QuMan: Profile-based Improvement of Cluster Utilization

YANNIS SFAKIANAKIS and CHRISTOS KOZANITIS, Foundation for Research and Technology–Hellas (FORTH), Greece
 CHRISTOS KOZYRAKIS, Stanford University, USA
 ANGELOS BILAS, Foundation for Research and Technology–Hellas (FORTH), Greece

Modern data centers consolidate workloads to increase server utilization and reduce total cost of ownership, and cope with scaling limitations. However, server resource sharing introduces performance interference across applications and, consequently, increases performance volatility, which negatively affects user experience. Thus, a challenging problem is to increase server utilization while maintaining application QoS.

In this article, we present *QuMan*, a server resource manager that uses application isolation and profiling to increase server utilization while controlling degradation of application QoS. Previous solutions, either estimate interference across applications and then restrict colocation to “compatible” applications, or assume that application requirements are known. Instead, *QuMan* estimates the required resources of applications. It uses an isolation mechanism to create properly-sized resource slices for applications, and arbitrarily colocates applications. *QuMan*’s mechanisms can be used with a variety of admission control policies, and we explore the potential of two such policies: (1) A policy that allows users to specify a minimum performance threshold and (2) an automated policy, which operates without user input and is based on a new combined QoS-utilization metric. We implement *QuMan* on top of Linux servers, and we evaluate its effectiveness using containers and real applications. Our single-node results show that *QuMan* balances highly effectively the tradeoff between server utilization and application performance, as it achieves 80% server utilization while the performance of each application does not drop below 80% the respective standalone performance. We also deploy *QuMan* on a cluster of 100 AWS instances that are managed by a modified version of the Sparrow scheduler [37] and, we observe a 48% increase in application performance on a highly utilized cluster, compared to the performance of the same cluster under the same load when it is managed by native Sparrow or Apache Mesos.

CCS Concepts: • **Software and its engineering** → **File systems management; Memory management; Information systems** → *Storage management*;

Additional Key Words and Phrases: Resource management, input/output, isolation, solid state disk, I/O caching, slices

ACM Reference format:

Yannis Sfakianakis, Christos Kozanitis, Christos Kozyrakis, and Angelos Bilas. 2018. *QuMan*: Profile-based Improvement of Cluster Utilization. *ACM Trans. Archit. Code Optim.* 15, 3, Article 27 (August 2018), 25 pages. <https://doi.org/10.1145/3210560>

We thankfully acknowledge the support of the European Commission under the Horizon 2020 Framework Programme for Research and Innovation through the VINEYARD (H2020-ICT-687628) project.

Authors’ addresses: Y. Sfakianakis, C. Kozanitis, and A. Bilas, Foundation for Research and Technology–Hellas (FORTH), N. Plastira 100, Vassilika Vouton, Heraklion, Heraklion 70013, Greece; emails: {jsfakian, kozanitis, bilas}@ics.forth.gr; C. Kozyrakis, Stanford University, 353 Serra Mall, Stanford, California; email: kozyraki@stanford.edu.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2018 Association for Computing Machinery.

1544-3566/2018/08-ART27 \$15.00

<https://doi.org/10.1145/3210560>

1 INTRODUCTION

Today, modern datacenters face a complex tradeoff while they manage applications and physical servers: On one hand, users require predictable performance with little variance in application execution. Therefore, operators overprovision and keep servers underutilized to avoid excessive interference across applications. For instance, the typical CPU utilization for private datacenters is approximately 12% [7, 49].

On the other hand, there is an increasing need for datacenter resources [17] to keep up with the data growth [12, 38, 45, 55]. In addition, technology limitations for scaling datacenters and the price competition among data center providers [27], dictate that operators increase server CPU utilization to reduce cost [21]. To achieve this, operators need to ensure that application QoS does not degrade due to interference, when the CPU utilization increases.

Recent work addresses interference by using two distinct approaches: (a) performance prediction and (b) resource isolation. Systems that use performance prediction, such as in References [13, 14, 20, 25, 29, 35, 36, 54] statistically predict or detect the effects of interference on the performance of each application. These systems either sample application runs in a sandboxed environment [13, 14, 35], or monitor their runtime execution [20, 29, 36, 54] or get performance metrics from historic data [25]. These approaches are limited by the complexity and accuracy of the involved models: Unless systems allow by design performance penalties [20], mispredictions cannot be tolerated, and they impose serious performance degradation to all colocated applications. To avoid problems caused by misprediction, these approaches conservatively colocate only applications with mutually exclusive resources, such as long running analytic jobs with latency-critical tasks, and thus they limit their applicability.

Other approaches, such as in References [11, 15, 22, 24, 26, 28, 32, 33, 53], avoid the complexity of predictions by employing resource isolation mechanisms to offer dedicated portions of the server resources to applications. To compensate for the absence of runtime profiling information, they typically use offline data to categorize applications in two to three dominant types for the cluster.

A major obstacle of both approaches, however, is the complexity of resource allocation. Utilization aware resource allocation needs to get combined with performance prediction in a centralized module that has to (1) understand the heterogeneous performance requirements of the applications they serve; (2) orchestrate the collection of task performance data by triggering sample runs for profiling, organizing performance counters, or looking up historic data; (3) understand the existing workload and isolation capabilities of worker nodes; (4) train complex machine-learning models to make predictions; and (5) produce utilization aware allocations. This complexity is also the reason why approaches with strong isolation mechanisms avoid online workload characterization by using offline data to characterize applications.

Thus, both approaches, today, are only applicable to simple workload compositions, where collocation decisions are easy. Utilization aware schedulers shine in the presence of tasks with evidently non-overlapping resource requirements. They simply match complementary tasks in terms of resource needs. They also handle workloads that consist of a mix of user facing (time critical) tasks and background running tasks. In that case, they simply make user facing tasks as first-class citizens and fill any slacks with non-time-critical tasks that run in the background. However, the existence of such workload types in most datacenters is not always guaranteed. Either the resource requirements of tasks are not 100% complementary or the priorities of running tasks do not allow the separation between first-class citizens and task fillers.

Cluster utilization aware resource allocation of more generic workload compositions, where possible task collocations do not affect the performance of each other uncontrollably, further increases the complexity of centralized schedulers. Instead, we advocate that decentralized scheduling allows the decoupling of challenging performance prediction from the resource allocation

module. In this way, the complexity of performance prediction moves to worker nodes, and existing resource allocation modules need only minor modifications to optimize for the CPU utilization of the cluster.

For worker nodes to support such a decentralized mechanism, they must be enhanced with the following three features:

- (1) Strong isolation guarantees: A decentralized scheduling mechanism only needs to know whether a particular task fits in a worker node, not how it interferes with other tasks.
- (2) Performance prediction capabilities: Given that the reason of decentralized scheduling is the pressure that performance prediction places on a central resource allocator, worker nodes must implement such a functionality to facilitate the decoupling of performance prediction from scheduling decisions.
- (3) A simple, but generic, API that allows applications to communicate their performance constraints: Such an API is not trivial as different frameworks can express their requirements with different metrics, such as latency, throughput, or cycles per instruction.

To the best of our knowledge, this work is the first to provide decentralized resource allocation that aims to increase the CPU utilization of the cluster, while providing performance guarantees to applications. The main vehicle of such a decentralization is the development of sophisticated cluster nodes that can calculate the exact fraction of resources that a submitted task needs to use to meet its performance guarantees. Therefore, we introduce *QuMan*, a node resource manager and admission controller for applications scheduled to a cluster node; *QuMan* increases server utilization and introduces policies that either allow users to provide application thresholds of acceptable QoS degradation or they automatically make decisions using a combined reward function of utilization and QoS for the entire workload.

QuMan consists of three main components and works as follows:

- (1) A software-based isolation mechanism that provides an interference-free execution environment to applications, using slices. Each slice receives a portion of the server that includes CPU cores, main memory, and the I/O path, which consists of I/O memory and SSD caches. The isolation mechanism provides applications with access only to specific slices of each resource. The slices are NUMA aware in terms of memory and CPU node location, which results into a reduction in the interference across per-core CPU caches. Of course, NUMA awareness does not guarantee CPU cache isolation, which along with the lack of network isolation are left for future work.
- (2) A profiler that performs on a dedicated server controlled runs of applications to determine the resource requirements of newly admitted applications. Unlike other systems that use profiling, *QuMan* ignores the interference dimension across applications because of its isolation mechanism. Therefore, it only needs to analyze the resource requirements of individual applications and not combinations of them, which is a significantly simpler problem to model and solve. Our profiler is deployed on the side of regular infrastructure and its overhead can vary from a few to at most one thousand seconds depending on the application and its input size. Profiling runs concurrently with the first execution of an application, and its results are available to subsequent invocations of the same application, without the overhead of the multiple runs.
- (3) An admission controller that employs different policies, to balance CPU utilization and application QoS. In this work, we explore two policies: a user-oriented and an operator-oriented. Our user-oriented policy, similar to other systems [14, 28, 29, 35, 49], ensures that application execution does not violate a user-provided performance threshold

Fig. 1. *QuMan*'s flowchart for assigning resources to application and performing admission control.

compared to stand-alone execution. The operator-oriented policy optimizes a new metric that we introduce, which is a reward function of both server CPU utilization and average QoS across all applications. Note that although the mechanisms of *QuMan* make no assumptions on the nature of the dominant resource of tasks, the operator-oriented policy uses the CPU usage as a proxy to measure server utilization.

