UNIVERSITY OF CRETE
DEPARTMENT OF COMPUTER SCIENCE
FACULTY OF SCIENCES AND ENGINEERING

QoS-based Resource Management for Increasing Server Utilization in the Cloud

by

Yannis Sfakianakis

B.Sc., Computer Science, University of Crete, Greece, 2005

M.Sc., Computer Science, University of Crete, Greece, 2008

PhD Dissertation

Presented

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

Heraklion, June 2023

UNIVERSITY OF CRETE

DEPARTMENT OF COMPUTER SCIENCE

QoS-based Resource Management for Increasing Server Utilization in the Cloud

PhD Dissertation Presented

by **Yannis Sfakianakis**

in Partial Fulfillment of the Requirements

for the Degree of Doctor of Philosophy

APPROVED BY:

Author: Yannis Sfakianakis

Supervisor: Angelos Bilas, Professor, University of Crete

Committee Member: Manolis Katevenis, Professor, University of Crete

---

Committee Member: Polyvios Pratikakis, Associate Professor, University of Crete

---

Committee Member: Kostas Magoutis, Associate Professor, University of Crete

---

Committee Member: Stergios Anastasiadis, Professor, University of Ioannina

---

Committee Member: Dimitrios Tsoumakos, Associate Professor, National Technical University of Athens

---

Committee Member: Sotirios Xydis, Assistant Professor, National Technical University of Athens

---

Department Chairman: Antonis Argyros, Professor, University of Crete

Heraklion, June 2023

Dedicated to my mother Effrosyni,
my wife Thania
my children Fenia and Mara,
and
my brother Manos.

Acknowledgments

During my Ph.D., I was fortunate to collaborate with highly skilled people that helped me grow as a researcher and a specialist. I am grateful to my supervisor Prof. Angelos Bilas for his constant guidance. He challenged most of my ideas which helped develop my technical skills as well as an engineer. He was always available to discuss my ideas and concerns, providing directions and invaluable advice on my research. His mentorship and his interdisciplinary approach were a source of inspiration during my Ph.D. studies.

I want to especially thank Manolis Marazakis, who was always willing to discuss and support any new ideas for my research, and for all the creative brainstorming time we spent together. Moreover, he was a key individual in times of frustration and always supported me psychologically.

I am grateful to my thesis committee members Manolis Katevenis, Polyvios Pratikakis, Kostas Magoutis, Stergios Athanasiadis, Dimitrios Tzoumakos, and Sotirios Ksydis for their feedback during my defense and for their comments that helped me prepare the final version of this thesis.

I want to express my deepest gratitude to the great people in this lab, Tasos Papagiannis, Manos Pavlidakis, Stelios Mavridis, Iacovos G. Kolokasis, Nikos Papakonstantinou, Foivos Zakkak, Antonis Chazapis, Eleni Kanellou, Christi Symeonidou, Christos Kozanitis, and all other past and present members of CARV for preserving the balance between work and real life, making the lab a fun place to be.

I am grateful to my wife Thania, for her patience, continuous support, and encouragement during the period of my doctoral research.

Last but not least, I would like to express my deepest and most sincere gratitude to my mother Effrosyni, and my brother Manos, for their love and support throughout all these years. Without your psychological support I would have never been able to complete my

Ph.D.

I would also like to thank the Institute of Computer Science (ICS) in Foundation for Research and Technology-Hellas (FORTH), which supported me with graduate scholarships throughout my doctoral studies. Funding comes from several European and Greek-funded projects that include: IOLanes (FP7-ICT-2009-4), CoherentPaaS (FP7-ICT-611068), LeanBigData (FP7-ICT-619606), Vineyard (GA 687628), EVOLVE (GA 825061), HiPEAC (GA 871174), EuroCC (GA 951732), and Sentitour at Scale (T1EDK-02857).

Abstract

Cloud computing is compelling because it simplifies the management of the infrastructure and allows elastic scaling of resources for applications. As a result, more and more professionals and enterprises rely on the cloud to process and store their information. This increasing demand for cloud resources forces providers to constantly expand their infrastructure and maintain more servers, significantly increasing costs. At the same time, cloud users care for application performance and tend to overprovision applications, which results in a significant portion of the cloud resources being unused. Therefore, providers must handle the tradeoff between application performance and resource utilization to reduce the cost of the infrastructure and, at the same time, keep users satisfied.

In this dissertation, we propose techniques for resource management that increase utilization while, at the same time, maintaining application performance above a user-defined level. The system estimates the resources required to achieve a certain level of performance and then places the application appropriately in the infrastructure.

First, we design and implement a profile-based approach in a sandboxed environment and create performance models for each application. The system uses these profiles to accurately correlate the allocated resources to application performance and minimize the unused resources of the infrastructure. However, profile-based approaches have limitations: (1) the system cannot handle "unknown" applications and (2) the runs required to generate the profiles can be overwhelming.

Next, we address these limitations using a reactive approach, generating performance models on-the-fly during the application execution. The system gradually learns the behavior of each application using a feedback loop controller and constantly improves the estimations about the required resources. Therefore, the reactive approach can handle applications without prior knowledge of their performance profiles and adapt to workload

changes. The limitation of reactive approaches is that they cannot handle well workloads with rapid changes in their load.

Finally, we propose applying the reactive approach to serverless computing, where we adjust the required resources based on the current changes in the workload. These predictions help the system adjust quickly to sudden changes in the load and maintain the tail latency of the application.

We evaluate the proposed system using synthetic workloads that resemble real data center workloads. To achieve that, we develop a methodology that: (1) processes data center traces from major providers, (2) extracts their most important characteristics, (3) scales the workload to match the underlying infrastructure, and (4) executes the workload using well-known cloud applications that match the ones used in the trace. Our results show significant improvement in the utilization of the infrastructure without a visible drop in application performance.

Keywords: Cloud computing, resource allocation, resource assignment, task scheduling, workload generation, cloud deployment

Supervisor: Angelos Bilas, Professor
Computer Science Department
University of Crete

Περίληψη

Το υπολογιστικό νέφος (cloud computing) είναι σημαντικό επειδή απλοποιεί τη διαχείριση του υποδομής και επιτρέπει σχεδόν άπειρη κλιμάκωση των πόρων. Άποτέλεσμα, όλο και περισσότεροι επαγγελματίες και επιχειρήσεις βασίζονται στο νέφος για την επεξεργασία και αποθηκεύουν τις πληροφορίες τους. Αυτή η αυξανόμενη ζήτηση για πόρους νέφους αναγκάζει τους παρόχους (providers) να επεκτείνουν συνεχώς την υποδομή τους και να διατηρούν περισσότερους διακομιστές (servers) γεγονός που αυξάνει σημαντικά το κόστος τους. Ταυτόχρονα, οι χρήστες του νέφους νοιάζονται για την απόδοση της εφαρμογής και τείνουν να κάνουν υπερπαροχή πόρων στις εφαρμογές τους, η οποία έχει ως αποτέλεσμα ένα σημαντικό τμήμα από τους πόρους του νέφους να είναι αχρησιμοποίητο. Επομένως, οι πάροχοι πρέπει να χειριστούν την ανταλλαγή μεταξύ απόδοση της εφαρμογής και χρήση πόρων για τη μείωση του κόστους της υποδομής και ταυτόχρονα να κρατούν τους χρήστες ικανοποιημένους.

