 Feature Selection for FG Applications

We observe the behaviors of two FGs (Web and MongoDB) by running benchmark tools (CloudSuite [38] and TPC-C[12] without BGs' execution. We then calculate the Pearson Correlation Coefficient between the FG's RT and the following features, including the number of requests and various system resources – CPU, memory, disk, and network IOs.

Figure 5 reports the measured correlation. Three factors show the highest correlation with the RT of web application (FG) – CPU, MEM (Memory), and NRX (Network RX Bytes). The coefficients are between 0.6 and 0.75. Note that we do not measure disk IO for the web application since it has very negligible disk operations. In the MongoDB benchmark, all features show relatively weaker correlations. Four factors – REQN (the request numbers/sec.), CPU, NRX, and NTX (Network TX Bytes) – show a moderate correlation with the MongoDB's RT. The coefficients are slightly over 0.3. While the MongoDB has weaker factors, we decide to consider all these (correlated) factors to model the RT estimator because the RT estimator aims to handle both or potentially more types of FGs. The selected factors are CPU, MEM, NRX, and NTX. We exclude REQN from this feature selection since NRX is a more comprehensive metric that covers REQN. For the other FGs, users may add other factors if necessary.

Additionally, we measure the correlation coefficient among these four factors since there could be a certain possibility that one factor can be correlated with other ones. i.e., a correlation between CPU and Network IO. We report the correlations among the factors represented by 1 to 3 of scale (1: weak, 2: moderate, and 3: strong correlation), and the results are reported in Table 1. CPU and Network IO are strongly correlated with each other for both FGs. MEM is moderately (Web) or lightly (MongoDB) correlated with the other two factors. MEM also shows low variance over the RT's fluctuation. i.e., (μ of 13, σ of 11) for Web, (μ of 23, σ of 2) for MongoDB.

12. https://github.com/apavlo/py-tpcc

TABLE 1
Correlation between the selected factors. (1: weak, 2: moderate, 3: strong correlation)

(a) Web Application

	CPU	MEM	NRX	NTX
CPU	–	2	3	3
MEM	2	–	2	1
NRX	3	2	–	3
NTX	3	1	3	–

(b) MongoDB

	CPU	MEM	NRX	NTX
CPU	–	1	3	3
MEM	1	–	1	1
NRX	3	1	–	3
NTX	3	1	3	–

3.2.2 Model Selection for FG Applications

We chose MVPR (Multivariate Polynomial Regression) [39] approach to model the RT estimator because MVPR considers both 1) multiple factors' contribution to the estimation target and 2) the correlation among the selected factors. The MVPR model is expressed as below:

$$f(x_1, x_2, ..., x_p) = \sum_{i=0}^{N} \beta_i \phi_i \tag{1}$$

where p indicates the number of independent variables, β_i is coefficient, $\phi_1 = 1$, $\phi_N = x_1^n \cdot x_2^n \cdot x_3^n \cdots x_p^n$, and n is the order of the MVPR.

When applying a polynomial model to a runtime system, the computational overhead is highly concerned. The overhead depends on both the number of the independent variables (p) and the order (n) of the model. With this concern, we use a harmonic mean of NTX and NRX ($\frac{2}{\frac{1}{NTX}+\frac{1}{NRX}}$), both representing network-IO statistics, and it helps to reduce the number of equation terms. i.e., with a quadratic model, 3 independent variables require 27 terms, and 4 variables generate 64 terms. Regarding the order of the model, we empirically test the overhead with a training dataset (1 hour of Web RT data) and various polynomial orders from two to ten. The overhead exponentially increases as the order of the polynomial model increases as shown in Figure 6. However, the overhead is not too high in the model with the orders less than 5. For example, the quartic model (order of 4) has an average computational overhead of 7.8ms and, its highest overhead of prediction is just as high as 87ms. We limit the order of the model with this observation. Moreover, the overhead of MVPR can be determined by the size of the training dataset. The RT estimation model with MVPR incorporates a sliding window technique [40] to limit the size of the training dataset as well as leverage the most recent observations.

3.2.3 Model Accuracy of FG Response Time Estimator

We measure the performance of the RT estimation model with Web and MongoDB. Figure 7 illustrates the RT (ground truth) from both FGs and prediction from the model with an order of 2. While this evaluation is performed with less

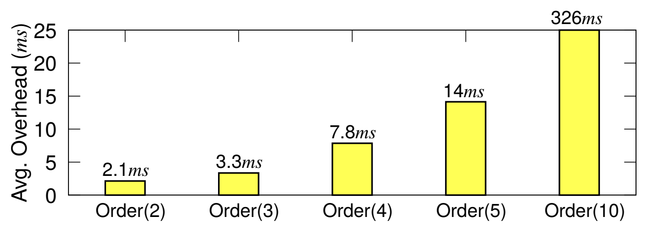


Fig. 6. Computational overhead of MVPR model with different polynomial orders.

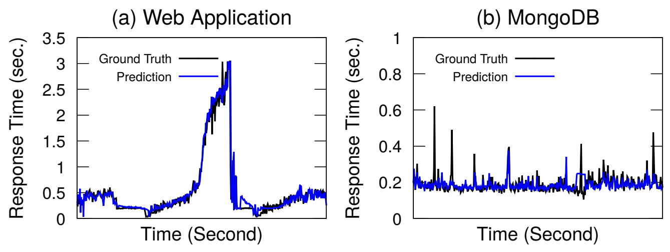


Fig. 7. Prediction results from MVPR model (order of 2) for RTs of Web and MongoDB.

challenging conditions (without BGs), it is obvious that the model accurately estimates the RT of the FGs. The prediction shape from the model successfully catches the trend of the RT variation for the web application. For the MongoDB, the model has more errors than the case of the web application, but it also shows robust predictions except for some outliers. We perform more comprehensive evaluations for the accuracy of the RT estimation model with the BGs' executions in the evaluation section.

3.3 Performance Model for BG Services

Orchestra requires a performance model that predicts BGs' execution time. The model is essential for monitoring and controlling BGs services because Orchestra needs to assure BGs' SLA satisfaction and/or minimizing their execution time. So this model performs a critical role in optimizing resource allocation with an accurate prediction of difference resource usages. To create such a model, we consider ClamAV[13] and AWS Sync[14] as examples of BGs.

3.3.1 Feature Selection for BG Performance Model

We perform a profiling study on two different Amazon EC2 instances[15] – m3.medium and c4.large – of Ubuntu 16.04 LTS with 35G (10K files) and 106G (50K files) dataset. In this measurement, we use the default configuration of Ubuntu OS and run these two BG services individually without any FGs' execution. The statistics and results from this profiling are shown in Table 2. ClamAV is observed as a CPU and Disk-IO (Read) bound application and moderately consumes memory resources. Sync mostly consumes CPU,

TABLE 2
Statistics of measurement results for understanding BG applications' characteristics on two EC2 instances.

	(a) ClamAV		**(B) Sync**	
Dataset	**35G**	**106G**	**35G**	**106G**
CPU[16]	73.8%	86.5%	78.6%	95.2%
MEM	14.2% (0.53G)	19.5% (0.73G)	3.3% (0.12G)	3.9% (0.15G)
DRD	8.3 MB/s	40 MB/s	51 MB/s	133 MB/s
DWT	79 KB/s	301 KB/s	2.2 KB/s	159 KB/s
NTX	–		55 MB/s	141 MB/s
NRX	–		1.4 MB/s	3.7 MB/s

Disk-IO (Read), and Network (TX) resources. Since both CPU and Disk-IO (Read) are common resource factors that can potentially affect the performance of the BGs, we decide these two resources as main features for the performance model of these two BGs.

Also, we consider leveraging application-assisted hints from these two applications. Intuitively, the BGs' performance could be highly related to the size and number of files they manage. Fortunately, two BGs, like many other applications, provide a capability to write log files that saved how many files they scanned or backed up. To test such hints' applicability, we measure the correlation between the size and numbers of files saved in the logs and the execution progress of two BGs. Figure 8 represents the progress of file size and numbers scanned or backed up by ClamAV and Sync as per their execution. Compared to the ideal progress (black line in Figure 8), while the progress of the processed numbers and size of files are slightly different from the progress of the ideal case, it is obvious that the processed files (numbers and size) are correlated with the ideal progress of BGs. On average, the progress by the number and size of processed files has 4.85 and 4.93 of MAE[17] (Mean Absolute Error), and a harmonic mean[18] of two factors has just 2.9 of MAE over the ideal progress. Thus, we consider such hint as a feature for the performance model of BGs and use the harmonic mean of them.