We deploy *QuMan* on Linux servers, where we evaluate its effectiveness using a mix of real applications: Apache, TPC-E, TPC-H, and BLAST. We show that, depending on the admission control policy, *QuMan* achieves a server utilization of 80% while it maintains performance levels of running applications above 60% of their standalone performance. Then, we integrate *QuMan* with a utilization aware modification of Sparrow [37] decentralized scheduler. We show that the execution of a heavy workload in an Amazon cluster of 100 instances that are managed by our modified version of Sparrow improves the aggregate *Performance Index* of the workload by 48% compared to the execution of the same workload on the same resources that are managed either by the native version of Sparrow or by Apache Mesos [23].

2 QUMAN DESIGN

This section describes the main components of *QuMan*: (a) The Framework for Isolation of Memory, I/O path, CPU and SSD cache (*FRIMICS*), which is a mechanism that ensures isolation across applications (Section 2.3), (b) a profiler, which identifies resource requirements for each application (Section 2.4), and (c) an admission controller, which admits applications to a server based on specific policies (Section 2.5).

2.1 Overview

Figure 1 shows a flowchart of how *QuMan* assigns server resources to applications. *QuMan* considers that all incoming applications have equal priorities and admits them as First Come First Served (FCFS). Applications can choose a performance threshold that satisfies their execution. Note that we do not enforce any restrictions on the nature of the performance metric, because *QuMan* is going to convert it into a *Performance Index* in its downstream analysis (Section 2.2). For each application, *QuMan* decides whether there are enough resources to accommodate its performance requirements, and if so, it allocates a proper slice of *FRIMICS* to the application. Otherwise,

it drops the application and it notifies the resource allocator about the failure. *QuMan* determines the performance behavior of all submitted applications using profiling.

The *profiler* either fetches statistical models that predict the performance of known applications, or creates them by executing the application in different hardware configuration settings. The first time that a profiler encounters a particular application, it submits it together with its original input into a sandboxed node to perform controlled runs with different resource configurations and collect the necessary datapoints. Then it produces a performance prediction function, which (i) it exposes to the admission controller, and (ii) it stores along with the application profile, so future submissions of the same applications do not repeat sandboxed runs. Note, that the profiling is expected to work well with short tasks, which dominate commercial datacenter workloads [3, 10, 31]. However, a non-negligible part of the workloads include also applications with long tasks, which put pressure on the profiler, because they need unrealistically a lot of time for profiling. *QuMan* follows the best practices of literature [14] and alleviates this pressure by profiling only a small part from the beginning (first 100s) of long tasks. This keeps profiling overheads to a minimum for tasks that take hours or days to complete.

When the profiler returns the performance prediction function, the *admission controller* of *QuMan* uses it to determine on what hardware configuration the application should run, depending on the desired policy of the cluster operator. If, for example, operators prefer to run applications while they respect their performance requirements, then they can follow the user-oriented policy (Section 2.5). If, however, they prefer to maximize the server utilization in a performance aware manner, they can use the provider-oriented policy. The provider-oriented policy optimizes an index that describes the performance-utilization tradeoff, called QUCI, and in the presence of workload changes it dynamically adjusts resource allocation of applications to keep QUCI optimal. *QuMan*'s mechanisms can handle those requests due to the dynamic nature of *FRIMICS*'s slices. The application suffers a penalty on its performance during the reallocation procedure; however, *FRIMICS* guarantees that the penalty is contained only to that application.

2.2 Performance Index

We define as *Performance Index* of an application the slowdown in its performance, when it executes on a portion of server resources compared to the stand-alone execution. Regardless of what performance metric users are interested in, be it latency, throughput, or execution time, *Performance Index* is a number without units and in the range between 0 and 1. To extract *Performance Index*, *QuMan* requires to know the metric that users are interested in and the file that reports it. This definition of *Performance Index* assumes that an application achieves its optimal performance when it runs on all available resources, even if it does not use all resources, which is typically the case. We acknowledge that there are applications that may slow down as resources increase, e.g., as the number of CPU cores increases. However, we expect that these applications should apply self-throttling mechanisms and avoid scaling to more cores than what they have been designed for, and that separate mechanisms should detect such behavior.

As an example of *Performance Index* calculation consider a web server application, where either throughput or latency can be used as a performance metric. In the case of throughput, the *Performance Index* is defined as the requests/second the web server achieves with a resource allocation, divided by the requests/second it achieves when allocating all available resources. In case the latency is the desired performance metric, the *Performance Index* is defined as the latency the web server achieves with all available resources, divided by the latency it achieves on a portion of the server resources.

Performance Index, as a function of the amount of each resource, typically takes the three possible shapes of Figure 2: linear, convex, or concave. For example, applications that are highly concurrent

Fig. 2. The three main patterns we assume for application *Performance Index*.

with little synchronization, such as Machine Learning training models that are known to scale linearly or sub-linearly [8, 47] may follow the linear curve. Applications that exhibit some sort of working set behavior with respect to memory may follow the concave curve, since less memory affects performance, but more memory does not help beyond a certain point. For example, assume that an application with a working set of 1MB receives an 8MB cache. For cache sizes greater than 8MB, the effect on its performance is minor until its cache allocation reduces below 1MB. The effect of further reductions can be quite small at first, but dramatic later on. Less memory affects performance, but more memory does not help beyond a certain point. Finally, the *Performance Index* of applications that are sensitive to SSD cache size may follow the convex curve; the more the cache size reduces, the more page faults access the disk and severely affect the performance.

2.3 FRIMICS

FRIMICS uses *cgroups* [39] for CPU and application memory isolation and a modified version of Vanguard [43] to isolate the I/O path. Vanguard statically partitions the I/O path of a server and isolates the access to I/O caches (memory and SSD), I/O buffers, allocator and control structures that include synchronization. *QuMan* extends Vanguard in two ways: (a) It provides a mechanism for dynamic slice creation, resizing, and deletion, and (b) it converts I/O buffer allocation mechanism to be NUMA aware. Future versions of *QuMan* will also isolate the LLC and the network.

The *cgroups* mechanisms provide two ways for limiting CPU usage by a process: *relative* and *absolute*. The *relative* mechanism uses the *cpu.shares* parameter that specifies the percentage of CPU offered to a *cgroup*, relative to the *active* set of *cgroups*, which is the set of *cgroups* with a running process. The underlying mechanism of *cgroups* divides the CPU in 1,024 shares and depending on the value of *cpu.shares* for each *cgroup* and the number of *active cgroups*, it allocates to it a portion of the remaining shares. The *absolute* mechanism allows users to define explicitly the access period of the CPU and the total time that a process takes. However, the libraries that implement this method induce significant overhead. For this reason, *FRIMICS* uses the relative approach.

FRIMICS combines the *share*-enabled dynamic CPU allocation of *cgroups*, with their static memory allocation, by controlling their NUMA placement. To map I/O buffer allocation to slices, *FRIMICS* uses CPU shares in combination with CPU masks. The CPU masks limit the CPU assignment only to specific NUMA nodes that are compatible with the I/O buffer allocation. In addition, *QuMan* ensures that the total shares of the running applications will be at most 100. *FRIMICS* converts the allocated CPU of each application, directly to CPU shares. For example, if an application requires 50% CPU, then it will get 50 CPU shares instead. Although this allocation policy is not equivalent to statically allocating CPU cores to applications, it is similar, however, on a highly loaded server.

Indirect LLC Isolation: FRIMICS does not explicitly isolate CPU caches (specifically the LLC) and memory bandwidth; however, it mitigates the interference in those resources by placing applications to a single NUMA node, when there are enough resources available, or uses neighboring NUMA nodes when applications are larger. Therefore, it indirectly reduces interference in per-core caches and directly traffic across different NUMA nodes, which can significantly improve I/O throughput up to $2\times$ [30].

Dynamic Slice Creation and Modification: FRIMICS accepts requests to create new slices with a certain amount for each resource: CPU cores, memory, I/O buffers, and SSD cache. When deleting a slice, FRIMICS flushes all pending requests from DRAM I/O cache and SSD cache to the underlying storage before it frees resources. Resizing a slice follows the same steps with slice delete and create. Finally, FRIMICS offers the ability to assign a new or re-assign a used slice to an application.

Memory Allocation: An implication of the memory allocation mechanism of FRIMICS is that it handles separately kernel I/O buffers, both in terms of placement and usage. However, *cgroups* do not distinguish between application memory and kernel I/O buffers during allocation and they include both types of memory with a single limit for each container. Therefore, when FRIMICS allocates memory for slices, it uses the sum of both types of memory in the *cgroups* limit.