Σε αυτή τη διατριβή, προτείνουμε τη χρήση έξυπνης διαχείρισης πόρων, που αυξάνει τη χρήση των πόρων στο όριο ενώ, ταυτόχρονα, διατηρεί την απόδοση των εφαρμογών πάνω από ένα ικανοποιητικό επίπεδο σύμφωνα με το χρήστη. Το σύστημα εκτιμά τους πόρους που απαιτούνται για την επίτευξη ενός συγκεκριμένου επιπέδου απόδοσης και στη συνέχεια τοποθετεί την εφαρμογή κατάλληλα στην υποδομή.

Αρχικά, σχεδιάζουμε και εφαρμόζουμε μια προσέγγιση βασισμένη σε προφίλ απόδοσης, όπου κάνουμε πολλαπλές εκτελέσεις σε ένα απομονωμένο περιβάλλον για τη δημιουργία μοντέλων απόδοσης για το κάθεμια από τις διαθέσιμες εφαρμογές. Το σύστημα χρησιμοποιεί αυτά τα προφίλ για να συσχετίσει με ακρίβεια τους δεσμευμένους πόρους με την απόδοση των εφαρμογών και ελαχιστοποιούν τους αχρησιμοποίητους πόρους της υποδομής. Ωστόσο, οι προσεγγίσεις που βασίζονται στο προφίλ έχουν κάποιους περιορισμούς: (1) το σύστημα δεν μπορεί να χειριστεί «άγνωστες» εφαρμογές και (2) οι εκτελέσεις που απαιτούνται για τη δημιουργία των προφίλ μπορεί να είναι υπερβολικά πολλές.

Στη συνέχεια, αντιμετωπίζουμε αυτούς τους περιορισμούς χρησιμοποιώντας μια αναδραστική προσέγγιση, όπου παράγουμε μοντέλα απόδοσης επί τόπου κατά την εκτέλεση της εφαρμογής. Το σύστημα μαθαίνει σταδιακά τη συμπεριφορά κάθε εφαρμογής χρησιμοποιώντας έναν ελεγχτή βρόχου ανάδρασης και βελτιώνει συνεχώς τις εκτιμήσεις σχετικά με τους απαιτούμενους πόρους. Επομένως, η αναδραστική προσέγγιση μπορεί να χειριστεί εφαρμογές χωρίς προηγούμενη γνώση των προφίλ απόδοσης τους και μπορεί να προσαρμοστεί στις αλλαγές του φόρτου εργασίας. Ο περιορισμός των αναδραστικών προσεγγίσεων είναι ότι δεν μπορούν να χειριστούν καλά φόρτους εργασίας με γρήγορες αλλαγές στα αιτήματα των χρηστών.

Τέλος, εφαρμόζουμε αυτές τις προβλέψεις σε περιβάλλοντα όπου υπαρχουν εκρήξεις εισερχόμενων αιτημάτων. Αυτές οι προβλέψεις βοηθούν το σύστημα να προσαρμοστεί γρήγορα στις ξαφνικές αλλαγές στο φόρτο εργασίας και να διατηρήσουν την καθυστέριση του χρόνου απόκρισης της εφαρμογής σε χαμηλά επίπεδα.

Αξιολογούμε το προτεινόμενο σύστημα χρησιμοποιώντας συνθετικούς φόρτους εργασίας εφαρμογών (workloads) που προσομοιώνουν πραγματικούς φόρτους εργασίας εφαρμογών, όπως αυτούς που αντιμετοπίζουν οι πάροχοι νέφους. Για να το πετύχουμε αυτό, αναπτύσσουμε μια μεθοδολογία που: (1) επεξεργάζεται ίχνη (traces) από κέντρα δεδομένων (data centers) από μεγάλους παρόχους, (2) εξάγει τα πιο σημαντικά τους χαρακτηριστικά, (3) κλιμακώνει το φόρτο εργασίας ώστε να ταιριάζει με την υποκείμενη υποδομή και (4) εκτελεί το φόρτο εργασίας χρησιμοποιώντας γνωστές εφαρμογές νέφους που ταιριάζουν με αυτές που χρησιμοποιούνται στο ίχνος. Στα αποτελέσματά μας, παρουσιάζουμε σημαντική βελτίωση στο αξιοποίηση της υποδομής χωρίς ορατή πτώση στην εφαρμογή εκτέλεση.

Λέξεις κλειδία: Υπολογισμός νέφους, κατανομή πόρων, ανάθεση πόρων, προγραμματισμός εργασιών, δημιουργία φόρτου εργασίας, ανάπτυξη νέφους

Επόπτης: Άγγελος Μπίλας, Καθηγητής

Τμήμα Επιστήμης Υπολογιστών

Πανεπιστήμιο Κρήτης

Bibliographic Notes

The publications related to this dissertation are:

(i) 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 **(ACM SoCC '14)**.

(ii) Yannis Sfakianakis, Christos Kozanitis, Christos Kozyrakis, and Angels Bilas. 2018. *Quman: Profile-based improvement of cluster utilization.* ACM Transactions on Architecture and Code Optimization **(TACO '18)**.

(iii) Yannis Sfakianakis, Manolis Marazakis, and Angelos Bilas. 2020. *DyRAC: Cost-aware Resource Assignment and Provider Selection for Dynamic Cloud Workloads.* 2020 IEEE 26th International Conference on Parallel and Distributed Systems **(ICPADS '20)**.

(iv) Yannis Sfakianakis, Eleni Kanellou, Manolis Marazakis, and Angelos Bilas. 2021. *Trace-Based Workload Generation and Execution.* In European Conference on Parallel Processing, pp. 37-54. Springer, Cham, 2021 **(Euro-Par '21)**.

(v) Yannis Sfakianakis, Manolis Marazakis, and Angelos Bilas. 2021. *Skynet: Performance-driven Resource Management for Dynamic Workloads.* 2021 IEEE 14th International Conference on Cloud Computing **(IEEE CLOUD '21)**.