3.3.2 Model Selection for BG Performance Model

We design the performance model with a multivariate linear regression [39] that models the linear relationship between independent variables (the features) and the corresponding variable y (BG's execution time). The model is formulated as below:

$$y = \sum_{i=1}^{n} \alpha_i x_i + \beta \tag{2}$$

where x is the independent variables ($x \in [CPU_{bg}, DRD_{bg}, HINT_{bg}]$) and β is a constant. The corresponding variable y means the (predicted) execution time of the BGs. In this work, we consider three features to design the performance model, and users can add/remove more features according to the performance characteristics of other BGs.

13. ClamAV is an open-source anti-virus engine used to defend user instance (e.g., VM) from computer viruses, Trojan, and other malicious threats.

14. AWS Sync is a backup application for Amazon EC2 instances similar to rsync. Sync recursively copies new or updated files from a source directory on an EC2 instance to S3 storage.

15. m3.medium instance has 1 vCPU, 3.75G RAM, and SSD drive [41]. c4.large instance has 2 vCPUs, 3.75G RAM, and SSD Drive [41].

16. 100% of CPU means the full usage of 1 vCPU.

17. $|Progress_{ideal} - Progress_{log}|$.

18. $2/(1/file\_size + 1/file\_numbers)$.

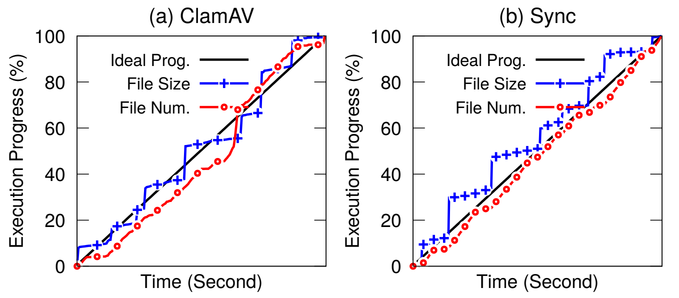


Fig. 8. Changes of file size and numbers managed by BG services. (Comparison with the ideal progress.)

3.4 Orchestra Resource Optimizer and Controller

This subsection describes Orchestra's decision process in resource allocation for both applications with two predictive models developed in the previous sections. The primary objective of resource allocation is to satisfy the FG's SLA, so the RT estimation model (Equation (1) in Section 3.2) should have the following constraint. Suppose SLA_{fg} indicates a SLA target for a FG:

$$f(CPU_{fg}, MEM_{fg}, NET_{fg}) \leq SLA_{fg} \tag{3}$$

To simplify this equation, we can consider MEM_{fg} as a *constant* because the memory resource has a weak correlation with other factors (shown in Table 1) as well as it has no significant variance with the fluctuations of FG's performance. We replace MEM_{fg} with the average memory utilization of the FG. We can also estimate NET_{fg} from EMA (Exponential Moving Average) [25]. This estimation may result in slightly inaccurate predictions for the RT estimation, but it greatly reduces the computation overhead for the RT estimation. i.e., $O(n^2)$ to $O(n)$. Now we transform the RT estimation model from multivariate to univariate model, depending on CPU_{fg}. We can obtain the minimum value of CPU_{fg} that satisfies the SLA_{fg} from the below equation:

$$\hat{CPU}_{fg} = \underset{CPU_{fg}}{\arg\min} f(CPU_{fg}) \leq SLA_{fg} \tag{4}$$

where $0 < CPU_{fg} < CPU_{max}$. CPU_{max} is the maximum amount of CPU resources in the VM. If CPU_{fg} from Equation (4) is greater than CPU_{max}, this means that the FG is impossible to meet SLA requirement with 100% CPU utilization on the instance. Thus, in this case, Orchestra provisions more resources to the FG by collaborating with cluster or application management techniques (e.g., autoscaling) to ensure SLA satisfaction. With CPU_{fg}, Orchestra can determine the CPU allocation for the BGs by:

$$CPU_{bg} = CPU_{max} - (CPU_{fg} + \varepsilon) \tag{5}$$

where ε is the CPU utilization for other applications (neither FG or BG) or the reserved amount of CPU for unknown processes.

Next, Orchestra performs an optimization to minimize Equation (2), which is the performance model of the BGs.

$$\begin{aligned}
\text{minimize:} \quad & \sum_{i=1}^{3} \alpha_i x_i + \beta, \text{ where} \\
& x_i \in \{CPU_{bg}, DRD_{bg}, HINT_{bg}\} && (6) \\
\text{subject to:} \quad & CPU_{bg} = CPU_{max} - (CPU_{fg} + \varepsilon) && (7) \\
& 0 \leq DRD_{bg} \leq DRD_{max} && (8) \\
& HINT_{bg} = 1, (100\% \text{ prog. of BG}) && (9)
\end{aligned}$$

The solution of this optimization determines the desired utilization of the BGs. From Equation (3) to (9), Orchestra determines all resource allocations to both FG and BGs. The set of resource allocation is sent to a node agent on the VM instance, and then the node agent reconfigures resource allocation with cgroups.

4 Implementation

The implementation of Orchestra follows the architectures of recent cluster management frameworks such as Kubernetes and Docker Swarm, as shown in Figure 3 (in Section 3). The main components of these architectures include a master and nodes to manage and orchestrate virtual machines, and the sidecar component is used in the micro-service architecture such as Netflix OSS and Istio[19] as a packet forwarder in both/either ingress and/or egress. In the master controller of Orchestra, two predictive models – the RT estimator (FG) and performance model (BG) – are implemented with various statistics and machine learning libraries.

The implementation of node agent focuses on resource monitoring and control. Sysstat[20] is used to periodically monitor the changes of resource utilization on the VM instances. To control multiple resources, Orchestra consults with two subsystems of cgroups – *cpu* and *blk_io* – to control the CPU and disk IO, and utilizes *tc*[21] for network IO. Whenever the new resource allocations are decided, Orchestra reconfigures a different set of tunable values to *cpu.shares* and *cfs_period_us* (for CPU control) and *read_iops* in *blk_io* (for disk IO control). While Orchestra can determine the proper reconfiguration for network IO and is fully implemented to utilize *tc*, according to our experience, reconfiguring *tc* is often unnecessary because applying the updated value of *cfs_period_us* automatically adjusts network IO as well. (Moreover, in Section 3.2, we discussed that both CPU and network IO are highly correlated.)

The sidecar – a performance monitor for the FG – is based on Nginx's reverse proxy [42] and load-balancing [43] functionality. Currently, the sidecar supports multiple protocols for the FG workloads – HTTP and data stream (TCP and UDP) requests – and it forwards the requests to the corresponding FG applications. With the Nginx's recent improvement, the sidecar can capture ingress and egress time of each request, and the statistics of the FG's RT are reported to the master controller in a real-time manner.

19. https://istio.io/
20. http://sebastien.godard.pagesperso-orange.fr/
21. http://tldp.org/HOWTO/Traffic-Control-HOWTO/intro.html

5 Performance Evaluation

We evaluate Orchestra on the real cloud environment. We first demonstrate the performance of Orchestra for controlling both FG and BGs to satisfy the FG's SLA goals under the resource storms (Section 5.2). Next, we analyze the overall accuracy of Orchestra as well as the accuracy of the RT estimator of FG and performance models of BGs (Section 5.3). We then report the overhead of Orchestra (Section 5.4).

5.1 Evaluation Setup

5.1.1 Evaluation Infrastructure

We use general purpose m4[22] instances of Amazon EC2 clouds since Orchestra aims to provide fine-grained control mechanisms of various VM resources to general cloud users (of course, all controls are performed in the user space). As several works reported [44–47], Amazon EC2 has *performance variance* due to the resource contentions and HW heterogeneity on the base infrastructure. We use EC2 spot instances in this evaluation because spot instances could have even higher level of the performance variance than that of on-demand instances. Multiple runs of experiments are performed, and we average the results to offset the *variance*.

5.1.2 FG Workloads

We consider Web application and MongoDB as representatives of FGs and use two different benchmarks for each FG to generate real workloads; CloudSuite for Web Serving benchmark and TPC-C for MongoDB. For the Web application, we generate web serving workloads from 50 to 250 concurrent users to create a sufficient level of workload fluctuation. We set up different VMs for the web server (m4.large[23]) and back-ends – Memcached and DBMS – (m4.xlarge[24]) and focus on the resource controls for the front-end (web server) VM. For MongoDB, we install the latest version of MongoDB on m4.large instance and continuously change the number of concurrent users from 2 to 20 to generate the realistic workloads.