2.4 Profiler

For a newly submitted application, the profiler estimates and provides to admission controller policies the following two functions:

- (1) A *Performance Index* prediction function that takes as input a hardware configuration, expressed as fractions of a server's CPU, memory, and IO.
- (2) A CPU utilization function that takes as input a hardware configuration and predicts the expected CPU utilization of the configuration. Note that the CPU utilization differs from the CPU allocation and it is usually lower. The CPU allocation corresponds to the maximum CPU utilization the application can achieve. It will match the allocation only if (1) the application has constant CPU demand throughout its execution and (2) the profiler predicts CPU requirements with 100% accuracy. Thus, the CPU utilization function acts as a proxy that measures the existence of performance bottlenecks in various allocations. Although a CPU is not always a key performance driver, its use in comparison to allocated CPU provides valuable insights as to the existence of any performance bottlenecks regardless of the resource of origin. For example, a memory-sensitive application will have page faults in the absence of enough memory, and this will affect the CPU utilization significantly.

The profiler performs a number of sample runs for each new application. It performs these runs in a dedicated server and in parallel to actual application execution. Hence, the profiling does not affect the scheduling of QuMan. This procedure will take from a few to at most a thousand seconds depending on the application and its input size. Afterwards, it feeds the results into a statistical model to predict how the application scales up. The profiler uses slices of increasing resources in each dimension separately to create a training set of data points for the *Performance Index* model. Note that the profiler is agnostic to application input parameters. For example training a machine learning model for 100 iterations is completely independent from training the same model with 200 iteration, and thus the two combinations get profiled independently. However, we do not re-profile applications if they execute on similar sized input datasets.

Initially, for each newly admitted application, the profiler uses a dedicated server to perform controlled runs on the user provided sample input on slices with different resources. It constructs

Fig. 3. The three possible ways that an application's performance change when the hardware resources on which it executes reduce. Note, that although the figure displays a single-dimensional resource for demonstration purposes, our modeling uses a three-dimensional space, as it is not straightforward how the combination of these resources affects application performance.

a four-dimensional representation of the *Performance Index*. Each run provides two data points, each of which is a tuple (X, Y) . $X \in \mathbb{R}^3$ is a vector that contains the configuration in terms of CPU, memory size, and SSD cache size of the slice on which the application ran. $Y \in \mathbb{R}$ is the measured *Performance Index* and CPU utilization respectively for each datapoint. For example, consider profile runs of BLAST on a server that consists of 8 CPU cores, 16GB of memory, and 32GB of SSD cache. If a run that uses all available resources takes 19s and causes 93.7% CPU utilization, then it will produce datapoints $\langle 8, 16, 32 \rangle, 1$ for *Performance Index* and $\langle 8, 16, 32 \rangle, 93.7$ for CPU utilization. If another run on the same server of the same software uses 4 CPU cores and same amount of memory and SSD, then it finishes in 39s and causes 87% CPU utilization (of those cores), it will produce datapoints $\langle 4, 16, 32 \rangle, 0.49$ for *Performance Index* and $\langle 4, 16, 32 \rangle, 87$ for CPU utilization.

Next, the profiler fits the observed datapoints for each metric, *Performance Index* and CPU utilization, with two S-shaped logistic functions. We use the logistic function for two reasons: First, it simplifies the expression of *Performance Index* as a function of the available resources. An S-shaped logistic curve, as shown in Figure 3, consists of three distinct areas, each resembling one of the three patterns we assume for the shape of the *Performance Index* (Figure 2). Therefore, an S-shaped curve models all cases, with different parameters. Second, CPU utilization is expected to follow the shape of *Performance Index* for applications whose performance follows Figure 2. The two curves for an application do not necessarily have the same shape, for example one can be convex and the other concave, but this is not a problem as we consider two distinct fitting functions.

Also note, that although *FRIMICS* provides resource isolation on more than three resources, the logistic regression of the profiler uses only three resources as features. There is a tradeoff in using more resources in the profiler; a possible increase in the number of features of the logistic regression might produce more accurate models, but it will require exponentially more datapoints to avoid underfitting. Given that every datapoint requires sample runs, and to keep the profiler simple both simple and accurate, we use only three features: the number of CPU cores, memory capacity and SSD cache capacity.

The logistic function that we use to fit the observed datapoints (X, Y) , is defined as:

$$f^{\mathfrak{g}}(X) = C \frac{1 - e^{-X\beta}}{1 + e^{-X\beta}}, \quad (1)$$

where C is scalar, β is the vector $(\beta_1, \beta_2, \beta_3)$, and $\mathfrak{g}$ all parameters (C and β) of each function.

Estimating C and β : QuMan uses the Newton–Raphson method to estimate C and β for the logistic function using the observed datapoints in the training set. Newton–Raphson finds the parameters that minimize the mean square error as given by

$$e(\vartheta) = \sum_{m=1}^M \left(Y_m - f^{\vartheta}(X_m) \right)^2, \quad (2)$$

where $\vartheta = (C, \beta)$, f^{ϑ} is the individual characteristic function, M denotes the total number of input points, and $f^{\vartheta}(X_m)$ is the estimated *Performance Index* or utilization value for the same resources.

The profiler identifies the parameters ϑ that minimize total error:

$$\hat{\vartheta} = \underset{\vartheta}{\operatorname{argmin}} e(\vartheta). \quad (3)$$

In our experiments, Newton–Raphson typically converges in less than 3000 iterations and it takes less than one second to run. Also the space overhead of storing the model parameters is negligible; for each application, the profiler only stores the four model parameters: $[C, \vartheta_1, \vartheta_2, \vartheta_3]$.

2.5 Profile-based Admission Controller

Our admission controller uses the prediction functions that the profiler provides to make decisions on whether to admit an application to the server and how to size its *FRIMICS* slice. In this work, we explore two different policies:

- (1) A user-oriented policy, based on QoS thresholds. This policy allows users to specify a minimum threshold for the *Performance Index* for each application, which the admission controller will meet. In our initial investigation, and for simplicity, we use the same threshold for all applications running on a server.
- (2) A provider-oriented policy driven by a new metric, QoS-Utilization Combined Index (QUCI). This policy does not require user input as it automatically balances QoS and utilization.

Both policies treat tasks independently, and optimize for task level performance metrics. Of course, one could improve this approach and build more sophisticated algorithms that also consider dependencies among tasks. However this does not change the core functionality of QuMan, which trades in a controlled manner performance guarantees of tasks for higher server utilization.

User-oriented Policy: The user-oriented policy allows users to provide a minimum performance threshold that they are willing to tolerate. QuMan is agnostic to the performance metric, because it converts all metrics to the *Performance Index*. The admission controller creates a *FRIMICS* slice for the application with configuration settings for each resource that meet the requested threshold.

The controller uses the *Performance Index* prediction function of the profiler to examine different slice configurations with combinations of the available resources. In case the controller detects multiple possible hardware configurations that provide the desired *Performance Index*, it chooses the one with the least amount of aggregate resources (as percentages). For instance, if the application can achieve the same performance with an assignment of either 35% CPU, 30% memory, and 25% SSD cache (35% + 30% + 25% = 90%), or 45% CPU, 40% memory, and 15% SSD cache (45% + 40% + 15% = 100%), then QuMan prefers the former. If QuMan finds multiple resource allocations with the same sum, then it randomly selects a configuration that fits server, without further resource prioritization.

Provider-oriented Policy: The provider-oriented policy assigns resources to applications automatically using only their profiling information. There are two fundamental requirements that we satisfy: Maximize server utilization while minimizing consolidation related effects on performance.

To accommodate the former requirement, our policy favors applications with high utilization levels of their offered resources. If, for example, applications A and B achieve the same *Performance Index* with the same slice configuration, but application A underutilizes its offered resources, then the policy reduces the number of resources that are available to A and assigns them to B.

However, to meet the second design requirement, the policy also favors applications with higher *Performance Index*. If, for example, applications C, D work on the same slice configuration with similar device utilization, and a new application F demands a slice that will need to “steal” resources from either C or D, then the policy will take resources from the application with the lowest possible effect on its performance.

As a proxy for the two requirements, the policy uses the two functions that the profiler provides: the *Performance Index* prediction and the CPU utilization prediction. Thus, the metric that this policy uses to quantify both requirements of the design is the *QoS-utilization combined index* (QUCI). We define QUCI for a workload with k applications as

$$QUCI(X_1, \dots, X_K) = \sum_k w_{PI} f_{PI}^{\theta_k}(X_k) \times w_{CPU} f_{CPU}^{\theta_k}(X_k), \quad (4)$$

where f_{PI} and f_{CPU} are the profiler functions that estimate the *Performance Index* and the CPU utilization for each application, w_{PI} and w_{CPU} are weights that specify the significance of the respective values in the metric, K is the total number of running applications, and k is the k th running application.