(vi) Yannis Sfakianakis, Manolis Marazakis, Christos Kozanitis, and Angelos Bilas. 2022. *LatEst: Vertical elasticity for millisecond serverless execution.* In 2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing **(CCGrid '22)**, pp. 879-885. IEEE, 2022.

More specifically, Chapter 2 is based on *(i)* & *(ii)*. Chapter 3 based on *(iv)* & *(v)*, Chapter 4 based on *(vi)*, and finally Chapter 5 based on *(iii)*.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Cloud computing has transformed the way we manage hardware resources. First, IT companies do not need to plan or commit to extensive infrastructure. They can scale their applications according to the workload without making elaborate plans for resource provisioning. Second, cloud computing provides the ability of short-term payments, which allows for minimizing the cost when scaling compute resources. Third, cloud users can devote more time developing their applications than managing the infrastructure.

For these reasons, the industry is shifting more and more towards cloud computing, which increases the data being processed in the cloud [97, 141] and the demand for cloud resources [56]. Hence, cloud providers are constantly increasing the available cloud resources to cope with this growing demand. However, there are two technological limitations that data centers face which prevents the constant increase of their infrastructure:

- Power is limited. Data centers require a large amount of power to operate, and the electrical grid may not be able to supply the required power in all data center locations [33, 117].

- Data centers consume a large amount of energy and the cost of this energy can be a significant expense for data center operators [25, 113, 87].

One key factor for improving data center processing capacity is server utilization [123]. By increasing server utilization the available servers and, in general, data centers can accommodate more applications per unit of time, which reduces the energy consumed and

the power required per unit of work performed. Increasing utilization has plenty of room for improvement, as the resource utilization in data centers is relatively low. For instance, the typical CPU utilization for data centers is approximately 12% [131, 28, 65].

In this dissertation, we focus on resource management and more specifically we contribute to the following categories of resource management:

Offline resource estimation: This approach [44, 59, 72, 43, 85, 140, 93] statistically predicts the performance of applications by running applications in a sandboxed environment [44, 43, 93], or analyzes performance metrics from historic data [85, 59, 140, 72]. Compared to the related work, we employ resource isolation to simplify performance profiling. Therefore, we create accurate performance profiles by running applications separately using only a few samples. Then, we can use these profiles to estimate the resources of the concurrently running applications of a workload.

We design and implement a framework to isolate the access to resources for applications, which is critical to simplify the training of performance profiles of our offline profiling approach [111]. The framework dedicates hardware resources to competing workloads to minimize their interference. It involves a stack of device drivers that create a full I/O path in the Linux kernel. We focus on two key resources: in-memory buffers for the filesystem, and space on SSD devices that serve as a transparent cache for block devices. This leads to increased server efficiency allowing the execution of the same workloads with fewer resources, or more workloads with the same resources. In our evaluation, we present results using mixes of transactional, streaming and analytical workloads.

Afterwards, we provide a profiler on top of the isolation framework that generates performance profiles for applications [107]. It performs controlled runs of applications on a dedicated server to determine the resource requirements of newly admitted applications. It 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. Our profiler is deployed on the side of regular infrastructure and with minimal overheads.

Reactive resource adaptation: In this approach [44, 143, 70, 104], the system adapts the resources assigned to applications after it detects a change in the incoming load or the running applications per se. Compared to the related work, we make the following improvements: (1) we define performance-level objectives for applications and change resources according to the workload to achieve them, (2) we manage arbitrary applications by supporting any performance objectives and multiple types of resources, and (3) we minimize resource fragmentation by increasing application consolidation in the servers.

First, we propose a workload generator [106] that generates realistic cloud workloads, which we use to evaluate reactive resource adaptation. We develop a robust and practical methodology to generate and execute real-world-like cloud workloads driven by analyzing public data center traces. Towards this goal, we first determine which trace parameters are the most characteristic to reflect them in generated actual executions. Parameter selection is not straightforward because each trace contains hundreds of parameters. Second, we develop models of these parameters for the generation of the data center workload. Finally, we define a trace-specific application pool for the execution of the generated workloads. Workload execution is a real challenge because traces do not contain all the necessary variables to execute a workload. For example, we do not know the specific application types that were executed during the trace creation.

Next, we design and implement a system for automated cloud resource management that increases utilization without degrading application performance significantly for generic and dynamic workloads [109]. The system estimates on-the-fly the number of resources required to achieve a specific level of performance for applications, using a Proportional Derivative Integral controller [103] per application. The PID is a simple but effective technique that adapts the value of a resource proportionally to the difference between observed and targeted performances. Essentially, it builds a mapping of the targeted performance to the required resources on the fly. Building this mapping online allows it to adapt to varying input loads and different application phases for each application. Adaptation is achieved by tuning the corresponding PID instance parameters to match changing operating conditions. It employs a modified version of the PID that performs online auto-tuning of its parameters and resource estimations concurrently [45].

Application to serverless resource allocation: In serverless computing, the next phase of cloud computing [105], the main focus is on how to minimize the start time of serverless functions [27, 118]. We apply techniques of the reactive approach to serverless frameworks to achieve millisecond latency for serverless functions. Our approach mitigates the limitations of the current state-of-the-art serverless frameworks concerning resource provisioning and assignment of serverless functions and mechanisms by applying our reactive resource adaptation approach.

In our work, we design and implement a controller for serverless frameworks that scale the resources of applications within hundreds of microseconds [110]. Our goal is to minimize the tail latency of serverless functions during workload bursts. We apply the reactive approach to serverless computing based on vertical elasticity that achieves millisecond-level tail latency. To achieve that, we add two components to the autoscaling mechanisms of the state-of-the-art serverless frameworks: (1) vertical elasticity that can scale resources within hundreds of microseconds and (2) resource prediction to handle the incoming load efficiently. That way, we minimize the times required to horizontally scale the resources, which significantly improves the serverless function tail latency.

Cost-aware cloud deployment: Finally, we utilize our previous findings in the deployment of cloud services to optimize costs associated with allocating cloud resources and selecting cloud providers [39, 64]. Related work approaches this issue by monitoring usage and costs, implementing strategies to reduce costs, and applying them when deploying applications in the cloud.

In our work, we explore how to minimize the cost of deployment of cloud services on real cloud providers [108]. We consider cost minimization of cloud services as the combined problem of selecting resource assignment (RA), VM type, and provider. Our goal is to choose the most suitable number, size, and type of VM instances that host an application. We focus on distributed interactive services that execute for a long time and respond to incoming request streams, e.g. a web server or a database server. Workloads are client applications that generate bursty requests to cloud services. Dimensioning VMs based on the specific resources required by a service, e.g. the number of VM cores, is often overlooked in related

work and practice, as a means to simplify resource provisioning. Assigning resources to VMs and servers based on actual service demands can significantly impact the cost.