5.1.3 BG Workloads

A 5GB of a dataset is used for BG workloads. i.e., ClamAV (virus scan) and AWS Sync (backup). This dataset is composed of about 25K of files with various sizes (μ of 1024K, σ of 1495.6). We use a different dataset from the dataset we used in Section 3.3 for a fair comparison.

5.1.4 Performance Metric

In Section 5.2, meeting a broad range of SLAs is the primary metric. We focus on tail latency – *95%tile* – for this measurement and define that a SLA requirement is satisfied if a 95%tile of a FG's response time is equal to or less than the SLA target. In Section 5.3, we use two well-known metrics of measuring the model accuracy; MAPE (Mean Absolute Percentage Error) and RMSE (Root Mean Square Error) as expressed below:

$$MAPE = \frac{1}{n}\sum_{i=1}^{n}\left|\frac{Actual_i - Predict_i}{Actual_i}\right| \tag{10}$$

22. m4 instances are the latest generation of VM instances and have balanced resource combinations, i.e., the ratio between CPU and memory is 1:4 [41].

23. m4.large has 2 vCPUs, 8GB RAM, and SSD storage.

24. m4.xlarge has 4 vCPUs, 16GB RAM, and SSD storage.

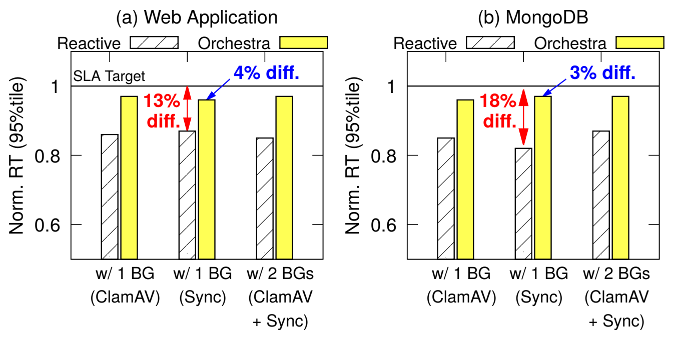


Fig. 9. Normalized RT (95%tile) of Web and MongoDB over a set of SLA targets (The best result should be 1.0.)

TABLE 3
95%tile response time (RT) of Web and MongoDB with and without resource storms from BGs.

	# BGs	Web	MongoDB
FG Only	0	0.92s	0.86s
FG + ClamAV	1	5.88s	2.71s
FG + Sync		8.65s	3.57s
FG + ClamAV + Sync	2	11.78s	4.31s

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(Predict_i - Actual_i)^2} \tag{11}$$

For both metrics, lower values indicate better accuracy. The overhead of Orchestra (Section 5.4) measures CPU consumption of the master controller and the managed nodes (including the overhead from the sidecar and node agent) when they monitor, predict, and control target applications and resources.

5.2 Overall Performance with FG Workloads

This section focuses on the overall performance of Orchestra; meeting the SLA goals while minimizing the BGs' execution time. Before we start this evaluation, we need to recognize the range (the upper and lower bound) of SLA targets that Orchestra should meet. We quantify RTs (Table 3) of four different scenarios when a FG runs with BGs (resource storms) and without BGs. We pick SLAs for each case within the range of between "RT without a BG" and "RT with BG(s)". Three SLAs – 25%, 37.5%, and 50% $\in$ [RT without a BG, RT with BG(s)] – are chosen for evaluations with resource storms. We define these three SLA requirements as a "tight SLA (with 25% padding)", "moderate SLA (with 37.5% padding)", and "loosed SLA (with 50% padding)."

We use a reactive approach from other works [12, 26] as the baseline. This reactive system relies on a dynamic adjustment of maximum resource capping. The reactive one sets a low value of the maximum cap (e.g., 5% of CPU utilization) for BGs' resource consumption when the FG's RT violates the SLA goals, and it allocates more resources to the BGs by releasing this cap when the FG shows a stable performance. Also, we add a warning system to this baseline, and the warning system sets a threshold (e.g., 10% gap from the SLAs). When the FG's 95%tile RT exceeds the

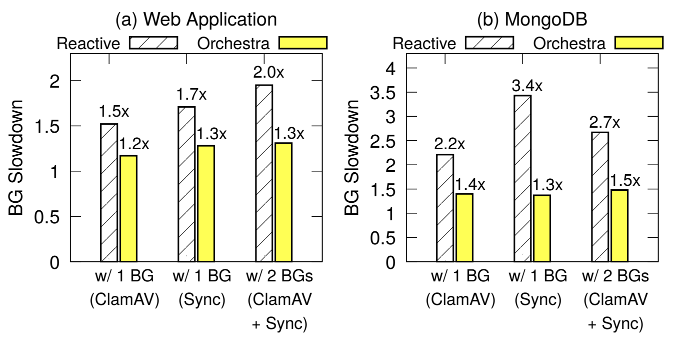


Fig. 10. Slowdown of BG's execution under two different control frameworks.

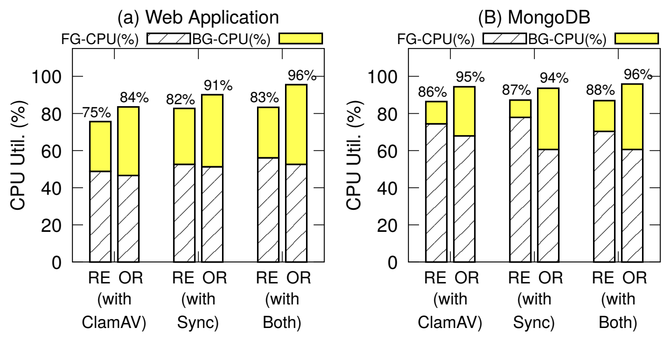


Fig. 11. Resource utilization (CPU) of VM with two control frameworks. (RE: reactive framework, OR: Orchestra)

threshold, the warning system alerts to the master controller, and the reactive framework reduces the resource usage of the BGs with a pre-defined step function. If the FG's RT violates the SLAs, the reactive one uses the resource capping explained above to quickly restore the FG's performance with the SLA.

Figure 9 shows the RT (95%tile) of the FG applications managed by Orchestra and reactive approach when the FGs are running with the BGs. All results are normalized over the SLA targets. If a result is equal to or less than 1.0, then the RT meets the SLA goals and vice versa. As shown in Figure 9, both frameworks can successfully control the FGs' RT with the SLAs. But Orchestra's results are much closer to the SLAs, and it only has 2 – 5% difference with the SLAs. However, the reactive system has about 15% differences with the SLA targets. These results indicate that the FGs controlled by Orchestra consumes significantly less amount of resources to meet the SLA targets as compared to the FGs with the reactive system. Again, the goal of this work is *not* maximizing a FG's performance, rather meeting the FG's SLA as well as augmenting the performance of a BG(s). These results have a significant impact on the BGs' performance since Orchestra enables the BGs to utilize more resources to boost their execution (more importantly, without SLA violations). Figure 10 reports the difference between the BG(s)'s performance and obviously shows the benefits from Orchestra. While Orchestra only has 1.25x (with Web) and 1.39x (with MongoDB) slowdown[25] of the BGs' execution, the reactive approach requires more sacrifice to the BGs, i.e., 1.73x (with Web) and 2.77x (with MongoDB) slowdown of the BGs. These results are because Orchestra's predictive ability and optimization mechanism could successfully determine the proper level of resource allocation to multiple applications so that Orchestra allows the BGs to consume as high resource as if the FG meets the SLAs.

Figure 11 shows resource utilization on the VMs with both approaches. We only show CPU utilization since it is the most representative resource of the VMs. For the Web application case (Figure 11(a)), Orchestra utilizes over 90% of CPUs, which is leveraging 10% more resources than the reactive approach. Interestingly, Orchestra uses a similar amount of CPU (compared to the reactive one, only 2% difference) for the Web (FG) and increases the overall utilization by allocating more resources to the BGs. In the evaluation with MongoDB (Figure 11(b)), Orchestra improves CPU utilization over 95% and provides balanced resource allocations to the both, implying that Orchestra is not only able to meet the FG's SLAs, but also boost the BG's performance, resulting in high resource utilization on the VMs. However, the reactive one allocates more than 75% of CPU to the FG, indicating it overly assigns the resources that lead to retard the BGs' execution.

25. This is normalized over the BGs' execution without FG workloads.

Figure 12 illustrates Orchestra's behavior for managing FG's RT and controlling VM resources. Please note that we determine SLA requirements for Web (3.0s) and MongoDB (2.5s) in a similar manner with the evaluation results reported in Figure 9, 10, and 11. We measured Orchestra's performance with different (both shorter and longer) SLA requirements, but the results were similar to Figure 12.