The policy maximizes QUCI under the constraints of available resources. It uses the *Lagrange multipliers* method to maximize the QUCI *reward function*, as described below:

$$(\hat{X}_1, \dots, \hat{X}_K) = \underset{(X_1, \dots, X_K)}{\operatorname{argmax}} QUCI(X_1, \dots, X_K), \text{ where } \sum_{k=1}^K (X_k) = 1, \quad (5)$$

X_k is a three-dimensional vector that represents slice configuration in terms of CPU, memory size, and SSD cache size of the server resources offered to applications. The *Lagrange multipliers* method is very quick and in our experiments it requires at most 100ms and on average 40ms.

QUCI increases when both *Performance Index* of applications and the CPU utilization of the server increase. For example, if we consider two workloads, one with a single application running at 1 *Performance Index* and utilizes 30% of its CPU allocation, and one with three applications, each of which runs with a *Performance Index* of 0.8 and utilizes CPU by 20%, and if we assume for simplicity equal weights of 1 in the QUCI equation, the QUCI of the first workload is 0.3 while the QUCI of the second workload is $3 \times (0.8 \times 0.2) = 0.48$. Thus, the admission controller will favor the second workload.

3 IMPLEMENTATION

We have implemented *QuMan* in 10,000 lines of C++ and python code. It runs on Linux and manages any binary that executes on the server. We then integrated *QuMan* with Sparrow cluster scheduler [37] to demonstrate its capabilities in a cluster environment.

3.1 Integration with Sparrow

Sparrow [37] is an open source resource scheduler that achieves high scheduling throughput for tasks. To schedule a task, Sparrow probes a number of randomly chosen worker nodes and assigns it to the node with the fastest response time.

Fig. 4. Resource negotiation steps of QuSparrow. (1) QuSparrow queries randomly a number of *QuMan* nodes whether they could run a task under a certain performance constraint. (2) If *QuMan* nodes do not know the performance profiles of the task, then they will request from the sandbox to perform profile runs. (3) The sandbox will notify one of the *QuMan* workers about the profiling results, and the later will broadcast those results to all workers in step 4. (5) *QuMan* nodes will decide whether the task fits in their workload, and if it does, they will notify the QuSparrow master about how many resources they will have left, after they admit the task. (6) The QuSparrow master will select the node that expects to be utilized the most, after it admits the task, to proceed with the execution.

We introduce a modified version of Sparrow, QuSparrow, which uses *QuMan* to achieve utilization aware scheduling. We changed the front end of Sparrow API to allow tasks to submit performance constraints that *QuMan* needs to satisfy, which afterwards get converted into *Performance Index*. Figure 4 shows the architecture of this integration, which uses a user-oriented policy for admission control (see Section 2.5). For every scheduling decision, QuSparrow sends task execution requests to a number of randomly chosen worker nodes that run *QuMan*; workers whose admission controllers can accept the task, respond (step 5 of Figure 4) with the amount of remaining resources they will have available *after* they accept the task, while the rest respond with negative numbers. QuSparrow collects the responses of all nodes and greedily assigns the task to the node with the *minimum* remaining resources (step 6 of Figure 4).

If *QuMan* nodes know the performance profile of a submitted task, then they will skip steps 2, 3, and 4 of Figure 4 and calculate directly their estimates; otherwise, they will submit that task together with the original input to a sandbox node to perform profile runs. For sandboxed executions, we use a spare node of the cluster. Given that it is the workers that submit tasks for profile runs, and QuSparrow probes multiple workers, the sandbox receives multiple requests to profile the same task (step 2 of Figure 4). The sandbox executes profile runs once, in response to the request that arrives first, and it ignores the rest requests for the same task. In case the sandbox is busy profiling a particular task and receives requests to profile different tasks, it adds those requests in a queue and serves them as soon as it finishes with the former. Finally, in step 4, the *QuMan* node that receives the profiling results from the sandbox broadcasts them to the rest nodes

of the cluster. Thus, whenever the same task gets submitted in the future at any node, they will omit steps 2, 3, and 4, as all of them will have synchronized lists of known profiles.

For the purposes of this integration, we have made two working assumptions. First, profile runs occur rarely and do not interfere with the scheduling latency goals of Sparrow. This is a reasonable assumption, since most tasks tend to appear with recurring patterns [25, 40]. Second, production cluster trace analyses shows that short tasks are dominant in datacenters, for example statistics from the Google datacenter, where 92% of the jobs have tasks that execute in less than a minute [41], Mishra et al. [31] report “tasks with short duration dominate the task population,” Lu et al. [3] report that the average task duration is 192s with maximum task duration of 29585s. Thus, we expect QuMan to perform well in the majority of realistic datacenter workloads.

Long Tasks: Although QuSparrow is built under the assumption that most tasks are short, it takes a shortcut when handling long tasks that appear occasionally. The sandbox of *QuMan* considers a task as *long* when the first run (a run with all available resources), takes longer than 100s to complete [41]. After 100s, it measures the CPU, memory utilization, and SSD utilization and then kills the task. The sandbox returns only one datapoint to the *QuMan* worker, which represents the *Performance Index* of 1 with the server utilization readings it took at the 100th second. In that case, the profiler of the *QuMan* worker assumes that *Performance Index* drops linearly with offered resources (form 2 of Figure 2) and for that long task it calculates a line that declines with 45° . Although such a shortcut might introduce error in the accuracy of long task profile estimation, it will increase the availability of the sandboxes to short tasks. Furthermore the effect of that error is insignificant, as the long tasks do not have strict service level objectives.

Data Locality Skew: A practical issue of the sandboxed runs is the skew on the profiling results from data locality. Although the measurements from the first run will contain data transfer associated overheads, subsequent runs will not have the same overheads as data will be already cached. To produce consistent profiling results, *QuMan* sandboxes flush their caches before initiating each run. Thus, all profiling related datapoints include data transfer related overheads.

Non-recurring Applications: Although typical datacenter workloads consists mainly of recurring tasks, they also include a significant portion of non-recurring applications. For this reason *QuMan* immediately schedules “unknown” applications with a predefined slice, which consists of 50% of a server’s resources. This strategy favors application performance over CPU utilization, because the “unknown” applications get a large slice to execute to ensures high performance; however, they might reserve more resources than necessary that can lead to decreased CPU utilization.

4 EVALUATION METHODOLOGY

4.1 Experimental Platform

We perform the single node evaluation on a server equipped with a quad-socket Tyan FT48-B8812 motherboard, with three 16-core AMD Opteron 6200 64-bit processors running at 2.1GHz and 48GB of DDR-III DRAM. For storage, we use four 32GB (128GB total) Intel X25-E SSDs as cache and two 1TB Western Digital Caviar Black as storage. The server runs Linux kernel v.3.10 (part of the CentOS distribution, v.7). All native experiments use the XFS filesystem with 4KB blocks. The I/O scheduler we use in all experiments is the default *noop elevator*.

For our distributed experiments, we deployed QuSparrow on a cluster of 100 AWS instances of type m4.xlarge. We also used an extra instance of the same type as a client that issues tasks and another instance as a sandbox for profiling.

4.2 Benchmarks

TPC-E [34] is a transactional workload that emulates the operations of a stock broker. We use 24 threads that issue transactions over a 49GB database. This workload consists of comparably localized randomly distributed small-sized I/O accesses, 10% of which are writes [9]. The metric we focus on is the transaction rate. The CPU load in our runs is approximately 50% of our system's CPUs, and a single SSD is sufficient to sustain the throughput requirements. TPC-E is latency sensitive and I/O bounded due to its transactions.

TPC-H [34] is a data-warehousing benchmark, generating business analytics queries to a database of sales data. We execute query Q5 in a loop using a 6.5GB database. We use as performance metric the average execution time for 20 consecutive executions of query Q5. We use this query, because it is both CPU and IO intensive.

We use Apache [2] and the *ab* benchmarking tool as our web-content serving application. We issue *ab* requests in batches with a duration of 10s. Each batch contains 100 *ab* instances that run in parallel, where each instance issues 5 concurrent requests. Our 8GB dataset is comprised of three different file classes with sizes 32KB, 256KB, and 1MB. The file on which *ab* operates is chosen randomly at the start of each batch, with a probability of 0.5, 0.3, and 0.2, respectively for each of the three file classes. Although using *ab* as described above produces significant I/O traffic, the read throughput of one of our SSDs is sufficient to sustain this load. The Apache server is latency sensitive so the disk latency and the available CPU resources are vital for its performance. The metric we report for this workload is the 95th percentile of request latency.

FIO [6] is a microbenchmark that stresses the I/O path by issuing concurrent streams of I/O requests to either files or block devices. The metric reported for FIO is the aggregate I/O throughput.

BLAST [5] is an application from the domain of genomics. We use the *blastn* program (v.2.2.27) that performs queries on nucleotide databases. In our setup, an instance of *blastn* issues concurrently queries to 16 separate databases. When databases are preloaded into memory, *blastn* heavily utilizes both the CPU and the I/O subsystem. The total size of all databases is 23GB. The metric we capture as application performance is cumulative execution time for all concurrent queries.

