Orchestra: Guaranteeing Performance SLAs for Cloud Applications by Avoiding Resource Storms

In Kee Kim
Computer Science
University of Virginia
ik2sb@virginia.edu

Jinho Hwang
T.J. Watson Research Center
IBM Research
jinho@us.ibm.com

Wei Wang
Computer Science
University of Texas at San Antonio
wei.wang@utsa.edu

Marty Humphrey
Computer Science
University of Virginia
humphrey@cs.virginia.edu

***Abstract*—This paper presents Orchestra, a cloud-specific framework for managing both foreground applications (e.g., Web, DBMS) and background services (e.g., backup, security check, batch jobs) in the user space. Orchestra is designed to address "*resource storms*" caused by sudden executions of the background services on the 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 and poor user experience. Orchestra takes an online approach using lightweight monitoring and creates performance models for multiple cloud applications on the fly. It then optimizes the allocations of shared resources to meet SLAs. We evaluate the performance of Orchestra on a production cloud (Amazon EC2) with a diverse range of SLA requirements. The experiment results show that Orchestra successfully guarantees the foreground application's performance to meet its SLA targets at all times. Moreover, Orchestra maintains the background's performance by minimizing its performance penalty with proper allocation of the shared resources.**

***Index Terms*—Cloud Computing; Resource Storms; Guaranteeing Performance SLA; Enterprise Cloud Management**

I. Introduction

Many enterprise cloud instances (e.g., Virtual Machines) often run both foreground applications (FGs) and background services (BGs) at the same time. The FG applications are latency-sensitive and user-facing applications like Web and DBMS. The BG services are executed to securely manage the cloud instances/data centers as well as to improve overall utilization/cost efficiency of cloud resources. These BGs include backup, security compliance, virus scan, patching, and batch tasks. Because the BG services frequently perform very critical operations for the management purposes, the BGs have to be executed as planned in many cases [1]. This requirement incurs *resource storms* that create high peaks of resource usage without knowing when the FGs need more resources. Such resource storms can retard processing time of FG applications and in turn the response time.

Figure 1 (blue-line) illustrates the performance (response time) degradation of a web application when the application co-runs with BGs. We observe that the tail latency (98%tile) of response time could show 36x slowdown. The degraded performance has a tremendous impact on the QoS of the FGs, resulting in frequent SLA violation and poor user experiences. For instance, Amazon has reported that every 100ms delay

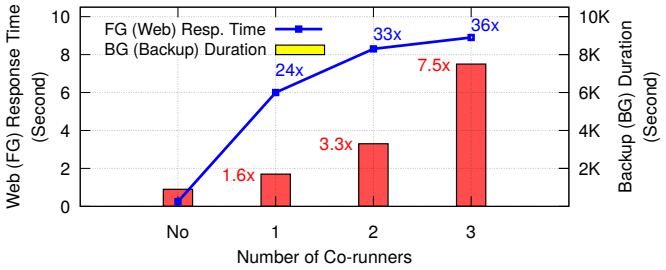

Fig. 1. Performance variation of FG and BG applications. The blue-line indicates the response time (98%tile) slowdown of a FG (Web) application response time when running together with BGs; The red-bar the slowdown of a 10G data backup duration as the FG workloads increase.

loses 1% of the sales profit [2], and video streaming (e.g., YouTube) users start abandoning videos after 2 seconds of buffering time [3].

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 [4, 5], so that SLAs of FGs suffer while parts of shared resources are consumed by BGs [6, 7]. OS modifications – *changing task priority* [8] or *designing a biased OS scheduler* [9] – have been proposed, but such tweaks are not feasible for normal users in the cloud environments due to the technical difficulties. Moreover, intuitive approaches – *terminating* or *suspending* the BGs – to guarantee the FGs' SLA are not sufficient in practice since particular BG tasks (e.g., backup and security checks) could have SLAs to finish the tasks due to the significance of such services [1]. As shown in Figure 1 (red-bar), the BG' execution time is also highly affected by the number of co-running 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 lead to under-utilization the cloud instances by overly controlling the BGs.

To solve this problem, we have created Orchestra, a framework for controlling the FG applications and BG services in the user space, aiming at meeting both SLAs. Orchestra

relies on an online approach with very lightweight performance/resource 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 [10, 11]. 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 [12]. Orchestra is complementary to widely used approaches for cloud application management. Orchestra components of performance monitoring and resource controlling offer finer-grained mechanisms than off-the-shelf monitoring/management tools like cloud auto-scaling[1] and CloudWatch[2] and help cloud users automatically determine *when* to scale.