The top graph in Figure 12 shows the changes in FG's RT and the SLA target. The middle graph reports the CPU control for the FG, and the bottom graph shows the CPU control for the BG. For the use case of Web (Figure 12(a)), while Orchestra controls the CPU resources based on its estimation mechanism, the most critical SLA violation happens in the first 100 seconds (the top-left graph). This violation is because the RT estimation model for FG does not have enough training dataset to build a robust model. After this initial training period, Orchestra performs the successful control in the CPU resources. The middle and bottom graphs in Figure 12(b) show that Orchestra increases CPU resources for the FG and, at the same time, decreases the resource for the BG when the FG's RT is close to or violates the SLA target. i.e., at 220, 250, 400, and 580 second. Similar behaviors are shown in the MongoDB case (Figure 12(b)). The first 350 second period (top right graph) can be considered as the training period. After this training period, Orchestra shows stable adjustments in CPU resources for both applications regarding SLA satisfaction. For instance, during the period between 600- and 800-second, Orchestra reduces the CPU allocation to the FG (middle graph) and allocates more to the BGs (bottom graph) as the MongoDB's RT is a lot faster than the SLA goal. Another example can be shown in the period between 900- and 1100-second. Orchestra increases the CPU allocation to the FG while the CPU allocation for the BG decreases since the MongoDB's RT is close or slightly violates the SLA goal. Then, the FG's RT becomes stable.

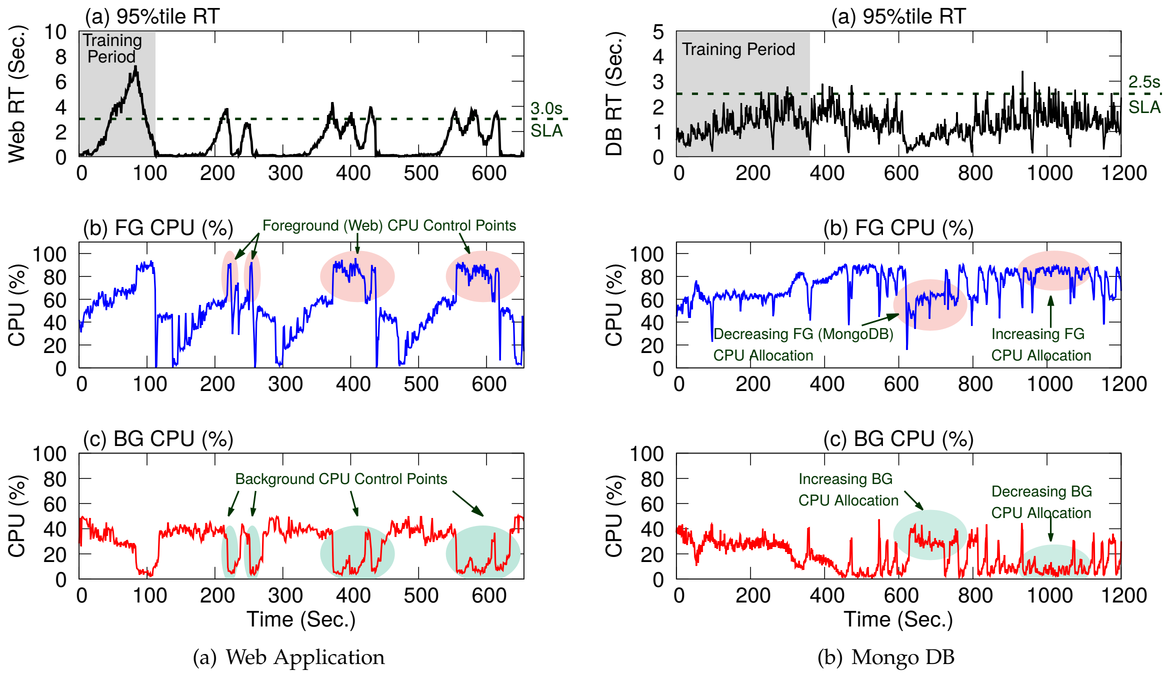


Fig. 12. Change of CPU resources for both a FG and BGs from Orchestra and the changes of 95%tile RT of the FG. (a): Web (FG) and Sync (BG) with a SLA of 3.0 sec.; (b): MongoDB (FG) and 2 BGs (Sync and ClamAV) with a SLA of 2.5 sec.

5.3 Orchestra Accuracy

Performance evaluations of Orchestra's accuracy include the overall model accuracy of Orchestra and the accuracy of two predictive models. i.e., the RT estimator (FG) and the BG performance model.

5.3.1 *Overall Orchestra Accuracy*

We first collect various system/application statistics (including RT, utilization, and throughput from FGs, and BG's execution) with a set of different resource configuration between a FG and a BG. We manually assign fixed resource ratios to the FG and BG(s) from 1:1 to 1:5. We then generate the identical FG workloads to Orchestra with SLA targets obtained from the static resource allocation. We compare the results/statistics from Orchestra with the ones from the static resource allocation and calculate the accuracy (normalized over the static allocation) of both cases.

First, we show the accuracy of Orchestra regarding FG behaviors. The results are shown in Table 4 and Figure 13. The changes in FG's RT with Orchestra are very close to the RT variations in static resource allocations. On average, Orchestra can control a FG with 2.7% errors in RT and 1.95% error for its throughput.

Then, we observe the difference in the resource utilization of the VM instance between Orchestra and the static resource allocations. Figure 14 illustrates the comparison results. Figure 14(b) and (d) have a "NET" (network IO) filed since the Sync (backup BG) is a network-bound application. For all comparison cases, Orchestra shows only 3% errors in CPU, 7% errors in disk IO (read), and 2% errors in disk IO (write), indicating that Orchestra is very accurate for FG applications. However, for the BGs, Orchestra shows slightly different statistics. Especially when Orchestra manages the Web application as a FG, it allows BGs to consume 17% to 20% more resources. This difference results in a distinct execution time of BGs. With more resources, Orchestra could reduce the execution time of BGs (15% – 17% reduction for both BGs). The differences in BG's utilization are due to the following reasons. Orchestra mechanism to manage BGs naturally allocates more resources to the BGs if the FG's RT meets a SLA, and the VM has residue resources possibly being consumed by the BGs. Furthermore, this is also related to FG's workloads. The workloads generated by CloudSuite have higher variation than DB workloads from TPC-C. This high variance is not only relevant to the changes in the number of requests but also associated with the diverse types of requests, i.e., from simple web renderings to complex social activities in web sites. Given the high fluctuating workloads, it is evident that Orchestra tries to minimize the execution duration of BGs, and this is a clear indication that Orchestra works correctly under such workloads. Similarly, the higher amount of network TX in Figure 14(b) is because of the same reason that the BG (Sync) sends the larger amount of backup data per second to S3 with more allocated resources.

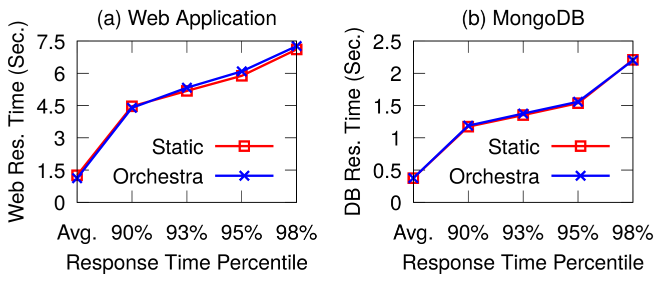


Fig. 13. Accuracy for the FG's RT; Comparison with the static resource allocation.

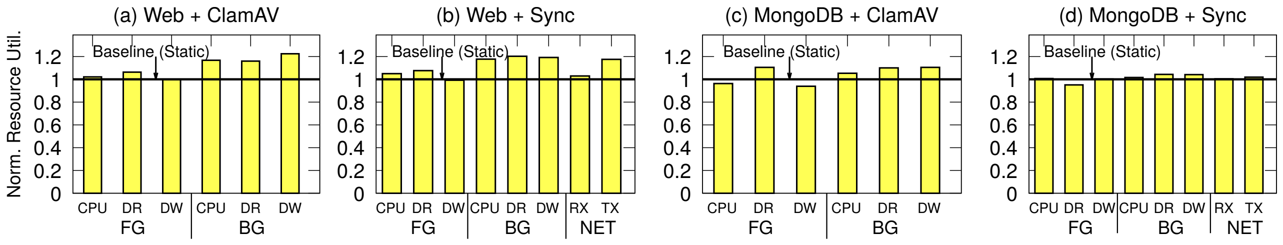


Fig. 14. Accuracy comparison of VM utilization between Orchestra and the static resource allocation. (NET: Network IO, DR: Disk Read bytes/sec., DW: Disk Write byte/sec., TX: Network TX bytes/sec., RX: Network RX bytes/sec.)