4.3 Workload Mix

In our single node evaluation, we assume that the server has a FIFO queue of applications that is ready to run and an external entity, e.g., a cluster resource allocator, places applications in this queue. In all scenarios the server admits one application at a time until it runs out of resources. In case available resources do not suffice to admit new applications, the queue stalls until one or more applications finish and release resources. For each application the server launches a docker container on top of a *FRIMICS* slice with the appropriate executables and data. In our runs, we select randomly a mix of 19 application instances from TPC-E, TPC-H, Blast, FIO, and Apache to form the initial queue.

In our AWS cloud deployment, a client creates with varying rates TPC-H task requests and measures task scheduling time, task completion time and it also monitors the utilization across the cluster. We ignore the results of the very first execution for each task, because they involve sandboxed profile runs, which we ignore according to our assumptions in Section 3.1.

5 COMPONENT EVALUATION

In this section, we present our results for (a) the isolation mechanism of *QuMan*, (b) its profiling approach, and (c) the two policies we examine.

Table 1. Slice Configurations for TPC-H, Apache, TPC-E, BLAST while They Run Concurrently and Each Receives a Guaranteed *Performance Index* of 0.8

Application	CPU(#cores)	Mem(GB)	SSD(GB)
TPC-H	2	6	8
Apache	16	14	24
TPC-E	12	10	16
BLAST	16	14	24

Fig. 5. Application QoS and server utilization for Vanguard vs. *FRIMICS*.

5.1 *FRIMICS* Isolation Mechanism

First, we examine the benefits from the extended isolation mechanism of *QuMan*. We compare Vanguard versus *FRIMICS* to quantify the impact of CPU isolation, application memory isolation, and mitigation of NUMA effects to application performance, in addition to I/O path isolation. As an interfering workload mix, we run concurrently four applications, Apache, TPC-H, TPC-E, and Blast on separate slices. Each application receives a slice whose configuration guarantees a *Performance Index* of 0.8 in the absence of any interference. Table 1 shows the exact hardware configuration that each application receives. Figure 5(a) presents the *Performance Index* over time for each application for both Vanguard and *FRIMICS*. We see that the latter improves the average *Performance Index* by 37.9%, where each application shows an individual improvement between 21.6% to 63.9%.

According to Figure 5(b), it is the better isolation mechanism of *FRIMICS* that leads to such a performance improvement. Compared to Vanguard, *FRIMICS* mitigates even more application interference, because it isolates more resources. Therefore, it further hedges the degradation of performance for consolidated applications that essentially leads to better application performance.

5.2 Number of Profiling Runs

In this section, we examine the number of samples that are required for accurate predictions by the profiler of *QuMan*. For this reason, we exhaustively search all possible combinations of CPU (48 cores), memory (42GB), and SSD cache (32GB) and we measure the *Performance Index* of each application as they execute individually in slices of all possible resource configurations. We collect 70,000 different datapoints for each application, and we use half of them as a training set in the curve fitting function and the rest for testing. Figure 6 shows the error rate, which we calculate

Fig. 6. Impact on the accuracy of the profiler as we increase the number of training points. The figure displays the first 50 points, because the error converges afterwards.

Fig. 7. Impact of profiler misprediction on application *Performance Index* as the second, mispredicted TPC-H instance is admitted at 150s from the experiment start.

using Function 2, as a function of the number of points that we consider during curve fitting. For clarity, we plot only the first 50 datapoints. The profiler accuracy improves until the size of the training set reaches 10 samples and then it converges to within 10% error for all remaining configurations.

5.3 Misprediction Penalty

An advantage of *QuMan* is its resilience to mispredictions, due to the isolation mechanism that it uses. For the purposes of this experiment, we ignore the profiler recommendations, and we deplete resources from an application on purpose. We choose a very small slice for the mispredicted application, because it causes more pressure to *FRIMICS*, but in practice the impact to a mispredicted application is much lower, because the profiler's mispredictions have a 10% volatility compared to the targeted *Performance Index*. In our setup, the correctly predicted applications are TPC-H, Apache, TPC-E, and BLAST, each allocated a slice adequate to ensure a minimum *Performance Index* of 0.75. The slice configuration for each application is the same as in Table 1.

After 150s of execution, a new TPC-H instance arrives and, to emulate misprediction, we assign to it a slice that consists of 2 cores, 4GB of memory, and 8GB of SSD cache. Observe, that this configuration uses 2GB of memory less than what is required to obtain a *Performance Index* of 0.75 for TPC-H. This, mispredicted configuration, is reasonable, given that the offered resources in combination with the aggregate resources of the running applications do not exceed the server capabilities (46 cores, 44GB of memory, and 72GB of SSD cache).

Figure 7 plots the *Performance Index* for each application during the first 300s of the execution. We note that although the performance impact on the new application is severe, the rest, properly

Table 2. *Performance Index (PI) and CPU Utilization for the User-oriented and Provider-oriented Policies*

Policy	PI	Util(%)	QUCI
User-oriented (1)	1	17	0.16
User-oriented (0.75)	0.79	52	0.36
User-oriented (0.5)	0.59	68	0.38
Provider-oriented	0.57	79	0.45

The user-oriented policy is configured with thresholds of 1, 0.75, and 0.5.

predicted workloads, remain relatively insensitive. This is due to the stronger isolation offered by *FRIMICS* compared to other approaches.

At this point, we also observe that there are some short time periods in Figure 7, whose duration does not exceed 5% of the total execution time, where the observed PI drops below *QuMan*'s threshold. There are three reasons that explain that behavior: (1) The application behavior is not constant, whereas the profiler assumes it is. (2) *FRIMICS* isolation is not ideal and does not completely eliminate interference with co-located applications. (3) Profiles are not 100% accurate.

5.4 Policy Evaluation

For the evaluation of the user and provider driven admission policies, we experiment with the admission of an application queue that contains a random order of 7 TPC-H, 6 BLAST, 4 TPC-E, 1 Apache, and 1 FIO instances. We execute this queue 4 times; each time we allocate slices to applications using a different policy: three runs follow the user-oriented policy with thresholds of 1, 0.75, and 0.5, respectively, and one run uses the provider-oriented policy. We also calculate QUCI for each run to see how QUCI ranks each experiment. In the provider-oriented policy, we assume equal values of 1 to the weights of the QUCI metric (Equation (4)).

Table 2 summarizes for each policy the average *Performance Index* across all applications and the average CPU utilization. The results show that the user-oriented policy stays above the threshold and penalizes CPU utilization accordingly: The higher the threshold required by the user, the lower the achieved CPU utilization. The provider-oriented policy automatically picks a higher utilization data point and achieves a similar, overall *Performance Index* for about 16% higher CPU utilization (79% vs. 68%). Table 2 also shows that QUCI is higher (as expected) in the provider-oriented policy. More interestingly, QUCI increases as the user-specified threshold is reduced in the user-oriented policy, which drives CPU utilization up significantly.

6 OVERALL SINGLE SERVER EVALUATION

To measure the behavior of *QuMan* in realistic scenarios, we submit to a server jobs from a queue, under the constraint that *Performance Index* does not drop below 0.8. *QuMan* keeps creating new slices for the incoming jobs, until 1 or more resources saturate.

Figure 8 shows the execution history of the same workload we use in Section 5.1 and different slice configurations. Each execution uses fix-sized slices, ranging from a small number of “fat” to a larger number of “thin” slices. Possible slice configurations consist of either 2 or 8 slices. Each slice uses an equal portion of each resource available in the server, e.g., 8 slices get 12.5% of the available CPU, memory size, and SSD cache size. Combining profiling information with the isolation mechanism allows *QuMan* to achieve 50% average CPU utilization with 0.84 application *Performance Index*. In addition to the CPU utilization, we show the SSD utilization to observe other

Fig. 8. Comparison of PI (left), CPU utilization (middle), and SSD Utilization(right) using profile-derived slices vs. fixed-size slices. The profile-derived slices follow the user-oriented policy with threshold 0.8.

Fig. 9. Comparison of PI (left), CPU utilization (middle), and SSD Utilization (right) using optimal user 0.8 vs. *QuMan* with user-0.8 mode vs. *QuMan* with provider mode vs. uncoordinated baseline.

components of the system as well. The fact that SSD utilization increases as well is a consequence of the increased CPU utilization and the system doing more work.

Apart from comparisons with baselines of 2 and 8 slices, we show how *QuMan* compares to an optimal case (Figure 9) and an uncoordinated case (where we run all applications simultaneously with no control). For the optimal case, we emulate a system that implements a perfect profiler, on zero interference and policy of *QuMan* with user-oriented policy with threshold 0.8. We use the training set to choose the most appropriate resources for each application of the workload, and we run them in isolation (stand-alone), one by one. We consider that applications execute in parallel until we exhaust the resources of the server. For example suppose that we have 28 cores, and our applications are BLAST, TPC-E, Apache, BLAST, and TPC-H, that need 16, 12, 16, 16, and 2 cores correspondingly. BLAST and TPC-E fit in the server (they need 28 cores), therefore they execute together and the rest wait. BLAST will finish first, because it takes 35s while TPC-E takes 150s, which means that at the 35th BLAST frees 16 cores and Apache will start running because it fits, and so on. *QuMan* operating with User-0.8 mode achieves average CPU Utilization 50% and average Performance Index 0.84. *QuMan* with provider mode achieves average CPU Utilization 80% and average Performance Index 0.6. Optimal configuration that emulates the best User-0.8 achieves average CPU Utilization 70% and average Performance Index 0.89. What we observe is that *QuMan* achieves comparable performance to the optimal with a small drop in CPU Utilization. However, if we are willing to trade performance for CPU utilization (provider-oriented policy), we significantly improve utilization.