We have implemented and evaluated Orchestra with real-world workloads on the production cloud. Our main workloads are a web service and a NoSQL database (MongoDB [13]) for FG applications, and backup (AWS Sync [14]) and virus/malware scanner (ClamAV [15]) 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.

We structure the rest of this paper as follows. Section-II reviews the state-of-the-art approaches that manage the performance of cloud applications. We then present the framework details of Orchestra in Section-III. In Section-IV, we evaluate Orchestra on a production cloud (AWS) with real-world FG and BG workloads. Finally, Section-V concludes this paper.

II. Related Work

A. Provider-centric approach

There are significant works from the research community to detect, prevent, and mitigate the resource storms/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-Cloud [16] predicts a SLA violation with a discrete time MIMO (Multi-Input and Multi-Output) model. DejaVu [17] employs a performance index that determines the interference by comparing with the identical executions on a sandbox. DeepDive [18] detects the performance interference with a warning system and manages it with VM cloning and workload duplication. Dirigent [19] 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 the safe task placement. Bubble-up [20] finds a safe co-location of multiple tasks on the same host with a sensitivity curve via offline profiling. Bubble-Flux [21] dynamically creates the sensitivity curve from online profiling with a short-term memory intensive workload. Paragon [22] performs minimal offline profiling for new workloads (e.g., 1 min. on two different HWs) and uses a collaborative filtering to place such tasks on the particular HWs. CPI^2 [4] 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 [23] finds a safe co-location of multiple tasks with a coordinated isolation mechanism of shared resources.

B. User-centric approach

There are several attempts to address this problem in the user space. IC^2 [24] and DIAL [25] mitigate the performance interference for web server clusters. The performance interference is detected by resource monitoring or statistic inference model. Once the interference is identified, the approaches change application configurations or reduce request flows to the low-performed web nodes. However, these approaches are application-specific (web service), but the current version of Orchestra supports diverse types of FGs. Moreover, Orchestra does not change any application configuration. Such modifications could result in high overhead and only mitigate a short-term performance interference [26]. Stay-away [27] 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, 28] and, more importantly, Orchestra is agnostic to user workload patterns.

III. Orchestra Framework

A. Orchestra Overview

Overall Architecture: We design Orchestra with a two-layer, distributed architecture, of *managed nodes* and a *master controller*. Figure 2 illustrates the architecture of Orchestra.

A **managed node** (VM instance) has two components in Orchestra – *sidecar* and *node agent*. The sidecar – *an online performance monitor* – is designed to watch performance variations 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.

The node agents are used for the following purposes;

- Monitoring the resource usage of the target applications.
- Monitoring the progress of BG's execution.
- Reconfiguring resources allocation for both FG and BG.

The monitoring of the resource usage focuses on collecting the 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.