This thesis addresses the challenge of improving the efficiency of using cloud resources. We propose performance profiling based on isolation for static workloads and a reactive approach based on a feedback loop controller for dynamic workloads. Additionally, we apply our proposed techniques in serverless frameworks and for cost minimization of cloud deployments.

1.1 Contributions

The specific contributions of this dissertation are:

1. We design and implement a resource partitioning mechanism in the Linux kernel that simplifies the modeling of estimating resources for cloud workloads. We design and implement a practical and accurate approach for resource estimation based on resource isolation and application performance profiling.

2. We define a methodology for the generation and execution of representative cloud workloads based on data center traces of major cloud providers. We design and implement an efficient feedback loop controller that estimates application resources on-the-fly, without prior knowledge and recurrent patterns, that address the limitations of static profiling techniques.

3. We design and implement a novel adaptive controller for serverless frameworks that manages the resources of serverless functions and achieves millisecond latency.

4. Finally, we propose a cost model based on the VM offerings of different cloud providers for deploying cloud services.

1.2 Organization

The rest of this dissertation is organized as follows. Chapter 2 presents the first part of the isolation framework that concerns the I/O path. Then, it presents the offline profiler

along with an extended version of the isolation framework that manages all resources of a server. Chapter 3 presents the workload generator. Next, it presents the adaptive cloud resource management system. Chapter 4 presents the controller for millisecond function execution of serverless frameworks. Chapter 5 presents our approach to minimizing the deployment costs for cloud services. Chapter 6 reviews related work. Finally, Chapter 7 presents directions for future work and concludes this thesis.

Chapter 2

Offline Resource Estimation

2.1 *QuMan*: Profile-based Resource Estimation

This section describes *QuMan* and its main components: (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 2.2.5), (b) a profiler, which identifies resource requirements for each application (Section 2.3), and (c) an admission controller, which admits applications to a server based on specific policies (Section 2.3.1).

2.1.1 Overview

Figure 2.1 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 2.1.2). 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

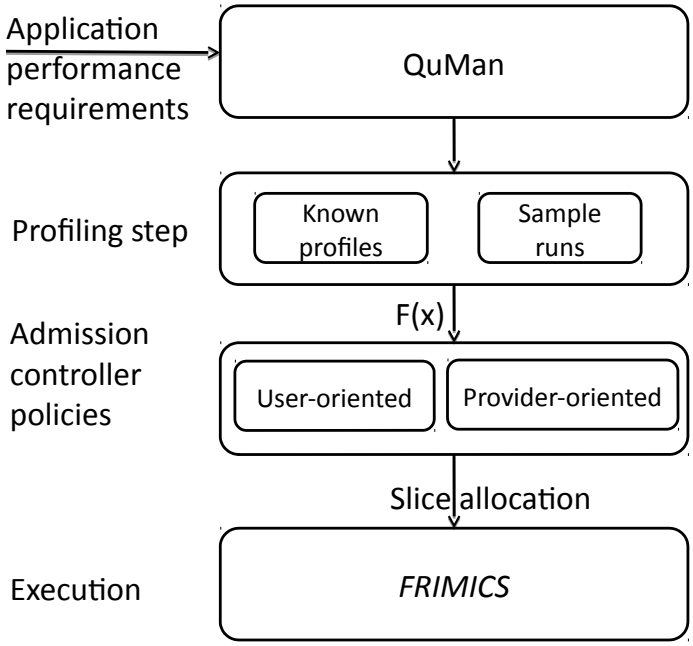


Figure 2.1: *QuMan*'s flowchart for assigning resources to application and performing admission control.

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 [90, 16, 37]. 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 ([44]) and alleviates this pressure by profiling only a small part from the beginning (first 100 seconds) 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, they can follow the user-oriented policy (Section 2.3.1). If, on the other hand, 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.

2.1.2 *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 standalone 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 2.2: linear, convex, or concave. For example, applications that are highly concurrent with little synchronization, such as Machine Learning training models which are known to scale linearly or sub-linearly [124, 29] may follow the linear curve. Applications that exhibit some sort of working set behavior concerning 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.

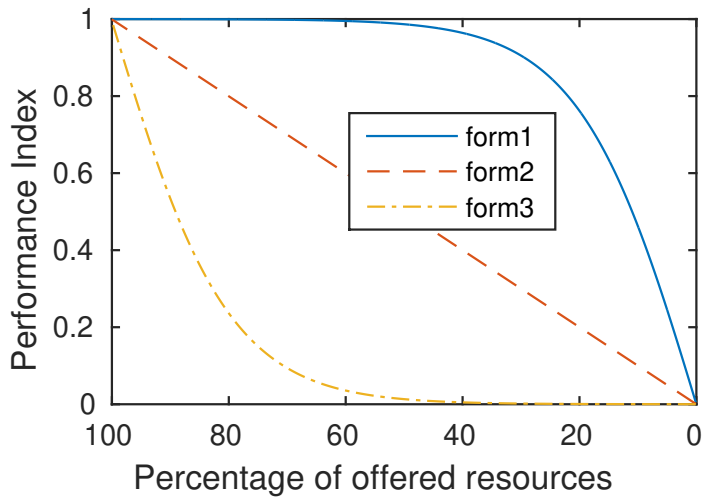


Figure 2.2: The three main patterns we assume for application *Performance Index*.

Table 2.1: *Vanguard* modules roles and responsibilities.

Module	Layer	Sliced Resource
pFS	VFS	Namespace management and CPU affinity control
pJournal	Journaling	Log Devices and Transactions
pCache	DRAM Cache	I/O Cache buffers and NUMA placement
pFlash	SSD-Cache	SSD space, device access path

2.2 Isolation Framework Design and Implementation

Our intuition for eliminating interference among workloads is enforcing separate I/O paths and guaranteeing undisturbed access to the underlying hardware. Essentially, we need to isolate each I/O related resource of the server, so that each VM utilizes a separate "*slice*" of the server, emulating that way a "standalone" execution. Our approach operates at the host, without requiring changes to VMs and application code. Our main contribution is the design and implementation of an I/O stack for the Linux kernel that significantly minimizes the performance degradation due to interference, as encountered in VM Hosts and other cloud and utility computing scenarios.

The I/O path consists of hardware resources (CPU cores, memory buffers, storage devices) and the associated systems software servicing the I/O requests initiated by applications. We apply the concept of slicing at various levels of the I/O stack. Table 2.1 shows the layer and resource that each module of *Vanguard* "*slices*". Every level/module is dedicated to the control of specific resources, We slice different resources in each layer, allowing us to distribute them independently and optimally for every scenario.