7 MULTI-NODE EVALUATION

In this section, we compare QuSparrow versus Sparrow and Mesos on a cluster of 5 Intel servers with 32 cores and 256GB of memory. A client generates a high load for this cluster by submitting 10 jobs/s for 100s, and the minimum *Performance Index* is set to 0.8. Figure 10 shows on the left the average *Performance Index* and on the right the average CPU utilization of active jobs. As we observe from Figure 10(b), all three systems operate at high load and they all achieve same levels of

Fig. 10. Performance Index (left) and average CPU utilization (right) of five servers running tasks through QuSparrow, Sparrow, and Mesos.

Fig. 11. Performance Index (left) and average CPU utilization (right) of 100 AWS instances running tasks through QuSparrow and Sparrow.

cluster utilization (approximately 65%). QuSparrow manages to maintain an average *Performance Index* of 0.90, while Sparrow and Mesos achieve a *Performance Index* of 0.65 and 0.60 respectively; in other words, QuSparrow increases the average *Performance Index* by 42% and 50% when compared to Sparrow and Mesos, respectively.

7.1 Cloud Deployment

In our large cluster deployment, a client submits jobs at a rate of 2 jobs per second for 300s and all worker nodes run with a *QuMan* configuration that set the admission controller policy to *user-oriented* with a minimum *Performance Index* of 0.8.

In this experiment, and to achieve uniform load across all servers, we show a case with a heavy load to keep all servers highly utilized at steady state. The cluster achieves a steady workload state after 100s, where both systems have similar CPU utilization.

Figure 11(a) shows the average *Performance Index* of QuSparrow in comparison to the *Performance Index* of Sparrow as they run on a 100-node cluster. As the load of the cluster increases, QuSparrow maintains the average *Performance Index* consistently above 0.8 while workload interference affects the average *Performance Index* of tasks under Sparrow as it drops down to 0.4.

To measure the impact on server utilization of QuSparrow, Figure 11(b) shows the average server utilization across the cluster, where in the case of QuSparrow it drops by 40% compared to Sparrow. QuMan achieves better *Performance Index*, because it manages to “pack” applications better.

8 DISCUSSION

Profiling Overhead: *QuMan* requires about 10 sample runs that last from a few seconds to 1,000 (due to the 100s cap per task). As the sandboxed runs take place only the first time an application is encountered, we do not expect this overhead to be an issue. In addition, this profiling happens concurrently with the first time the application executes. Of course, there is room for further reduction to the number of required samples in future developments, where the profiling can benefit from heuristics, such as optimal design space [47] to reduce the time that a profiler spends for each application.

Profiling vs. Execution Correctness Trade Off: A possible performance optimization of QuSparrow would be the elimination of sandboxing. In this scenario, given that profile runs occur with the actual inputs in a dedicated node, one could argue that QuSparrow could avoid steps 5 and 6 of Figure 4 as it already executed the same task multiple times. However, despite the obvious performance gains, such an approach could be harmful for applications' state. As *QuMan* performs multiple profiling runs, it overwrites the same output files multiple times, and thus there are consistency related risks that need to be considered. To avoid those risks, the current version of QuSparrow avoids this optimization.

Profiling Limitations: The profiler does not handle well applications with discretized performance, (e.g., video encoding can have bitrates of 1080P, 720P, 480P, etc.). Such applications are modeled better using a step function that differs significantly from the logistic function we use. A solution to this issue is to consider each piece of the step function as a different application. For example, 480p video encoding is one application, 720p encoding is a second, 1,080p is a third, and so on. This technique will convert the step function into a few step functions of one piece, which are modeled more accurately by the logistic function.

Isolation Limitations: The isolation mechanism of *FRIMICS*, deals only with limited resources at this point and at varying degrees. *QuMan* does not deal directly with LLC, TLB, memory bandwidth, network (memory, PCI, external). There has been work that shows the significance of interference on these resources. However, even with the incomplete isolation achieved by *FRIMICS*, *QuMan*, is still able to improve the tradeoff utilization-QoS. And indeed, there is room from improvement by more and better isolation as well as improved profiling.

Resizing Overheads: Another overhead of *QuMan* is the resizing of *FRIMICS*'s slices. Resizing commands cause increased I/O traffic as *FRIMICS* flushes dirty pages in the resized slices, which can take significant amount of time in case of large SSD caches. However, these overheads do not appear in the creation of new slices, it is up to the admission control policies to decide the level of resizing given its overheads. The policies that we implement in this article do not use resizing, and thus our results do not contain those overheads. We leave as future work the design of admission control policies that use resizing and cover the respective overheads in their metrics.

Admission Controller Policies: In this article, we make a first step towards different admission controller policies that can significantly affect application performance and server utilization. We do not make an effort to find the best policy or to extensively explore different dimensions in policies, and we leave these for future work.

Admission Controller Overhead: Both user and provider driven admission policies incur negligible overhead to the admission controller. The user driven policy queries 2–3 times the *Performance Index* prediction function of the profiler and it evaluates whether there are enough available server resources to support the desired threshold of *Performance Index*. However, the provider driven policy requires a few (less than 10) matrix calculations of size proportional to the number of the profiles. The controller takes at most a few hundred milliseconds for both policies.

Scheduling Latency of QuSparrow: In all fairness, the utilization gains of QuSparrow come at a cost of scheduling latency, as the latency of QuSparrow is about 10ms on average while Sparrow takes 7ms. QuSparrow, therefore, adds 3ms, because it waits for all workers to respond, plus it assigns an application to the most proper worker. We do not include profiling overheads in the scheduling overhead of QuSparrow, because QuSparrow workers do not wait for the profile of a “first time” application before it starts running it. We developed QuSparrow only to evaluate the utilization and isolation goals of *QuMan* in large-scale experiments; therefore, we leave as a future work the deployment of a scheduler that also optimizes the scheduling latency.

Scheduling of Jobs with Different Priorities: We note that the underlying assumption behind the design and evaluation of *QuMan* is that all jobs have equal priorities. We leave for future work the implementation of policies that accommodate tasks of jobs of different priorities. *QuMan*, deals with applications in an FCFS manner and does not deal with priorities. *QuMan* solves the problem of improving the utilization-performance tradeoff, and we assume that all tasks have equal priorities. It can be extended to look at priorities by using multiple queues (one for each priority), but this does not change the mechanisms discussed in this article.

Managing Datacenter Heterogeneity: In our evaluation, we use machines of the same generation that include similar hardware. Typical datacenters include three different server generations [13], which would require to either specify a generation for each application or to profile each application as many times as the generations of servers that might be used.

Handling Distributed Applications: Our intention is to handle distributed workloads that is why we ported *QuMan* to Sparrow cluster manager. The idea is that distributed workloads will keep using the APIs of the cluster manager that we use, which in our case is QuSparrow, a modified version of Sparrow; distributed frameworks, such as the Apache Spark, have been ported over Sparrow. In case of Apache Spark workloads, the profiler will profile each parallel task once. In the example of Spark over Sparrow, if a map stage produces 1,000 tasks, all of whom contain the same code that will run in different partitions, Sparrow will profile that code once and will use that profile to allocate resources to the rest tasks.

9 RELATED WORK

There are two categories of systems that assign resources to applications. The first category involves systems that assign resources to applications based on user requests. This generation includes systems, such as, Mesos [23], Omega [42], and Yarn [46]. They receive requests and they offer cluster resources to running applications. However, applications tend to overestimate their resource requirements and thus, hardware remains underutilized [14].

QuMan belongs to the second category of resource management systems, whose main goal is to increase resource utilization by workload consolidation, while they minimize effects on application execution. Cluster management systems of the second category colocate applications using either, or both, of two approaches: (a) resource isolation and (b) performance prediction. Systems that predict performance consider either, or both, of two distinct dimensions: (1) impact of offered resources and (2) impact of interference.

In the rest of this section, we review systems that aim to improve server utilization; we omit systems that focus on other dimensions, such as fairness and latency.

9.1 Performance Prediction

Paragon [13] and Q-Clouds [35] identify applications compatible for colocation. They perform small-scale interference tests between each application and controlled levels of background

applications for each resource. Then they use complicated multi variable statistical classifiers to predict the expected interference among applications at the time of colocation. Quasar [14] uses similar techniques to predict the impact of interference, but also the impact of scale up and scale out. However, mispredictions are costly on all these systems, because they affect all running applications. This, in combination to the complexity of the statistical predictions leads those systems to be conservatively selective over what applications to colocate and they favor pairs of applications with almost mutually exclusive resource requirements. However, *QuMan* has a higher degree of freedom over the type of applications that colocates because of its isolation mechanism. Thus, in *QuMan* (1) statistical modeling is simpler as it uses fewer variables and (2) mispredictions do not affect running applications.