TABLE 4
Orchestra's accuracy (MAPE) over the static resource allocation. The results show FG statistics. (MDB: MongoDB, T-put: Throughput)

FG	BG	RT Statistics (%tile)				T-put (Req/s)
		Avg.	90%	95%	98%	
Web	ClamAV	0.050	0.061	0.038	0.011	0.007
	Sync	0.132	0.008	0.029	0.028	0.020
MDB	ClamAV	0.010	0.015	0.006	0.013	0.036
	Sync	0.003	0.012	0.014	0.003	0.015

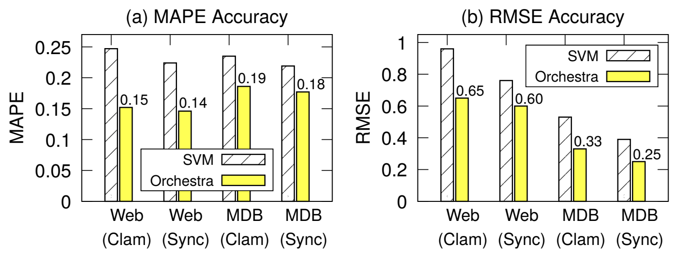


Fig. 15. Accuracy comparison of RT estimation of Orchestra with SVM. Lower values mean better accuracy.

5.3.2 *Accuracy of RT Estimator (FG)*

Next, we measure the accuracy for the RT estimator (described in Section 3.2) in Orchestra. We calculate the accuracy of the RT estimator by comparing its prediction results with the actual RT (the ground truth) from the FGs. The results are that Orchestra shows 0.17 of MAPE and 0.45 of RMSE.

To validate these results are sufficient, we compare the performance of the RT estimator with a SVM (Support Vector Machine)-based predictor. SVM [39] is a well-known classifier widely used in data mining and machine learning area, but it shows very robust performance for regression (prediction) problems, i.e., application performance modeling [48]. To optimize the performance of the baseline, we employ RBF (Radial Basis Function) as the kernel of the SVM and test a broad set of (soft-margin and kernel) parameters with a grid approach [49]. Figure 15 illustrates the comparison results of both models. The RT estimator of Orchestra outperforms the SVM and has approximately 40% lower prediction errors than the results from the SVM model. The SVM model just shows 0.23 of MAPE and 0.64 of RMSE.

5.3.3 *Accuracy of BG Performance Model*

We also measure the accuracy of the BG performance model (described in Section 3.3) in Orchestra. The model predicts the BGs' execution time whenever Orchestra changes the resource allocation to the FG and BGs. If the workloads have high fluctuations, Orchestra predicts the BGs' execution time more often. Given the life-cycle of a BG's execution, the prediction model can show higher accuracy when the BG is approaching the end of the execution. However, it is important for Orchestra to accurately predict the BGs' execution time (or finishing time), particularly when it reconfigures resource allocations to the BG in the middle of its life-cycle. We thus measure accuracy at two different points, i.e., 1) the overall accuracy – averaging all predictions during BG's execution and 2) the performance in the critical section. We define "30% – 70%" of a BG's execution as the critical section. For this evaluation, we employ an application-specific predictor as a baseline. This predictor only relies on application-assisted hints (execution progress) and forecasts the BG's execution time by calculating the progress ratio.

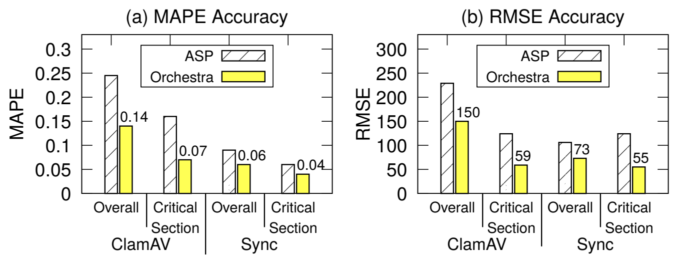


Fig. 16. Accuracy comparison of the BG performance model in Orchestra with the application-specific predictor.

Figure 16 shows the accuracy results. While both predictors show good accuracy, Orchestra outperforms the baseline in all cases. On average, Orchestra has 0.11 of MAPE and 112 of RMSE, indicating that it has 67% and 50% less error than the baseline. More interestingly, Orchestra produces more accurate predictions in the critical section, and it only makes 0.06 of MAPE and 59 of RMSE, showing 2x better accuracy than the overall results.

5.4 Orchestra Overheads

Orchestra's overhead is also a significant characteristic in the evaluation. Since Orchestra dynamically controls VM resources at runtime, it is also desired that the framework should not interfere with the performance of FG and BGs. In other words, Orchestra should not generate resource storms. The overheads are measured from two different components in Orchestra, i.e., the managed node and the master controller.

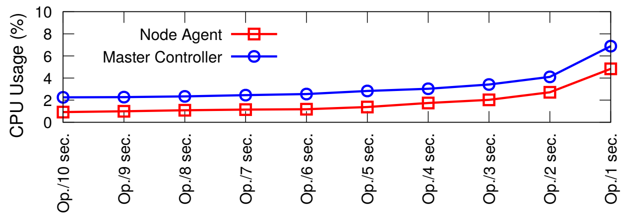


Fig. 17. Orchestra Overhead.

Figure 17 reports the overhead (CPU usage) in Orchestra. The result measured in the managed node includes the overhead in performance monitoring from the sidecar, and resource usage monitoring and reconfiguring cgroups from the node agent. The master controller's results focus on the computational overheads for the management including prediction and optimization cost. The CPU consumption increases as the frequency of management operation in Orchestra increases. Orchestra consumes 1% – 5% of CPU resources on the managed node and 2% – 7% on the master controller. In the previous evaluation with FG workloads (§5.2), while Orchestra performs such monitoring and control operations in every 5-seconds, it shows a desirable performance in managing both FG and BG's performance, indicating that 1% – 2% of CPU resources is sufficient to Orchestra. However, it is worth noting that using a fixed-interval (e.g., every 5-seconds) for monitoring and controlling operation may not be the optimal approach. In particular, by considering temporal randomness of resource storms, it would better to employ variable monitoring and control intervals. Therefore, in the near future, we will investigate a variable interval-based approach that can minimize the management overhead as well as address the resource storm more efficiently.

6 Discussion

6.1 Comparison with Static Resource Allocation

To prevent the performance interference caused by resource storm, users can consider static resource allocation scheme. For example, by using cgroups or CPU hard-capping [12], the users can allocate the fixed portion of resources to FG and BGs. i.e., 60% of CPU to FG, 40% of CPU to BGs. One benefit from this static allocation is that BGs' resource storm or heavy workload will not degrade the FG's performance because the BGs' resource consumption is strictly controlled by the limitation (e.g., 60%).

However, this static allocation is not efficient enough to maximize the resource utilization of a VM. For example, it often has an idle (unused) portion of resources while either FG or BG has heavy workloads. Differing from the static approach, Orchestra uses dynamic resource allocation based on the SLA and workload changes. Orchestra is designed to guarantee the performance SLA of FG as well as maximize the execution of BGs. To satisfy these two goals, the dynamic resource allocation is more desirable for Orchestra because it enables to dynamically adjust the resource allocation to both FG and BGs as per the performance changes.

6.2 Can Auto-scaling be a Solution for Resource Storms?

Auto-scaling is a widely used approach for the application management on the cloud that fully leverages the benefit – *elasticity* – of cloud infrastructure. Intuitively, the auto-scaling might be considered as an approach to address resource storm by adding more resources to cloud applications when the resource storms happen. (Note that auto-scaling is different from the resource over-provisioning in Section 2.1.) The auto-scaling relies on VMs' resource utilization, and its scaling mechanism could be automatically triggered with high peaks of resource utilization by the resource storms. However, auto-scaling is not sufficient to handle resource storm for the following two reasons.

First, resource storms can occur in any cloud instances (VMs). The VMs, newly added by the auto-scaling, are often exposed to the resource storms as well. The BG tasks – *security*, *compliance*, and *patch* – are inevitable even for the new VMs in practice. A worst-case scenario is that cascading resource storms can incur poor QoS for the entire service from a cloud application. Moreover, after finishing resource storm, the resource provisioning state of the cloud applications quickly becomes over-provisioning, which hurts the cost-efficiency of the cloud application by wasting the resources.