[1]https://aws.amazon.com/autoscaling/

[2]https://aws.amazon.com/cloudwatch/

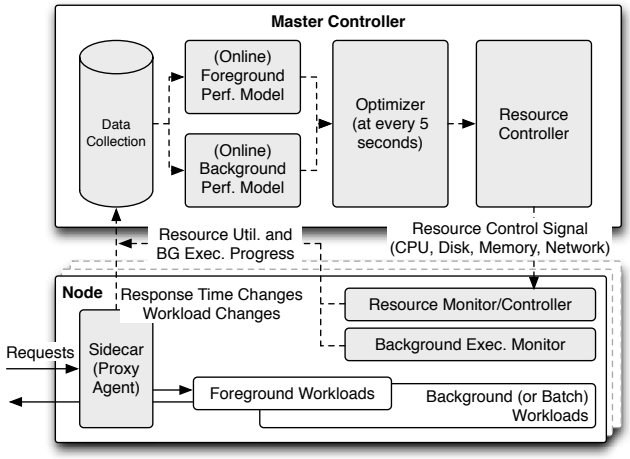

Fig. 2. Overall architecture of Orchestra

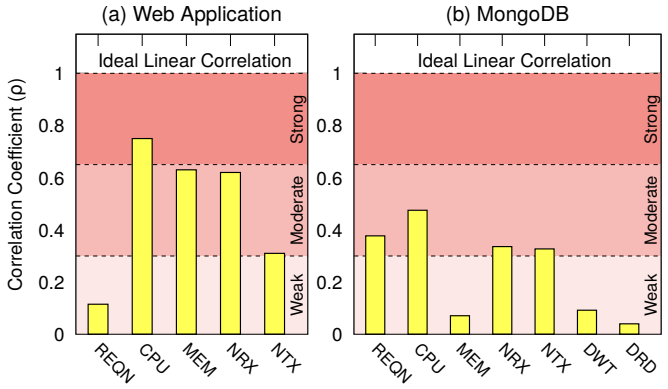

Fig. 3. Correlation coefficient of factors that could affect FG's response time (NRX: Network RX Bytes/sec. NTX: Network TX Bytes/sec., DWT: Disk Write Bytes/sec., DRD: Disk Read Byte/sec.)

The **master controller** plays the most important role in Orchestra framework 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-III-B) and *performance model* (Section-III-C) for both applications on the fly. With these models, the master controller optimizes the resource allocations to achieve the management goal (Section-III-D).

B. 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 response time or DB transaction time with a broad range of resource utilization.

Regarding the feature selection of the RT estimator, we observe the behaviors of two FGs (Web and MongoDB) by running benchmark tools (CloudSuite [29] and TPC-C [30]) without BGs' execution. We then calculate the Pearson Correlation Coefficient between the FG's RT and the resource metrics including the number of requests and various system resources – CPU, memory, disk and network IOs.

Figure 3 reports the measured correlation. Three factors show the highest correlation with the RT of the web application (FG) – CPU, MEM (Memory) and NRX (Network RX Bytes). The coefficients are between 0.6 and 0.75. In the MongoDB benchmark, the 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.

Moreover, we measure the correlation coefficient among these four factors since there could be a certain possibility that one factor can be correlated with other factors. i.e., a correlation of 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 I. CPU and Network IO are strongly correlated each other for both FGs. MEM is moderately (Web) or lightly (MongoDB) correlated with 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.

TABLE I
CORRELATION BETWEEN THE SELECTED FACTORS (1: WEAK, 2: MODERATE, 3: STRONG CORRELATION)

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

Based on the above observation, we chose MVPR (Multivariate Polynomial Regression) [10] 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:

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

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. Moreover, to simplify the model computation, we use a harmonic mean ($2/(1/NTX + 1/NRX)$) of NTX

and NRX, both representing network-IO statistics and it helps to reduce the number of equation terms.

C. 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 the resource allocation with an accurate prediction of difference resource usages. To create such a model, we consider the following BG services for this work.

- **ClamAV** [15] is an open-source anti-virus engine used to defend user instance (e.g., VM) from computer viruses, Trojan, and other malicious threats.
- **AWS Sync** [14] 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[3] storage.

To select features for the BG performance model, we first perform a profiling study on two different Amazon EC2 instances[4] – m3.medium and c4.large – of Ubuntu 16.04 LTS without FGs' executions. We also use two different datasets of 35G (10K files) and 106G (50K files) for this profiling study. 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 II. ClamAV is observed as a CPU and Disk-IO (Read) bound application and moderately consumes memory resources. Sync mostly consumes CPU, Disk-IO (Read) and Network (TX) resources on the instances. 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.

TABLE II
STATISTICS OF MEASUREMENT RESULTS FOR UNDERSTANDING BG APPLICATIONS' CHARACTERISTICS ON TWO EC2 INSTANCES

	(a) ClamAV		(b) Sync	
Dataset	**35G**	**106G**	**35G**	**106G**
CPU[5]	73.8%	86.5%	78.6%	95.2%
Memory	14.2% (0.53G)	19.5% (0 73G)	3.3% (0.12G)	3.9% (0.15G)
Disk Read	8.3 MB/s	40 MB/s	51 MB/s	133 MB/s
Disk Write	79 KB/s	301 KB/s	2.2 KB/s	159 KB/s
Network TX	-		55 MB/s	141 MB/s
Network RX	-		1.4 MB/s	3.7 MB/s

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, the BGs, like many other applications, support 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 4 represents the progress of file size and numbers scanned by ClamAV according to the BGs' execution. Compared to the ideal progress (black line in Figure 4), 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 (Mean Absolute Error)[6] and a harmonic mean[7] 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.

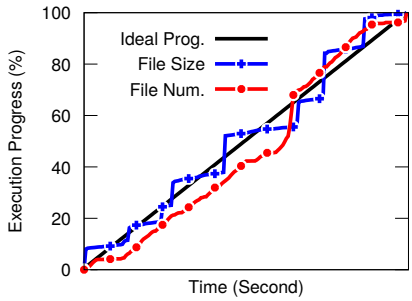

Fig. 4. Progress of scanned file size and numbers by ClamAV (Comparison with the ideal progress.)