Morpheus [25] operates on production environments that mostly execute similar tasks at the same time every day. This recurring pattern enables it to prepare off line resource allocations that are optimized for the jobs it expects. *QuMan* also assumes that the majority (around 70%) of tasks are recurring and stores the profiling information of each task to avoid unnecessary profiling runs. For the rest 30% of the workload it operates reactively and performs online profiling to the non-recurrent tasks.

Carbyne [20] uses job metadata and it “steals” resources that YARN allocates to jobs to reschedule them elsewhere. Carbyne relies on fair scheduling to offer isolation guarantees, but it does not avoid completely the unpredictable consequences of collocation interference.

Deepdive [36] uses hardware counters to monitor continuously a server for interference among colocated Virtual Machines (VMs). When it detects high interference that can seriously degrade the performance of running applications, it migrates a VM to reduce interference in the server using predictive placement. However, VM migration itself can entail significant overhead. For example, the scripts that launch Amazon EC2 VMs to demonstrate the Apache Spark software stack on small collections of data take between 20 and 40 minutes to run [1]. Instead, the isolation mechanism of *QuMan* mitigates the impact of mispredictions and hence does not need VM migrations.

Bubble-Up [29] uses profiling via “bubble” to predict precisely performance degradation across workloads. Bubble-Flux [50] enhances this approach to online interference and QoS management and uses a dynamic bubble technique to probe servers for resource pressure, prior to taking colocation decisions. Bubble-Up and Bubble-Flux target latency-sensitive workloads that have a specific QoS target. *QuMan* uses a more general notion of QoS, defined relative to stand-alone execution. Bubble-Up and Bubble-flux rely on detecting (offline or online) and avoiding interference via probes and appropriate colocation decisions, whereas *QuMan* avoids interference by creating dedicated, properly-sized slices for each container, as estimated with application profiling.

PAC [18] and its successors PRESS [19] and CloudScale [44] identify resource demands online by statistically modeling application performance whereas, *QuMan*, first, mitigates interference using *FRIMICS*, to reduce mispredictions, and, second, it performs a controlled offline training phase, rather than estimating resource demands on the fly.

There are also systems, which although they combine small-scale profiling runs with Statistical Inference, they solve different problems than resource allocators and resource assigners. Ernest and CherryPick [4, 47], for instance, are advisory tools that assists users of a *single* application to minimize the number of instances they provision in the Cloud. *QuMan*, however, is a cluster manager that serves multiple workloads from multiple users, estimates the resources they need, and assigns them in a way that maximizes the number of applications that can run concurrently in clusters.

9.2 Resource Isolation

Another group of systems provide to applications isolated access to system resources.

Heracles [28] provides isolated hardware resources to workloads and it monitors application performance at runtime. It isolates hardware with respect to cores, memory, network, and L1 CPU cache. Heracles treats performance critical workloads with high priority and offers them enough resources to meet their deadlines, while it allocates remaining server resources to batch workloads. Heracles assumes certain workload characteristics, typical for Google workloads, and shines at workloads that consist of both latency critical and batch applications. The profiling mechanism of *QuMan*, however, enables it to serve more diverse workloads.

9.3 Performance Prediction and Resource Isolation

Ubik [26] takes the first steps towards offering resources in isolation, while it also considers profiling information. It reduces the tail latency by providing to applications isolated access to LLC. *QuMan*, however, while not isolating access to LLC, it isolates more resources, compared to Ubik.

MemGuard [51] provides memory access isolation to applications, such that the average memory access latency or request is not larger than on a dedicated memory system. MemGuard, does not use the predictor to identify the appropriate resources for applications as *QuMan*, instead it requires an external user-level daemon to configure it.

ARIA [48] and Jockey [16] implement a profiler-based resource allocation technique for MapReduce environments. Each job comes with a completion deadline that is its SLO. Their profiler extracts job profiles that estimate the expected execution time given the number of resources allocated for the job. ARIA implements a job scheduler to meet as many SLO as possible by reordering the execution of jobs, whereas Jockey monitors the progress of each job and dynamically releases resources from jobs that have not pressure in meeting their SLO and reallocates them to jobs that might miss their SLO. *QuMan* implements a similar mechanism for profiling, with the difference that it can model any performance metric. Additionally, ARIA and Jockey are evaluated only for MapReduce, and we do not know how they perform on other frameworks.

Haoyu Zhang et al. [52] also propose profile-based scheduling for live streaming video queries. They base their scheduling on quality and lag goals of videos. They highlight the main challenge in video streaming that is how to choose the right values for multiple knobs to achieve the requested video quality. They implement two profiling phases one that is offline and one that is online. The profiler in the offline phase identifies a handful of knob configurations that satisfy the resource-quality of queries. The scheduler in the online phase allocates resources only among these configurations. *QuMan* faces a similar challenge, because multiple allocations can lead to the same performance. Haoyu Zhang et al. finds a good allocation and subsequently searches for mathematically equivalent ones, whereas *QuMan* uses a heuristic to find equivalent allocations.

10 CONCLUSIONS

Server utilization in modern data centers has emerged as an important challenge due to both cost, power and technology limitations. In this article we propose a new approach that increases server utilization in a cluster, while it keeps under control the degradation of application performance. We presented *QuMan*, a system that consists of (1) an isolation mechanism in the Linux kernel, (2) a user-space profiler, and (3) an admission controller, on which we implemented two admission policies. We evaluated *QuMan* with real workloads, and our results show the effectiveness of *QuMan*: The user-oriented policy allows explicit control over minimum QoS and results in CPU utilization of 52% and 68% for QoS targets of 75%, 50%, respectively. The provider-oriented policy achieves a QoS-utilization balance of 57%–79%. Finally, the *QuMan* profiler overhead is small after collecting the samples in the training set, samples are collected asynchronously once per applications, and mispredictions affect only one application.

ACKNOWLEDGMENTS

We are thankful to Yannis Pantazis and Eleni Kanellou for their comments on mathematical aspects of our model; Thodoris Zois for implementing *QuMan* on Mesos; and Christi Symeonidou, Anastasios Papagiannis, and Manolis Marazakis for their overall comments.

REFERENCES

- [1] Launching a Spark/Shark Cluster on EC2. Retrieved from <http://ampcamp.berkeley.edu/exercises-strata-conf-2013/launching-a-cluster.html>.
- [2] The Apache HTTP Server. Retrieved from <http://httpd.apache.org>.
- [3] Omar Arif Abdul-Rahman and Kento Aida. 2014. Towards understanding the usage behavior of Google cloud users: The mice and elephants phenomenon. In *Proceedings of the 2014 IEEE 6th International Conference on Cloud Computing Technology and Science (CloudCom'14)*. IEEE, 272–277.
- [4] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. 2017. CherryPick: Adaptively unearthing the best cloud configurations for big data analytics. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI'17)*. USENIX Association, 469–482.
- [5] Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Myers, and David J. Lipman. 1990. Basic local alignment search tool. *J. Molec. Biol.* 215, 3 (1990), 403–410.
- [6] Jens Axboe. Flexible I/O Tester. Retrieved from <https://github.com/axboe>.
- [7] Robert Birke, Lydia Y. Chen, and Evgenia Smirni. 2012. Data centers in the wild: A large performance study. Technical Report Z1204-002. IBM Research – Zurich. Retrieved from [http://domino.research.ibm.com/library/cyberdig.nsf/papers/0C306B31CF0D3861852579E40045F17F/\\$File/rz3820.pdf](http://domino.research.ibm.com/library/cyberdig.nsf/papers/0C306B31CF0D3861852579E40045F17F/$File/rz3820.pdf).
- [8] Leon Bottou and Olivier Bousquet. 2007. The tradeoffs of large scale learning. In *Proceedings of the 20th International Conference on Neural Information Processing Systems (NIPS'07)*. Curran Associates Inc., 161–168.
- [9] Shimin Chen, Anastasia Ailamaki, Manos Athanassoulis, Phillip B. Gibbons, Ryan Johnson, Ippokratis Pandis, and Radu Stoica. 2011. TPC-E vs. TPC-C: Characterizing the new TPC-E benchmark via an I/O comparison study. *ACM SIGMOD Rec.* 39, 3 (2011), 5–10.
- [10] Yanpei Chen, Sara Alspaugh, and Randy Katz. 2012. Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads. *Proc. VLDB Endow.* 5, 12 (2012), 1802–1813.
- [11] Henry Cook, Miquel Moreto, Sarah Bird, Khanh Dao, David A. Patterson, and Krste Asanovic. 2013. A hardware evaluation of cache partitioning to improve utilization and energy-efficiency while preserving responsiveness. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA'13)*. ACM, New York, NY, 308–319.
- [12] Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y. Ng. 2012. Large scale distributed deep networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems (NIPS'12)*. Curran Associates Inc., 1223–1231.
- [13] Christina Delimitrou and Christos Kozyrakis. 2013. 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)*. ACM, New York, NY, 77–88.
- [14] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: Resource-efficient and QoS-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)*. ACM, New York, NY, 127–144.
- [15] Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N. Patt. 2010. Fairness via source throttling: A configurable and high-performance fairness substrate for multi-core memory systems. In *Proceedings of the 15th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV)*. ACM, New York, NY, 335–346.
- [16] Andrew D. Ferguson, Peter Bodik, Srikanth Kandula, Eric Boutin, and Rodrigo Fonseca. 2012. Jockey: Guaranteed job latency in data parallel clusters. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys'12)*. ACM, New York, NY, 99–112. DOI: <http://dx.doi.org/10.1145/2168836.2168847>
- [17] Daniel Gmach, Jerry Rolia, Ludmila Cherkasova, and Alfons Kemper. 2007. Workload analysis and demand prediction of enterprise data center applications. In *Proceedings of the 2007 IEEE 10th International Symposium on Workload Characterization (IISWC'07)*. IEEE Computer Society, 171–180.
- [18] Z. Gong and X. Gu. 2010. PAC: Pattern-driven application consolidation for efficient cloud computing. In *Proceedings of the 2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*. 24–33.