Second, it is often very challenging for the auto-scaling to immediately respond to the resource storms. According to our personal communication with cloud practitioners, the duration of resource storm tends to be short (often less than 10 minutes) [36]. We also confirm the similar characteristics of resource storm from the measurement reported in Figure 2 (Section 2.1). However, the auto-scaling relies on VM monitoring tools like CloudWatch that monitors the resource utilization on the VMs and catches an unusual resource spike. Unfortunately, a previous work [50] reported that CloudWatch has 1 to 5 minutes of delay in obtaining proper monitoring results. Also, a VM provisioning has a delay of 3 to 5 minutes [51]. Enterprise cloud instances generally contain various SW stacks (e.g., web server, DBMS, Memcached) and would take a longer provisioning time than that of vanilla VMs. Furthermore, the research community proposed hybrid and/or predictive autoscaling approaches [52–54] that can be considered for addressing resources storm. However, as we discussed in Section 2.1, resource storm often abruptly happens at any time, so it is extremely challenging to provision more VM resources a priori with an accurate prediction of the occurrence of resource storm. This problem is even more complicated when considering large-scale deployment and multiple BGs. Therefore, it is more appropriate to design an avoidance mechanism for resource storms inside VMs instead of employing auto-scaling mechanisms.

6.3 Determining Optimal Parameters for Orchestra

Orchestra has three input parameters for addressing resource storm; (1) degree of tail latency for FG applications, (2) SLA requirement, and (3) monitoring and controlling interval.

Regarding the degree of the tail latency, while we used 95%tile as the latency requirement for FG applications in the evaluation, this degree is a parameter that is provided by

Orchestra users, indicating that diverse values can be used for the latency requirements such as 90%tile, 95%tile, or 99%tile. As shown in Figure 13 and Table 4 (in Section 5.3.1), Orchestra can accurately estimate the different degrees of tail latencies, and, based on the accurate prediction, Orchestra can control VM resources and address resource storm. However, a tighter requirement of tail latency may require more frequent operations of monitoring and resource control. We will further discuss the overhead in the last paragraph of this section.

The SLA requirement is another parameter provided by the Orchestra users. In our evaluation, we determined the SLA requirement between a possible lower bound of SLA and an upper bound SLA. The lower bound can be measured when the target FG application is running without any BG services, indicating that the FG can leverage all the available resources in a VM. The upper bound can be measured when the FG is running with BG services without any resource control. Basically, Orchestra can manage the performance of FG within this range. If the users set a too-small value for SLA requirement (e.g., less than the lower bound), Orchestra cannot support this SLA requirement. In this case, the users need to consider migrating the FG application to a higher performance VM (with more resources).

The last parameter is the monitoring/controlling interval. As shown in Figure 17, a longer interval consumes fewer CPU resources, and a shorter (more frequent) interval consumes more CPU resources. Therefore, the users should determine proper monitoring interval by considering the overhead as well as the degree of tail latency. However, as we reported, the high frequency of the monitoring/controlling operation may consume up to 10% of CPU, but in general, Orchestra consumes fewer than 2% of CPU resources. We believe 2% of resource consumption may not be a significant overhead caused by Orchestra.

7 Related Work

7.1 Infrastructure-Level Resource Management

There are significant works from the research community to detect, prevent, mitigate, and manage the performance interference caused by multiple co-located tasks/applications on the same physical HW. One direction is to design an intelligent QoS management framework that detects the performance interference and schedules multiple tasks to avoid a FG's SLA violations. Q-Clouds [23] predicts a SLA violation with a discrete-time MIMO (Multi-Input and Multi-Output) model. DejaVu [20] employs a performance index that determines the interference by comparing it with the identical executions on a sandbox. DeepDive [21] detects the performance interference with a warning system and manages it with VM cloning and workload duplication. Dirigent [25] controls is a FG's QoS while improving batch tasks' throughput. Dirigent is relying on particular performance isolation techniques. i.e., Intel's cache allocation technique.

The other direction is to determine safe task placement. Bubble-up [3] finds a safe co-location of multiple tasks on the same host with a sensitivity curve via offline profiling. Bubble-Flux [26] dynamically creates the sensitivity curve from online profiling with a short-term memory-intensive workload. Paragon [5] performs minimal offline profiling for new workloads (e.g., 1 min. on two different HWs) and uses collaborative filtering to place such tasks on the particular HWs. CPI^2 [12] suggests CPI (cycle-per-instruction) as a performance indicator to detect performance interference and manages the FG's QoS with CPU hard-capping to antagonists (source of the interference). Heracles [24] finds a safe co-location of multiple tasks with a coordinated isolation mechanism of shared resources.

However, the approaches mentioned above have the following limitations and differences; they often require 1) an on/off-line profiling to model the performance interference, 2) HW information (e.g., CPI, cache miss rate.), and 3) an expensive VM cloning and sand-box executions. On the other hand, Orchestra does not perform such profiling; rather Orchestra employs an online model to determine the optimal resource allocation to both FG and BGs on the same VM. Orchestra does not collect HW information, but periodically gathers system/application-level statistics. i.e., CPU, IO, and RT. This policy is because we consider that our target users are consumers for public IaaS clouds where provide limited access to the hypervisor layer and host machines [29].

7.2 User-Level Application Management

There are several attempts to address this problem in the user space. IC^2 [28], ICE [55], and Amannejad et al. [56] proposed systems to detect and mitigate the performance interference for web applications. The performance interference is detected by resource monitoring, statistic inference model or collaborative filtering. Once the interference is identified, the approaches often change application configurations for increasing throughput of web services. However, these approaches are commonly application-specific (web service); on the other hand, the Orchestra framework supports diverse types of FGs. Moreover, Orchestra does not change any application configuration. Such modifications could result in high overhead and only mitigate short-term performance interference [57]. Stay-away [55] manages process containers (LCX or Docker) and mitigates the interference by throttling BGs. However, the control mechanism depends on a *diurnal* pattern of user-workloads, having a clear period of low intensity. This may not be true for modern cloud applications [1, 30] and, more importantly, Orchestra is agnostic to user workload patterns.

7.3 Intelligent Scheduling and Load Balancing

Chiang and Huang proposed an interference-aware scheduler called TRACON [58], which predicts interference in IO operations in data-intensive applications and performs intelligent scheduling based on the predicted IO-interference of VMs. Pu et al. [59] proposed a similar approach that mitigates IO interference on VMs on the same physical machine. MIMP [60] is a two-schedulers system to improve the performance of both interactive and batch (Hadoop) applications by mitigating performance interference caused by multiple VMs hosted on the same physical machine. CloudScope [61] is another interference-aware scheduling for Xen to detect performance interference for multi-tenant cloud systems. CloudScope predicts the performance interference of VMs with a discrete-event Markov Chain model and reconfigures the placement of VMs on physical machines. However, these approaches do not consider co-running applications on the same VMs and require modification of the hypervisor scheduler (e.g., Xen) to mitigate the performance interference. Thus, none can be leveraged by public cloud users because public cloud users do not have control over the hypervisor layer.

Additionally, there are several load-balancing approaches to address performance interference in various cloud applications. i.e., HPC [62, 63], storage systems [64], web [65], and OLTP systems [52]. These approaches commonly try to detect or infer resource contention in underlying cloud systems and mitigate the interference with adopting improved load-balancing schemes. In particular, DIAL [52, 65] is a user-centric interference-aware load-balancing scheme for cloud applications. Similar to Orchestra, it is designed to be used in public cloud environments and improve the tail latency of cloud applications. However, these load balancing schemes do not solve the resource storm problem targeted in this paper for two reasons. First, many BG services, including backup and virus scans, cannot be scheduled for other VMs to be executed. They must be executed on the VM (or server) to be maintained (e.g., backup and virus scan). Second, most existing load balancing works do not address the resource contention problem in a VM, which has been a significant cause for performance degradation under resource storms. For example, prior work has shown that a contention-oblivious scheduling algorithm can perform sub-optimally in contended environments [5].

8 Conclusion

Resource storms in enterprise cloud environments become a significant challenge for managing the performance of cloud applications. To improve this situation, we presented Orchestra, *cloud-specific framework for controlling both FG and BGs in the user space* to guarantee the FG's performance while minimizing the performance penalty of BGs. Orchestra measures a FG's performance (RT) in a real-time manner, and it creates a lightweight RT estimation and performance models for both applications on the fly. With the resource statistics and such predictive models, Orchestra optimizes the resource allocation to multiple cloud applications with SLA targets.

We have implemented and evaluated Orchestra with real workloads on Amazon EC2. Our primary workloads are a web service and a NoSQL database (MongoDB) for FGs and AWS Sync (backup) and ClamAV (virus and malware scanner) for BGs. We also presented the performance of Orchestra with a number of SLA constraints. Orchestra guarantees the FG's SLA satisfaction at all times with 70% performance improvement of BGs. Moreover, Orchestra shows a very high overall correctness (less than 5% error), 16.5% of MAPE for the FGs' response time estimation, and over 90% accuracy for the BGs' performance prediction.