With the above observation, we design the performance model with a multivariate linear regression [10] 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.

D. Orchestra Resource Optimizer and Controller

Orchestra determines the resource allocation to both applications with two predictive models described in the previous sections. The primary objective of this decision is to satisfy the FG's SLA, so the RT estimation model (Equation (1) in

[3]https://aws.amazon.com/s3/

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

[5]100% of CPU means the full usage of 1 vCPU.

[6]$|Progress_{ideal} - Progress_{log}|$

[7]$2/(1/file\_size + 1/file\_numbers)$

Section-III-B) should have the following condition. 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 I) 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) [19]. This estimation may result in slightly inaccurate prediction 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., auto-scaling) to ensure the 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 a third application 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.

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

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

E. Orchestra Implementation

The main components of Orchestra 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[8] 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 the resource monitoring and control. Sysstat[9] 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 [12] – *cpu* and *blk_io* – to control the CPU and disk 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).

The sidecar – a performance monitor for the FG – is based on Nginx's reverse proxy[10] and load-balancing[11] 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.

IV. 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. We then present how Orchestra minimizes the slowdown of BG's execution time. We further investigate both Orchestra's resource allocation for both the FGs and the BGs and the performance of dynamic resource control by Orchestra.

A. Evaluation Setup

Evaluation Infrastructure: We use general purpose m4[12] 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 [31–33], 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. Since spot instances have even higher level of the performance variance, multiple runs and averaging them offset the *variance*.

[8]https://istio.io/

[9]http://sebastien.godard.pagesperso-orange.fr/

[10]https://www.nginx.com/resources/admin-guide/reverse-proxy/

[11]https://www.nginx.com/resources/admin-guide/tcp-load-balancing/

[12]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.

FG Workloads: We consider Web application and MongoDB as representatives of FG workloads and use two different benchmarks for each FG to generate real workloads; CloudSuite 3.0 [29] - Web Serving benchmark and TPC-C [30] 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[13]) and back-ends – Memcached and DBMS – (m4.xlarge[14]) 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.

BG Workloads: A dataset with 5GBytes is used for BG workloads. i.e., ClamAV (virus scan) and AWS Sync (backup). This dataset has approximately 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-III-C for a fair comparison.

Performance Metric: In Section-IV-B, meeting a broad range of SLA targets is the primary metric for the FG control. 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-IV-B, we measure the slowdown of BG's execution time by:

$$BG\ Slowdown = \frac{BG\_Exec\_Time_{with\_FG\_Workload}}{BG\_Exec\_Time_{Standalone}} \quad (10)$$

where the denominator indicates a BG's execution time without co-runners and the numerator means the BG's execution time with FG workload.

B. Meeting FG's Response Time SLA under Resource Storms

This section focuses on the overall performance of Orchestra. The objective of this evaluation is meeting the SLA targets for FG applications by controlling the FGs' RT as close as the given SLA targets. The ideal case for this evaluation is that the FG's RT is the same as the SLA targets.

Before we start this evaluation, we recognize the range (the upper and lower bound) of SLA targets that Orchestra should meet. We quantify RTs (Table III) of three 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".

TABLE III
95%TILE RESPONSE TIME (RT) OF WEB AND MONGODB WITH AND WITHOUT RESOURCE STORMS FROM BGS.

	Number of BGs	Web	MongoDB
FG Only	0	0.92s	0.86s
FG + ClamAV	1	5.88s	2.71s
FG + Sync		8.65s	3.57s

[13]m4.large has 2 vCPUs, 8GB RAM, and SSD storage.
[14]m4.xlarge has 4 vCPUs, 16GB RAM, and SSD storage.

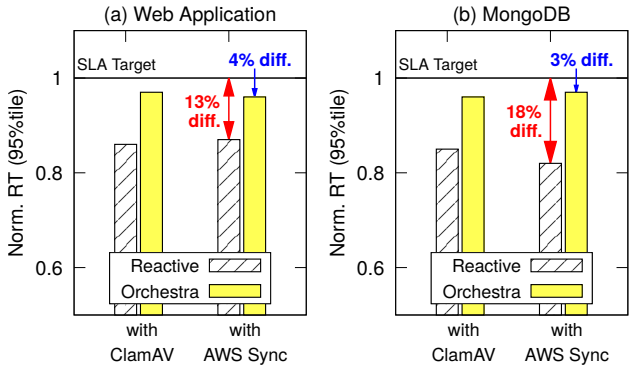

Fig. 5. Normalized RT (95%tile) of Web and MongoDB over a set of SLA targets (The best result should be 1.0.)