- [19] Zhenhuan Gong, Xiaohui Gu, and J. Wilkes. 2010. PRESS: PRedictive elastic resource scaling for cloud systems. In *Proceedings of the 2010 International Conference on Network and Service Management*. 9–16.
- [20] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. 2016. Altruistic scheduling in multi-resource clusters. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*. 65.
- [21] James Hamilton. 2010. Overall Data Center Costs. Retrieved from <http://perspectives.mvdirona.com/2010/09/overall-data-center-costs>.
- [22] Andrew Herdrich, Ramesh Illikkal, Ravi Iyer, Don Newell, Vineet Chadha, and Jaideep Moses. 2009. Rate-based QoS techniques for cache/memory in CMP platforms. In *Proceedings of the 23rd International Conference on Supercomputing (ICS'09)*. ACM, New York, NY, 479–488.
- [23] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI'11)*. USENIX Association, 295–308.
- [24] Ravi Iyer, Li Zhao, Fei Guo, Ramesh Illikkal, Srihari Makineni, Don Newell, Yan Solihin, Lisa Hsu, and Steve Reinhardt. 2007. QoS policies and architecture for cache/memory in CMP platforms. In *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'07)*. ACM, New York, NY, 25–36.
- [25] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Íñigo Goiri, Subru Krishnan, Janardhan Kulkarni, and Sriram Rao. 2016. Morpheus: Towards automated SLOs for enterprise clusters. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*. 117.
- [26] Harshad Kasture and Daniel Sanchez. 2014. Ubik: Efficient cache sharing with strict Qos for latency-critical workloads. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)*. ACM, New York, NY, 729–742.
- [27] Eugene Kim. 2015. This One Chart Shows The Vicious Price War Going On In Cloud Computing. Retrieved from <http://www.businessinsider.com/cloud-computing-price-war-in-one-chart-2015-1>.
- [28] David Lo, Liquan Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2015. Heracles: Improving resource efficiency at scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA'15)*. ACM, New York, NY, 450–462.
- [29] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. 2011. Bubble-Up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44)*. ACM, New York, NY, 248–259.
- [30] S. Mavridis, Y. Sfakianakis, A. Papagiannis, M. Marazakis, and A. Bilas. 2014. Jericho: Achieving scalability through optimal data placement on multicore systems. In *Proceedings of the 2014 30th Symposium on Mass Storage Systems and Technologies (MSST'14)*. 1–10.
- [31] Asit K. Mishra, Joseph L. Hellerstein, Walfredo Cirne, and Chita R. Das. 2010. Towards characterizing cloud backend workloads: Insights from google compute clusters. *ACM SIGMETRICS Perf. Eval. Rev.* 37, 4 (2010), 34–41.
- [32] Onur Mutlu and Thomas Moscibroda. 2007. Stall-time fair memory access scheduling for chip multiprocessors. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-40)*. IEEE Computer Society, 146–160.
- [33] Onur Mutlu and Thomas Moscibroda. 2008. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems. In *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA'08)*. IEEE Computer Society, 63–74.
- [34] Raghunath Nambiar, Nicholas Wakou, Forrest Carman, and Michael Majdalany. 2011. Transaction Processing Performance Council (TPC): State of the Council 2010.
- [35] Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. 2010. Q-clouds: Managing performance interference effects for QoS-aware clouds. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys'10)*. ACM, New York, NY, 237–250.
- [36] Dejan Novaković, Nedeljko Vasić, Stanko Novaković, Dejan Kostić, and Ricardo Bianchini. 2013. DeepDive: Transparently identifying and managing performance interference in virtualized environments. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference (USENIX ATC'13)*. USENIX Association, 219–230.
- [37] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. 2013. Sparrow: Distributed, low latency scheduling. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*. ACM, 69–84.
- [38] Benedict Paten, Mark Diekhans, Brian J. Druker, Stephen Friend, Justin Guinney, Nadine Gassner, Mitchell Guttman, W. James Kent, Patrick Mantey, Adam A. Margolin, M. Massie, A. M. Novak, F. Nothaft, L. Pachter, D. Patterson, M. Smuga-Otto, J. M. Stuard, L. Van't Veer, B. Wold, and D. Haussler. 2015. The NIH BD2K center for big data in translational genomics. *J. Am. Med. Inform. Assoc.* 22, 6 (Nov. 2015), 1143–1147.

- [39] Paul Menage. 2006. Linux Kernel cgroups Documentation. The Linux Kernel Archives: cgroups features, including cpusets and memory controller. <http://www.kernel.org/doc/Documentation/cgroups/>.
- [40] Jeff Rasley, Konstantinos Karanasos, Srikanth Kandula, Rodrigo Fonseca, Milan Vojnovic, and Sriram Rao. 2016. Efficient queue management for cluster scheduling. In *Proceedings of the 11th European Conference on Computer Systems*. ACM, 36.
- [41] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. 2012. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the 3rd ACM Symposium on Cloud Computing (SoCC'12)*. ACM, New York, NY, Article 7, 13 pages. DOI : <http://dx.doi.org/10.1145/2391229.2391236>
- [42] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. 2013. Omega: Flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys'13)*. ACM, New York, NY, 351–364.
- [43] Yannis Sfakianakis, Stelios Mavridis, Anastasios Papagiannis, Spyridon Papageorgiou, Markos Fountoulakis, Manolis Marazakis, and Angelos Bilas. 2014. Vanguard: Increasing server efficiency via workload isolation in the storage I/O path. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC'14)*. ACM, New York, NY, Article 19, 13 pages.
- [44] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. 2011. CloudScale: Elastic resource scaling for multi-tenant cloud systems. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SOCC'11)*. ACM, New York, NY, Article 5, 14 pages.
- [45] A. Torralba, R. Fergus, and W. T. Freeman. 2008. 80 million tiny images: A large data set for nonparametric object and scene recognition. *IEEE Trans. Pattern Anal. Mach. Intell.* 30, 11 (Nov. 2008), 1958–1970.
- [46] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. 2013. Apache hadoop YARN: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing (SOCC'13)*. ACM, New York, NY, Article 5, 16 pages.
- [47] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. 2016. Ernest: Efficient performance prediction for large-scale advanced analytics. In *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI'16)*. USENIX Association, Santa Clara, CA, 363–378.
- [48] Abhishek Verma, Ludmila Cherkasova, and Roy H. Campbell. 2011. ARIA: Automatic resource inference and allocation for mapreduce environments. In *Proceedings of the 8th ACM International Conference on Autonomic Computing (ICAC'11)*. ACM, New York, NY, 235–244. DOI : <http://dx.doi.org/10.1145/1998582.1998637>
- [49] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. 2013. 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)*. ACM, New York, NY, 607–618.
- [50] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. 2013. 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)*. ACM, New York, NY, 607–618.
- [51] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. 2013. MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *Proceedings of the 2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS'13)*. 55–64.
- [52] Haoyu Zhang, Ganesh Ananthanarayanan, Peter Bodik, Matthai Philipose, Paramvir Bahl, and Michael J. Freedman. 2017. Live video analytics at scale with approximation and delay-tolerance. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI'17)*. USENIX Association, 377–392. <https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/zhang>.
- [53] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. 2013. CPI2: CPU performance isolation for shared compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys'13)*. ACM, New York, NY, 379–391.
- [54] Yunqi Zhang, Michael A. Laurenzano, Jason Mars, and Lingjia Tang. 2014. Smite: Precise qos prediction on real-system smt processors to improve utilization in warehouse scale computers. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 406–418.
- [55] Z. Zhang, K. Barbary, F. A. Nothaft, E. Sparks, O. Zahn, M. J. Franklin, D. A. Patterson, and S. Perlmuter. 2015. Scientific computing meets big data technology: An astronomy use case. In *Proceedings of the 2015 IEEE International Conference on Big Data (Big Data'15)*. 918–927.

Received September 2017; revised March 2018; accepted April 2018