References

[1] George Amvrosiadis, Angela Demke Brown, and Ashvin Goel. Opportunistic Storage Maintenance. In *25th Symposium on Operating Systems Principles (SOSP '15)*, Monterey, CA, USA, October 2015.

[2] In Kee Kim, Jinho Hwang, Wei Wang, and Marty Humphrey. Orchestra: Guaranteeing Performance SLA for Cloud Applications by Avoiding Resource Storms. In *IEEE International Symposium on Parallel and Distributed Computing (ISPDC '18)*, Geneva, Switzerland, June 2018.

[3] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *44th International Symposium on Microarchitecture (Micro '11)*, Porto Alegre, Brazil, December 2011.

[4] Lingjia Tang, Jason Mars, Wei Wang, Tanima Dey, and Mary Lou Soffa. ReQoS: Reactive Static/Dynamic Compilation for QoS in Warehouse Scale Computer. In *18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*, Houston, TX, USA, March 2013.

[5] Christina Delimitrou and Christos Kozyrakis. Paragon: QoS-aware Scheduling for Heterogeneous Datacenters. In *18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*, Houston, TX, USA, March 2013.

[6] Francisco Romero and Christina Delimitrou. Mage: Online and Interference-aware Scheduling for Multi-scale Heterogeneous Systems. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, 2018.

[7] Lingjia Tang, Jason Mars, and Mary Lou Soffa. Compiling for Niceness: Mitigating Contention for QoS in Warehouse Scale Computers. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, 2012.

[8] Ram Srivatsa Kannan, Lavanya Subramanian, Ashwin Raju, Jeongseob Ahn, Jason Mars, and Lingjia Tang. GrandSLAm: Guaranteeing SLAs for Jobs in Microservices Execution Frameworks. In *Proceedings of the Fourteenth EuroSys Conference 2019*, 2019.

[9] Lingjia Tang, Jason Mars, Neil Vachharajani, Robert Hundt, and Mary Lou Soffa. The Impact of Memory Subsystem Resource Sharing on Datacenter Applications. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, 2011.

[10] Greg Linden. Make Data Useful. http://www.gduchamp.com/media/StanfordDataMining.2006-11-28.pdf. [ONLINE].

[11] S. Shunmuga Krishnan and Ramesh K. Sitaraman. Video Stream Quality Impacts Viewer Behavior: Inferring Causality Using Quasi-Experimental Designs. In *ACM SIGCOMM Internet Measurement Conference (IMC '12)*, Boston, MA, USA, November 2012.

[12] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. CPI^2: CPU Performance Isolation for Shared Compute Clusters. In *8th ACM European Conference on Computer Systems (Eurosys '13)*, Prague, Czech Republic, April 2013.

[13] Jacob Leverich and Christos Kozyrakis. Reconciling High Server Utilization and Sub-millisecond Quality-of-Service. In *9th ACM European Conference on Computer Systems (Eurosys '14)*, Amsterdam, Netherlands, April 2014.

[14] Alexandra Fedorova, Margo Seltzer, and Michael D. Smith. Improving Performance Isolation on Chip Multiprocessors via an Operating System Scheduler. In *16th International Conference on Parallel Architecture and Compilation Techniques (PACT '07)*, Brasov, Romania, September 2007.

[15] Daniel Shelepov and Juan Carlos Saez Alcaide and Stacey Jeffery and Alexandra Fedorova and Nestor Perez and Zhi Feng Huang and Sergey Blagodurov and Viren Kumar. HASS: A Scheduler for Heterogeneous Multicore Systems. *ACM SIGOPS Operating Systems Review*, 43(2):66–75, April 2009.

[16] OpenSUSE, Tuning the Task Scheduler. https://doc.opensuse.org/documentation/leap/tuning/html/book.sle.tuning/cha.tuning.taskscheduler.html, 2019. [ONLINE].

[17] David Koufaty and Dheeraj Reddy and Scott Hahn. Bias Scheduling in Heterogeneous Multi-core Architectures. In *5th ACM European Conference on Computer Systems (Eurosys '10)*, Paris, France, April 2010.

[18] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale Cluster Management at Google with Borg. In *10th ACM European Conference on Computer Systems (Eurosys '15)*, Bordeaux, France, April 2015.

[19] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: Flexible, Scalable Schedulers for Large Compute Clusters. In *8th ACM European Conference on Computer Systems (Eurosys '13)*, Prague, Czech Republic, April 2013.

[20] Nedeljko Vasic, Dejan Novakovic, Svetozar Miucin, Dejan Kostic, and Ricardo Bianchini. DejaVu: Accelerating Resource Allocation

in Virtualized Environments. In *17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '12)*, Bordeaux, France, December 2012.

[21] Dejan Novakovic, Nedeljko Vasic, Stanko Novakovic, Dejan Kostic, and Ricardo Bianchini. DeepDive: Transparently Identifying and Managing Performance Interference in Virtualized Environments. In *2013 USENIX Annual Technical Conference (ATC '13)*, San Jose, CA, USA, June 2013.

[22] Harshad Kasture and Daniel Sanchez. Ubik: Efficient Cache Sharing with Strict QoS for Latency-Critical Workloads. In *19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*, Salt Lake City, UT, USA, March 2014.

[23] Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. Q-Clouds: Managing Performance Interference Effects for QoS-Aware Clouds. In *5th ACM European Conference on Computer Systems (Eurosys '10)*, Paris, France, April 2010.

[24] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving Resource Efficiency at Scale. In *42nd Annual International Symposium on Computer Architecture (ISCA '15)*, Portland, OR, June 2015.

[25] Haishan Zhu and Mattan Erez. Dirigent: Enforcing QoS for Latency-Critical Tasks on Shared Multicore Systems. In *21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*, Atlanta, GA, USA, April 2016.

[26] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. Bubble-Flux: Precise Online QoS Management for Increased Utilization in Warehouse Scale Computers. In *40th Annual International Symposium on Computer Architecture (ISCA '13)*, Tel-Aviv, Israel, June 2013.

[27] The PMCs of EC2: Measuring IPC. http://www.brendangregg.com/blog/2017-05-04/the-pmcs-of-ec2.html, 2019. [ONLINE].

[28] Amiya K. Maji, Subrata Mitra, Bowen Zhou, Saurabh Bagchi, and Akshat Verma. Mitigating interference in cloud services by middleware reconfiguration. In *15th International Middleware Conference (Middleware '14)*, Bordeaux, France, December 2014.

[29] Anshul Gandhi, Parijat Dube, Alexei Karve, Andrzej Kochut, and Harsha Ellanti. The Unobservability Problem in Clouds. In *International Conference on Cloud and Autonomic Computing (ICCAC '15)*, Cambridge, MA, USA, September 2015.

[30] Sadeka Islam, Srikumar Venugopal, and Anna Liu. Evaluating the Impact of Fine-scale Burstiness on Cloud Elasticity. In *ACM Symposium on Cloud Computing (SoCC '15)*, Hawaii, USA, August 2015.

[31] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis. In *ACM Symposium on Cloud Computing (SoCC '13)*, San Jose, CA, USA, October 2013.

[32] Marcus Carvalho, Walfredo Cirne, Francisco Brasileiro, and John Wilkes. Long-term SLOs for Reclaimed Cloud Computing Resources. In *ACM Symposium on Cloud Computing (SoCC '14)*, Seattle, WA, USA,, November 2014.

[33] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-Efficient and QoS-Aware Cluster Management. In *19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*, Salt Lake City, UT, USA, March 2014.

[34] In Kee Kim, Sai Zeng, Christopher C. Young, Jinho Hwang, and Marty Humphrey:. A Supervised Learning Model for Identifying Inactive VMs in Private Cloud Data Centers. In *ACM/IFIP/USENIX International Middleware Conference (Middleware '16)*, Trento, Italy, December 2016.

[35] In Kee Kim, Sai Zeng, Christopher C. Young, Jinho Hwang, and Marty Humphrey. iCSI: A Cloud Garbage VM Collector for Addressing Inactive VMs with Machine Learning. In *IEEE International Conference on Cloud Engineering (IC2E '17)*, Vancouver, Canada, April 2017.

[36] Personal communication with the IBM enterprise data center infrastructure team, 2019.

[37] Marius A. Eriksen. Trickle: A userland bandwidth shaper for unix-like systems. In *USENIX Annual Technical Conference, FREENIX Track*, 2005.

[38] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the Clouds – A Study of Emerging Scale-out Workloads on Modern Hardware. In *17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '12)*, Bordeaux, France, December 2012.