As the baseline, we use a reactive approach from other works [4, 21]. 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 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 5 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 5, both frameworks can successfully control the FGs' RT with the SLAs. But Orchestra's results are much closer to the SLA targets, and it only has 2 – 5% difference with the SLAs. However, the reactive system has about 15% differences with the SLA targets. These results imply that the FGs controlled by Orchestra maintain the application performance by consuming potentially less amount of resources as compared to the FGs with the reactive system.

C. Minimizing BG's Execution Slowdown

Next, we measure the slowdown of BGs' execution. As we confirmed in the previous evlation, Orchestra's FG control, *maintaining the FG's RT as close as the SLA target*, is beneficial to minimize the BGs' execution time. Orchestra allows the BGs to utilize more resources to boost their execution (more importantly, without SLA violations). Figure 6 reports the difference of the BG's execution time controlled by two approaches. While Orchestra only has 1.25x (with Web) and 1.39x (with MongoDB) slowdown (as expressed in equation-(10)) of the BGs' execution, the reactive approach

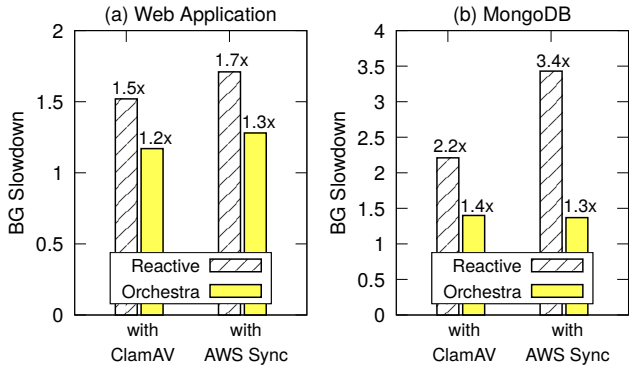

Fig. 6. Slowdown of BG's execution under two different control frameworks

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.

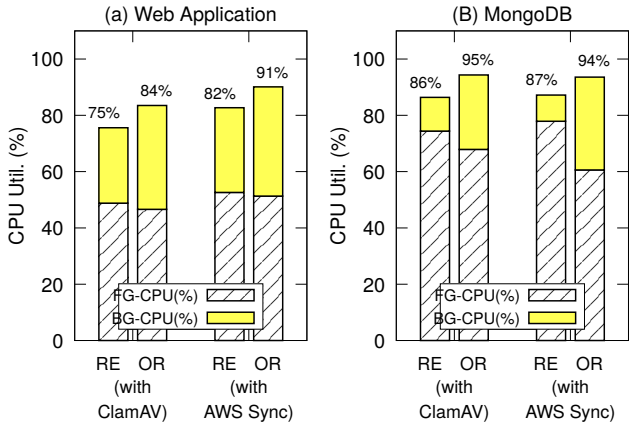

Fig. 7. Resource utilization (CPU) of VM with two control frameworks (RE: reactive framework, OR: Orchestra)

Figure 7 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 7(a)), Orchestra utilizes over 90% of CPUs, which is leveraging 10% more resources than the reactive approach. Interestingly, Orchestra uses the 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 7(b)), Orchestra improves CPU utilization over 95% and provides balanced resource allocations to the both, implying 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.