I/O Stack

Figure 2.3 juxtaposes the existing I/O stack in Linux (left) with our *Vanguard* design (right). In Linux, the following software layers are present:

- the Linux kernel VFS, which serves as the entry point for filesystems via software hooks;

- a POSIX-compliant filesystem that uses a transaction log for meta-data and in some cases for data protection;
- the unified page and buffer cache, that caches filesystem blocks in DRAM;
- any block-level filter drivers for I/O virtualization such as software RAID and SSD Caching;
- the block-level I/O scheduler for storage device performance optimization;
- the SATA or SCSI controller drivers for interfacing with storage devices.

A crucial observation is that all application workloads co-located on a host issue their I/O requests to a shared filesystem, which in turn uses the global page cache for holding recently used blocks of data in DRAM for all of them, usually with no distinction between workloads. Further down the I/O stack, all I/O requests are served by going through shared and potentially contended code-paths for access to the storage devices that provide the backing storage for the filesystem. In our results, we demonstrate that this approach leads to unacceptable performance interference between co-located workloads, whereas *Vanguard* exhibits significantly better performance under the same conditions.

To avoid extensive modifications to the Linux kernel, we implement our design as a stack of loadable modules. The entry point for application code is a journal-based filesystem (*pJournal* and *pFS*) that follows VFS conventions, so that application code does not need modifying or rebuilding. Having a filesystem in place, we achieve control of the processing flow for I/O requests; specifically, we bypass the unified buffer/page cache, instead using our own DRAM caching layer (*pCache*) providing us with per-workload pools of filesystem buffers. A further degree of control is the capability to provide per-workload instances of the filesystem journal, thus providing each workload with a separate access path to the underlying storage devices. In this chapter, we evaluate the effectiveness of this approach for a variety of workloads running on a virtualization host that has two types of devices, solid-state devices (SSDs) and hard disks, arranged so that the SSDs serve as a transparent cache (*pFlash*) for the underlying hard disks([78]). With *Vanguard* we achieve three prerequisites for performance isolation: (1) enforce per-workload limits

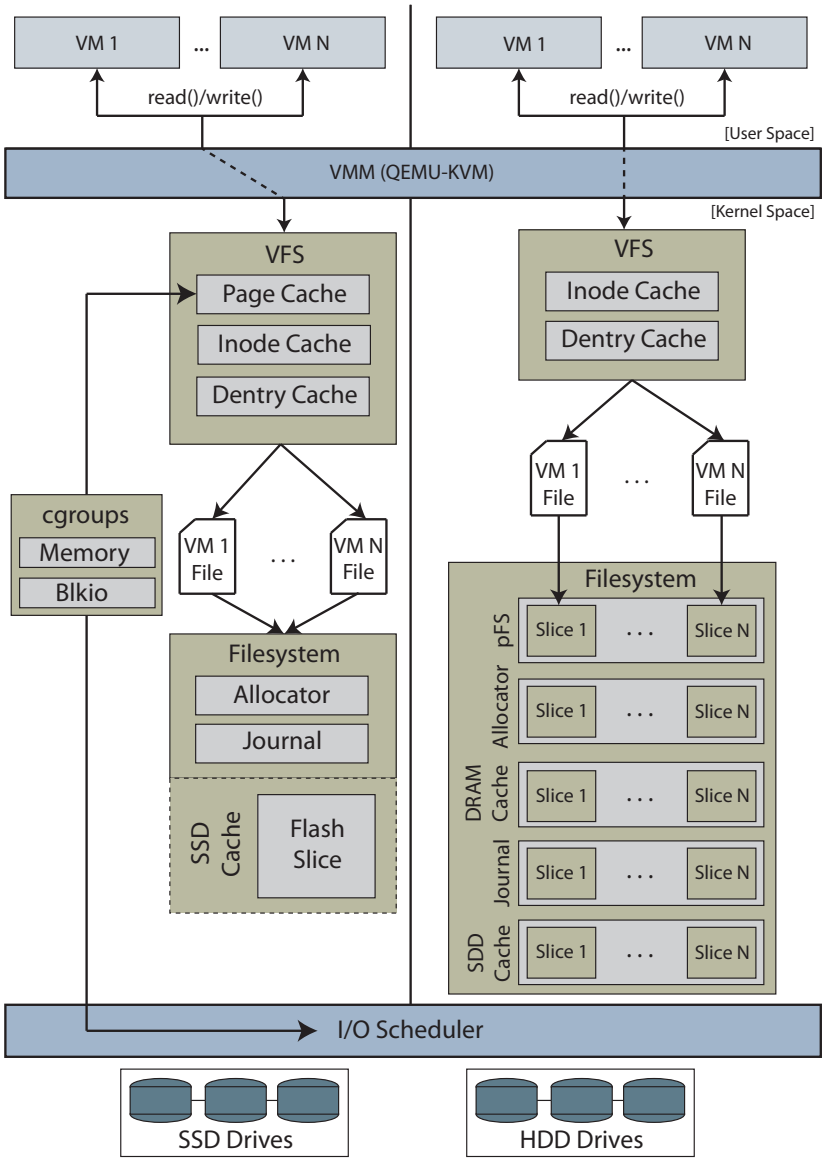


Figure 2.3: Overall design of the native (left) and *Vanguard* (right) I/O stack.

on the amount of I/O memory and SSD cache, (2) reduce contention across workloads due to synchronization and global policies in the hypervisor, and (3) allow I/O memory and threads to be placed with improved thread/data affinity, on servers that exhibit more pronounced NUMA and contention effects. We do not explicitly regulate traffic in the memory and system interconnects; however, our evaluation shows that regulating resource allocations across workloads is sufficient to reduce performance interference.

Table 2.2 provides an overview of the internal structures of *Vanguard*, listing the thread-safe data structures and execution contexts involved in each layer. The following information is listed for each data structure: function, data structure type (e.g. array, list, hash table), number of instances (e.g. per-*slice* vs global), and persistence. For each execution context, we identify the type of context (e.g. thread context vs bottom-half context), as well as the number of instances and placement (e.g. per *slice*, per CPU core). As shown in this table almost all data structures used in our design have a separate instance per *slice* to eliminate contention. The only data structures that differ are: (a) the memory pool in *pCache* has per CPU per *slice* instances to ensure locality and (b) the VFS layer that is not modified so that applications still see a compatible filesystem.

Table 2.2: Data structures and execution contexts for the layers of the *Vanguard* partitioned I/O stack.