[39] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. The Element of Statistical Learning: Data Mining, Inference, and Prediction. 2011.

[40] Sadeka Islam, Jacky Keung, Kevin Lee, and Anna Liu. Empirical Prediction models for Adaptive Resource Provisioning in the Cloud. *Future Generation Computer Systems*, 28(1), 2012.

[41] Amazon EC2 Instance Types. https://aws.amazon.com/ec2/instance-types/, 2019. [ONLINE].

[42] Nginx Reverse Proxy. https://www.nginx.com/resources/admin-guide/reverse-proxy/, 2019. [ONLINE].

[43] Nginx Load Balancing – TCP and UDP Load Balancer. https://www.nginx.com/resources/admin-guide/tcp-load-balancing/, 2019. [ONLINE].

[44] Zhonghong Ou, Hao Zhuang, Jukka K. Nurminen, Antti Yl-Jski, and Pan Hui. Exploiting Hardware Heterogeneity within the Same Instance Type of Amazon EC2. In *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '12)*, Boston, MA, USA, June 2012.

[45] In Kee Kim, Wei Wang, and Marty Humphrey. PICS: A Public IaaS Cloud Simulator. In *IEEE International Conference on Cloud Computing (CLOUD '15)*, New York, NY, USA, 2015.

[46] Philipp Leitner and Jrgen Cito. Patterns in the Chaos–A Study of Performance Variation and Predictability in Public IaaS Clouds. *ACM Transactions on Internet Technology*, 16(15), August 2016.

[47] Sen He, Glenna Manns, John Saunders, Wei Wang, Lori Pollock, and Mary Lou Soffa. A Statistics-based Performance Testing Methodology for Cloud Applications. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019)*, Tallinn, Estonia, August 2019.

[48] Ron C. Chiang, Jinho Hwang, H. Howie Huang, and Timothy Wood. Matrix: Achieving Predictable Virtual Machine Performance in the Clouds. In *11th International Conference on Autonomic Computing (ICAC '14)*, Philadelphia, PA, USA, June 2014.

[49] James Bergstra, Remi Bardenet, Yoshua Bengio, and Balazs Kegl. Algorithms for Hyper-Parameter Optimization. In *Neural Information and Processing Systems (NIPS '11)*, Granada, Spain, December 2011.

[50] Michael Smit, BradleySimmons, and Marin Litoiu. Distributed, application-level monitoring for heterogeneous clouds using stream processing. *Future Generation Computer Systems*, 29(8), 2013.

[51] Ming Mao and Marty Humphrey. A Performance Study on the VM Startup Time in the Cloud. In *IEEE International Conference on Cloud Computing (CLOUD '12)*, Honolulu, HI, USA, June 2012.

[52] Seyyed Ahmad Javadi and Anshul Gandh. User-Centric Interference-Aware Load Balancing for Cloud-Deployed Applications. *IEEE Transactions on Cloud Computing*, 2019.

[53] Waheed Iqbal, Abdelkarim Erradi, Muhammad Abdullah, and Arif Mahmood. Predictive Auto-scaling of Multi-tier Applications Using Performance Varying Cloud Resources. *IEEE Transactions on Cloud Computing*, 2019.

[54] Shashank Shekhar, Hamzah Abdel-Aziz, Anirban Bhattacharjee, Aniruddha S. Gokhale, and Xenofon D. Koutsoukos. Performance Interference-Aware Vertical Elasticity for Cloud-Hosted Latency-Sensitive Applications. In *11th IEEE International Conference on Cloud Computing, (CLOUD)*, San Francisco, CA, USA, July 2018.

[55] Navaneeth Rameshan, Leandro Navarro, Enric Monte, and Vladimir Vlassov. Stay-Away, protecting sensitive applications from performance interference. In *15th ACM/IFIP/USENIX International Middleware Conference (Middleware '14)*, Bordeaux, France, December 2014.

[56] Yasaman Amannejad, Diwakar Krishnamurthy, and Behrouz Homayoun Far. Managing Performance Interference in Cloud-Based Web Services. *IEEE Transactions on Network and Service Management*, 12(3):320–333, 2015.

[57] Amiya K. Maji, Subrata Mitra, and Saurabh Bagchi. ICE: An Integrated Configuration Engine for Interference Mitigation in Cloud Services. In *IEEE International Conference on Autonomic Computing (ICAC '15)*, Grenoble, France, July 2015.

[58] Ron C. Chiang and H. Howie Huang. TRACON: Interference-Aware Scheduling for Data-Intensive Applications in Virtualized Environments. In *International Conference on High Performance Computing Networking, Storage and Analysis (SC '11)*, Seattle, WA, USA, November 2011.

[59] Xing Pu, Ling Liu, Yiduo Mei, Sankaran Sivathanu, Younggyun

Koh, Calton Pu, and Yuanda Cao. Who Is Your Neighbor: Net I/O Performance Interference in Virtualized Clouds. *IEEE Transactions on Services Computing*, 6(3):314–329, 2013.

[60] Wei Zhang, Sundaresan Rajasekaran, Shaohua Duan, Timothy Wood, and Mingfa Zhu. MIMP: Deadline and Interference Aware Scheduling of Hadoop Virtual Machines. In *14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid '14)*, Chicago, IL, USA, May 2014.

[61] Xi Chen, Lukas Rupprecht, Rasha Osman, Peter R. Pietzuch, Felipe Franciosi, and William J. Knottenbelt. CloudScope: Diagnosing and Managing Performance Interference in Multi-tenant Clouds. In *23rd IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, Atlanta, GA, USA, October 2015.

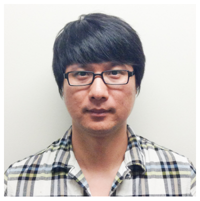
[62] Abhishek Gupta, Osman Sarood, Laxmikant V. Kalé, and Dejan S. Milojicic. Improving HPC Application Performance in Cloud through Dynamic Load Balancing. In *13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing (CCGrid)*, Delft, Netherlands, May 2013.

[63] Hongyi Ma, Liqiang Wang, Byung-Chul Tak, Long Wang, and Chunqiang Tang. Auto-tuning Performance of MPI Parallel Programs Using Resource Management in Container-Based Virtual Cloud. In *9th IEEE International Conference on Cloud Computing (CLOUD)*, San Francisco, CA, USA, June 2016.

[64] Ridwan Rashid Noel and Palden Lama. Taming Performance Hotspots in Cloud Storage with Dynamic Load Redistribution. In *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*, Honolulu, HI, USA, June 2017.

[65] Seyyed Ahmad Javadi and Anshul Gandhi. DIAL: Reducing Tail Latencies for Cloud Applications via Dynamic Interference-aware Load Balancing. In *IEEE International Conference on Autonomic Computing (ICAC '17)*, Columbus, OH, USA, July 2017.

In Kee Kim received a Ph.D. in Computer Science from University of Virginia in 2018. He is currently an Assistant Professor at the Department of Computer Science of the University of Georgia. His research areas include cloud computing, large-scale distributed systems, IoT/edge computing, and machine learning systems. He is a member of the IEEE.



Jinho Hwang is a Research Staff Member at IBM T.J. Watson Research Center. He received the Ph.D. from The George Washington University in 2013. He was with The George Washington University from 2005 to 2006 as a visiting scholar, and with POSCO ICT R&D center in South Korea from 2007 to 2010. He interned at IBM T.J. Watson Research Center and AT&T Labs-Research in 2012 and 2013 summers, respectively. He has published more than 50 papers, filed more than 50 patents, and has won 4 best paper awards. His research interests include cloud computing optimization, network virtualization centered around software-defined clouds to enable customers to adapt quickly to the heterogeneous cloud environments. He is a member of the IEEE.

Wei Wang holds a Ph.D. in computer science from University of Virginia in 2015. He is currently an Assistant Professor at the Computer Science Department of the University of Texas at San Antonio. His research interests include system software, cloud computing, computer architecture and software engineering. He is a member of the IEEE.

Marty Humphrey is an Associate Professor in the Department of Computer Science at the University of Virginia. He received a B.S. and M.S. degree in Electrical Engineering from Clarkson University in 1986 and 1989, respectively. He received his Ph.D. degree in Computer Science from the University of Massachusetts in 1996. From 1996-1998, he was an Assistant Professor of Computer Science and Engineering at the University of Colorado at Denver. From 1998-2002, he was a Research Assistant Professor at UVA. He has co-authored over 75 publications and has been a principal investigator on a number of projects funded through government agencies (such as the National Science Foundation and the Department of Energy) and private sector (such as Sun Microsystems and Microsoft Corporation).