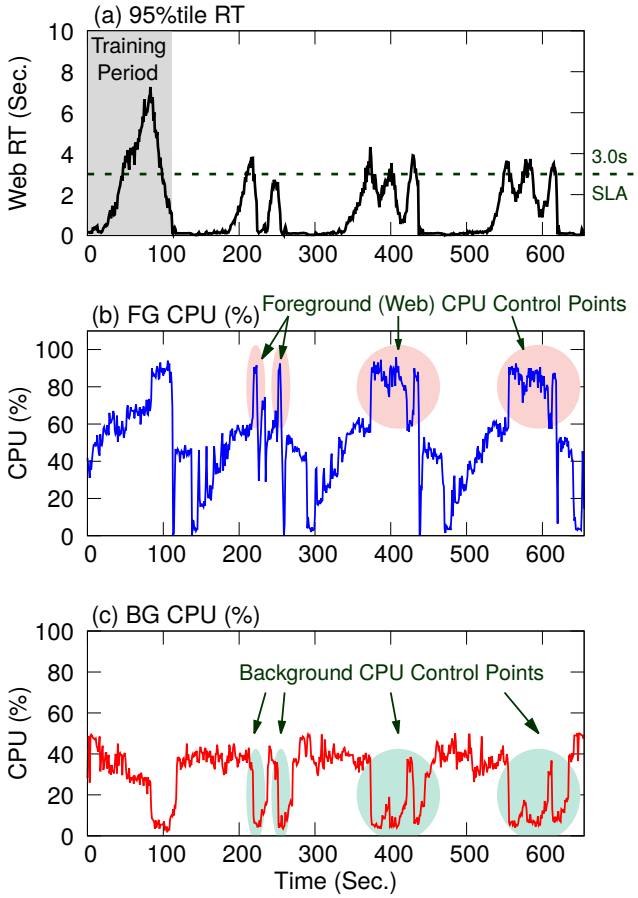

Fig. 8. Change of CPU resources for both a FG (Web application) and a BG (AWS Sync) from Orchestra and the changes of 95%tile RT of the FG

D. Orchestra's Dynamic Resource Control

Figure 8 illustrates Orchestra's behavior for the performance (RT) management of FG (Web Application) and dynamic control of VM resources. The top graph (Figure 8(a)) shows the changes in the FG's RT and the SLA target. The middle graph (Figure 8(b)) reports the dynamic CPU allocation for the FG, and the bottom graph (Figure 8(c)) shows the CPU allocation for the BG. As shown in the top graph (at the beginning of Orchestra control), while Orchestra controls the CPU resources based on its estimation mechanism, the most significant SLA violation happens in the first 100 seconds. However, this violation is expected and is mostly because of that the RT estimation model for FG does not have enough training dataset to build a robust FG RT estimator. After this initial training period, Orchestra performs the successful control in the CPU resources. The middle and bottom graphs 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 580second. Also, when the FG's RT is stable, Orchestra always allocates more resources to the BG to boost its execution.

V. 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. 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. The results show that Orchestra guarantees the FG's SLA satisfaction at all times with 70% performance improvement of BGs.

References

[1] George Amvrosiadis, Angela Demke Brown, and Ashvin Goel. Opportunistic Storage Maintenance. In *Proceedings of the 25th Symposium on Operating Systems Principles*, Monterey, CA, USA, October 2015.

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

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

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

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

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

[7] 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.

[8] OpenSUSE, Tuning the Task Scheduler. https://doc.opensuse.org/documentation/leap/tuning/html/book.sle.tuning/cha.tuning.taskscheduler.html, 2018. [ONLINE].

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

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

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

[12] CGROUPS. https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt, 2017. [ONLINE].

[13] MongoDB. https://www.mongodb.com/, 2018. [ONLINE].

[14] AWS sync. http://docs.aws.amazon.com/cli/latest/reference/s3/sync.html, 2018. [ONLINE].

[15] ClamAV. https://www.clamav.net/, 2018. [ONLINE].

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

[17] Nedeljko Vasic, Dejan Novakovic, Svetozar Miucin, Dejan Kostic, and Ricardo Bianchini. DejaVu: Accelerating Resource Allocation in Virtualized Environments. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '12)*, Bordeaux, France, December 2012.

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

[19] Haishan Zhu and Mattan Erez. Dirigent: Enforcing QoS for Latency-Critical Tasks on Shared Multicore Systems. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems*, April 2016.

[20] 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 *Proceedings of the 44th International Symposium on Microarchitecture (Micro '11)*, Porto Alegre, Brazil, December 2011.

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

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

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

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

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

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

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

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

[29] 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 *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '12)*, Bordeaux, France, December 2012.

[30] Andy Pavlo. Python TPC-C. https://github.com/apavlo/py-tpcc, 2018. [ONLINE].

[31] 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 *Proceedings of the USENIX Workshop on Hot Topics in Cloud Computing*, Boston, MA, USA, June 2012.

[32] Benjamin Farley, Ari Juelsand Venkatanathan Varadarajanand Thomas Ristenpartand Kevin D. Bowers, and Michael M. Swift. More for Your Money: Exploiting Performance Heterogeneity in Public Cloud. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC '12)*, San Jose, CA, USA, October 2012.

[33] 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.