Layer	**Data Structures (protected/thread-safe)**				**Execution Contexts**	
	function	structure	instances	stored	function	instances
pFS	slice map	array	1	MEM/DSK	application thread	inline
	container	block allocator	depends on disk space	DSK		
	container list	linked list	per slice	MEM/DSK		
	VFS inode cache	Linux structure	1	MEM		
	block map	array	per inode	MEM/DSK		
pJournal	transaction FIFO	circular array	per slice	DSK	application thread	inline
	transaction buffer	linked list	per slice	MEM	log write-back thread	per slice
	block map	hash table	per slice	MEM	device completion	s/w interrupt
pCache	block map	hash table	per slice	MEM	application thread	inline
					pipeline thread	per slice
	memory pool	stack	per slice	MEM	evictor thread	per slice
			per NUMA		device completion, top-half	s/w interrupt
			per CPU		device completion bottom-half workq	per slice
pFlash	block map	hash table	per slice	MEM	request thread	per slice/core
	element dependencies	array	per slice	MEM	device completion	s/w interrupt

Next, we discuss each layer in detail.

2.2.1 Journaling Filesystem

We design a new VFS compliant filesystem *pFS*. Existing filesystems employ techniques such as extents and allocation groups to combat scalability issues. We apply the '*slicing*' concept to completely decouple different workloads running on a single filesystem instance,

i.e. multiple independent *pFS slices* co-exist below a single mountpoint. All *slices* access dedicated block ranges in the underlying storage and in-memory data structures, reducing contention and interference across *slices*. *Slice* parameters are user defined and are chosen on a per workload basis.

A step further to *slicing* resources is their assignment to user processes. In *pFS* we implement a mechanism to assign a *slice* to a process for all I/O issuing tasks, depending on their related memory buffers. By implementing task placement in the filesystem, we reduce performance interference and improve system efficiency in multiple areas:

- Minimize remote accesses, therefore eliminating interference caused by NUMA effects and memory controller contention;
- Minimize task migrations, thereby mitigating CPU scheduler load and reduce interference in the CPU caches.

A similar approach is presented in [142], where the I/O threads are placed to the processors closer to the underlying SSD, resulting in twice the I/O performance. In [88] using *Vanguard*, we show the improvement in memory throughput when applying task placement, where we minimize NUMA effects in the system.

Much research has been done on finding fair/optimal resource allocation in servers running multiple workloads. In this chapter, we focus on eliminating interference; for this reason, we have chosen to implement a simple policy for assigning slices to workloads. We implement a round-robin policy where every directory within the root of *pFS* is assigned a 'target' *slice*. We copy the data for each workload to a top-level directory to ensure their undisturbed access to the I/O path.

pFS manages storage space by allocating *containers* for each *slice* (similar in function to the *allocation groups* in the XFS filesystem) and extents within containers.

Our *slicing* approach reduces interference but complicates the fault tolerance of the filesystem. We implement *pJournal* not only to implement fault tolerance, but also to *slice* the journal's operation. *pJournal* implements atomic operation sequences(transactions) with all-or-nothing semantics. *slicing* in the journaling layer corresponds to providing different block ranges (and potentially devices) for the transaction log of each *pJournal*

slice. Transactions are accelerated by an in-memory transaction buffer that mirrors the active part of the transaction log on disk.

Read operations are served directly from the underlying device *slice*, unless the data blocks are found in the transaction buffer of *pJournal*. Lookup operations on the transaction buffer are served by a hash-table in-memory structure. The handling of metadata writes is more complex, as we need to preserve atomicity. First, we copy the buffers from the I/O request into the in-memory transaction buffers. Subsequently, we mark the data from transactions as ended and write them to the transaction log.

Several transactions are grouped together, and served as one batch. The batching results in large writes to the transaction log. Having the data in the persistent transaction log allows for replay of completed transactions to recover from failures. A separate thread (one for each *slice* of *pJournal*) performs writes to the transaction log. This thread finally issues the original write requests to the device slice.

Transaction identifiers are unique across all instances of *pJournal*. A shared atomic counter across journal instances, provides the next transaction identifier. A co-ordination protocol ensures filesystem consistency for transactions that span across instances, such as moving a top-level directory.

2.2.2 DRAM I/O Cache

pCache [88] provides a partitioned DRAM I/O cache to reduce memory contention due to global policies and to allow NUMA placement. In typical servers today all workloads use the shared Linux page cache. Concurrent workloads compete for memory buffers from the page cache, potentially stealing buffers or polluting the cache in a way that destroys aggregate performance, due to the common replacement policy. Also, the page cache uses a single look-up structure, which can lead to thread synchronization overheads. In *Vanguard* we provide one DRAM cache instance per *slice* mitigating the effects of both the replacement policies, look-up structures, and NUMA effects.

pCache supports multiple independent caches over a shared pool of buffers, with a hash table per cache instance. Each hash table consists of double-linked bucket lists

and each entry contains a packed array of elements. An element describes a page-sized buffer and contains an atomic reference counter, flags and a timestamp. The cache is fully associative and implements a timestamp-based LRU approximation for replacement. The cache supports both write-back and write-through policies.

Each cache instance is supported mainly by two threads:

- The *evictor thread* that implements the replacement policy, flushes dirty elements and keeps the cache within user-specified size limits.
- The *pipeline thread*, responsible for processing cache requests. In the case of cache misses, the pipeline thread handles the I/O request processing.

For cache hits on the other hand, which are expected to be as fast as a memory copy, context switch overheads are not tolerable. For this purpose, the application's I/O issuing user-space thread context is used (*Inlining*), to avoid the penalty of two context switches (back-and-forth) between the application context and the pipeline thread.

pCache isolates locks in two dimensions: CPU cores and *pCache slices*. *pCache* data structures are duplicated across both dimensions. Each *pCache slice* has its own hash table, which results in reduced lock contention on the bucket list locks. Per-request metadata are allocated from per-CPU dedicated pools, further reducing lock contention across CPU cores. Data buffers are allocated from pools which are split across both CPU cores and *pCache slices*, reducing lock contention even further by using both dimensions for isolation.

2.2.3 SSD Cache

Completing our I/O-Stack we include *pFlash*, a block-level write-back SSD cache, derived from our own previous work [78]. *pFlash* is a transparent cache, as it exports a block device with size equal to the size of the device being cached. Our *slicing* technique splits the SSD device in several independent cache instances. Every *slice* is given a different block range and size of SSD space, selected by the user for its intended workload. For every *pFlash slice* we maintain separate LRU lists for the eviction policy. This greatly improves the eviction I/O pattern, increasing the I/O throughput observed at the underlying hard disks. With *slices*

an I/O-intensive workload cannot cause consume SSD space beyond the limit imposed by its *slice*.

2.2.4 Vanguard Summary

For each *I/O path slice*, the following instances of our modules are in place: one *allocator slice*, one *pCache slice*, and one *pJournal* instance. For the *pJournal* instance, two instances of the *pFlash* are used for the core and journal space, respectively. The I/O path for workloads executing within VMs proceeds as follows: An application I/O request from inside the VM is either served from the VM's page-cache or reaches a virtual device. In the latter case, the request goes through the virtio channel, reaches the hypervisor (kvm module in the host's kernel) and gets forwarded to qemu, which then issues a read/write system-call to the host's filesystem (where the virtual disk image is placed). For native configurations I/O requests first go through the page-cache; only misses are served from *pFlash* (either SSD or HDD). In *Vanguard*, the request will be dispatched by *pFS* to the proper *pCache slice*. Misses are served by *pFlash* for the specific *slice*.

2.2.5 *FRIMICS*

FRIMICS uses cgroups [98] for CPU and application memory isolation and a modified version of Vanguard [111] 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 1024 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 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 2x [89].

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.

2.3 Profiler

For a newly submitted application, the profiler estimates and provides to admission controller policies 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 several 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 a 4-d representation of the *Performance Index*. Each run provides two data points, each of which is a tuple $(\boldsymbol{X}, Y)$. $\boldsymbol{X} \in \mathbb{R}^3$ 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, 16 GB of memory, and 32 GB of SSD cache. If a run that uses all available resources takes 19 seconds and causes 93.7% CPU utilization, it will produce datapoints (<8, 16, 32>, 1) for *Performance Index* and (<8, 16, 32>, 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, it finishes in 39 seconds and causes 87% CPU utilization (of those cores), it will produce datapoints (<4, 16, 32>, 0.49) for *Performance Index* and (<4, 16, 32>, 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 2.4, consists of three distinct areas, each resembling one of the three patterns we assume for the shape of the *Performance Index* (Figure 2.2). 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 2.2. 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 that 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.

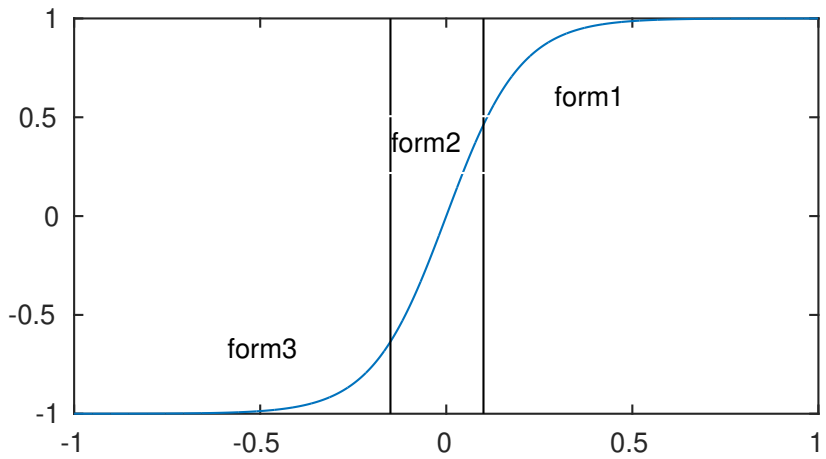


Figure 2.4: The three possible ways that an application's performance change when the hardware resources on which it executes reduce. Note, that although the figure displays a single-dimensional resource for demonstration purposes, our modeling uses a three-dimensional space, as it is not straightforward how the combination of these resources affects application performance.

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

$$f^{\vartheta}(\boldsymbol{X}) = C\frac{1 - e^{-\boldsymbol{X}\boldsymbol{\beta}}}{1 + e^{-\boldsymbol{X}\boldsymbol{\beta}}} \tag{2.1}$$

where C is scalar, $\boldsymbol{\beta}$ is the vector $\left(\beta_1, \beta_2, \beta_3\right)$, and ϑ all parameters (C and $\boldsymbol{\beta}$) 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:

$$e(\vartheta) = \sum_{m=1}^{M} (Y_m - f^{\vartheta}(\boldsymbol{X}_m))^2 \tag{2.2}$$

where $\vartheta = (C, \boldsymbol{\beta})$, f^{ϑ} is the individual characteristic function, M denotes the total number

of input points, and $f^{\vartheta}(\boldsymbol{X}_m)$ is the estimated *Performance Index* or utilization value for the same resources.

The profiler identifies the parameters ϑ that minimize total error:

$$\hat{\vartheta} = \underset{\vartheta}{\operatorname{argmin}}\, e(\vartheta) \tag{2.3}$$

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: $[C, \vartheta_1, \vartheta_2, \vartheta_3]$.

2.3.1 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, 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, the policy reduces the number of resources that are available to A and assigns them to B.

On the other hand, 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, 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:

$$QUCI(\boldsymbol{X_1}, \ldots, \boldsymbol{X}_K) = \sum_k w_{PI} f_{PI}^{\vartheta_k}(\boldsymbol{X}_k) \times w_{CPU} f_{CPU}^{\vartheta_k}(\boldsymbol{X}_k) \tag{2.4}$$

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 kth 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:

$$(\hat{\boldsymbol{X}_1}, \ldots, \hat{\boldsymbol{X}_K}) = \underset{(\boldsymbol{X_1}, \ldots, \boldsymbol{X}_K)}{\operatorname{argmax}} \; QUCI(\boldsymbol{X_1}, \ldots, \boldsymbol{X}_K), \textit{where} \sum_{k=1}^{K} (\boldsymbol{X}_k) = 1, \tag{2.5}$$

$\boldsymbol{X}_k$ 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.

2.4 *QuMan* Implementation

We have implemented *QuMan* in 10000 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 [96] to demonstrate its capabilities in a cluster environment.

2.4.1 Integration with Sparrow

Sparrow [96] is an open source resource scheduler that achieves high scheduling throughput for tasks. In order to schedule a task, Sparrow probes a number of randomly chosen worker nodes and assigns it to the node with the fastest response time.

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 2.5 shows the architecture of this integration, which uses a user-oriented policy for admission control (see Section 2.3.1). 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 2.5) 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 2.5).

If *QuMan* nodes know the performance profile of a submitted task, they will skip steps 2, 3, and 4 of Figure 2.5 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 2.5). 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 [72, 99]. 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 [100], Mishra et. al. [90] report "tasks with short duration dominate the task population", Lu et. al. [16] report that the average task duration is 192 seconds with maximum task duration of 29585 seconds. 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 100 seconds to complete [100]. After 100 seconds 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 2.2) and for that long task it calculates a line that declines with 45 degrees. 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. In order to produce consistent profiling results, *QuMan* sandboxes flush their caches before initiating each run. Thus, all profiling related datapoints include data transfer related overheads.

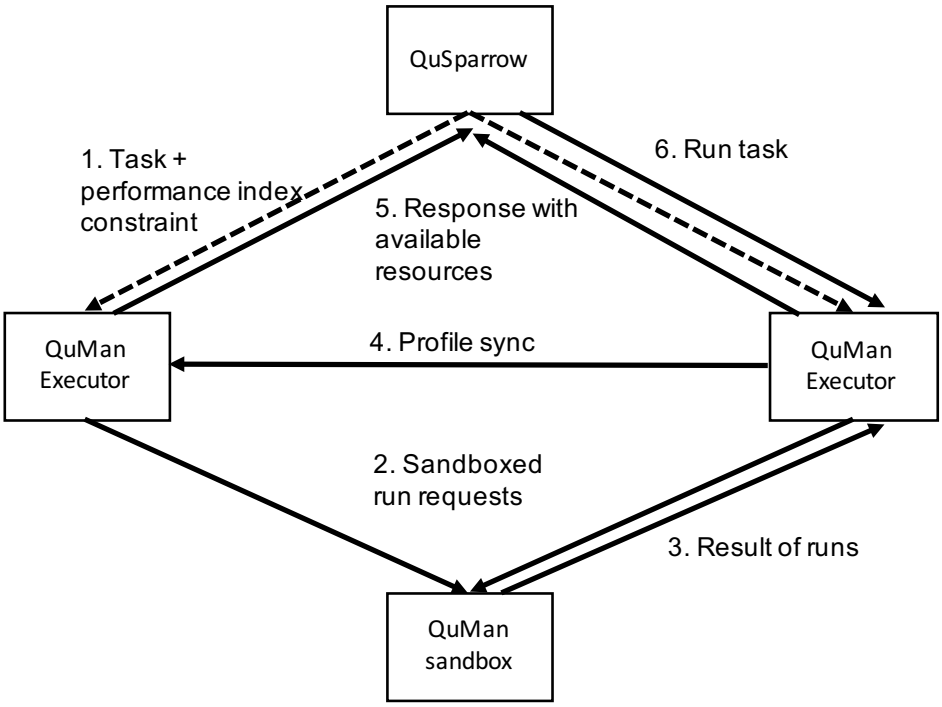


Figure 2.5: Resource negotiation steps of QuSparrow. 1) QuSparrow queries randomly a number of *QuMan* nodes whether they could run a task under a certain performance constraint. 2) If *QuMan* nodes do not know the performance profiles of the task, they will request from the sandbox to perform profile 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.

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 which can lead to decreased CPU utilization.

2.5 Vanguard Evaluation Methodology

In this section we present the experimental platform and workloads we use in our evaluation. We assume that an external entity, such as a billing system, assigns workloads to I/O path slices.

2.5.1 Experimental Platform

We perform our evaluation on a server equipped with a dual-socket Tyan S7025 motherboard, two 4-core Intel Xeon 5620 64-bit processors (with hyper-threading enabled) running at 2.4 GHz, 48GB of DDR-III DRAM and four 32GB enterprise-grade Intel X25-E SLC NAND-Flash SSDs, each individually connected to one of four LSI MegaRAID SAS 9260-8i controllers. We also utilize eight 500GB Western Digital WD50001AALS-00L3B2 SATA-II disks connected to an Areca ARC-1680-IX-12 SAS/SATA storage controller and assembled in a hardware RAID-0 configuration. The server runs Linux kernel v.2.6.32-279.5.2 (part of the CentOS distribution, v.6.3). All Native experiments use the *xfs* filesystem (with 4KB blocks). The I/O scheduler used in all experiments is the default *noop elevator*.

2.5.2 Individual and Combined Workloads

In our experiments we measure per-workload performance on a virtualization host with up to 6 VMs, running different combinations of six workloads: TPC-E, TPC-W, TPC-H, Apache, BLAST, and FIO. In most of the experiments, these six workloads are treated as having equal value, i.e. they are assigned to I/O path slices with equal resource allocations. These experiments capture plausible operating points of the server. Load can build up gradually over time, or in other cases can vary over a wide range, by having different combinations of workloads active at successive measurement intervals. In our evaluation, we present results for all possible combinations of four workloads out of the six listed above. In the rest of the experiments, we focus on excessive load, i.e. situations where the aggregate load exceeds server resource capacities. Under such conditions, we distinguish between workloads as follows: one workload is marked as *primary*, and the background load consists

of combinations of the remaining five workloads. In all experiments, there is a change in the aggregate load profile every 300 seconds. We describe the workloads of our evaluation below.

TPC-E [92] is a transactional workload that emulates the operations of a stock broker. We use 24 threads that issue transactions over a 20GB database. This workload consists of comparably localized randomly distributed small-sized I/O accesses, 15% of which are writes. The metric we focus on is the transaction rate. The CPU load in our runs is more less 50% of our system's CPUs and a single SSD is sufficient to sustain the throughput requirements. TPC-E is latency sensitive due to each transactions and therefore is I/O bounded. TPC-H [92] 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. The metric we focus on is the average execution time for twenty consecutive executions of query Q5. This query requires a lot of CPU processing before issuing the I/Os. TPC-H uses only one thread, which becomes a bottleneck if the underlying storage can achieve more or less 250MB/s fairly sequential read operations. TPC-W [92] is a transactional workload that emulates the operations of an online shop. We use the *order taking* profile with 2000 emulated clients. In this profile 50% of the client sessions include shopping-cart updates. The database size is around 3GB, and the metric we focus on is the average transaction response time. Due to its large thread count, TPC-W consumes almost all the CPUs throughout the experiment. Similar to TPC-E , disk latency affects performance especially when PostgreSQL issues flash operations for its transaction log. We run TPC-E on MySQL (v.5.0.77) and TPC-H and TPC-W on PostgreSQL (v.9.0.3).

BLAST [22] is an application from the domain of comparative genomics. We use the *blastn* program (v.2.2.27) which performs queries on nucleotide databases. Sixteen instances of *blastn* issue simultaneously queries to separate databases. *blastn* therefore, heavily utilizes both the CPU subsystems using many instances and the I/O subsystem with 16 undisturbed outstanding I/Os. We use 23GB databases in our experiments, and the metric we focus on is the average execution time over six consecutive executions.

We use Apache [9] and the *ab* benchmarking tool, to evaluate web-content serving performance. We invoke *ab* invocations in batches with a duration of 10 seconds. 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: files with size 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 [24] 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.

2.5.3 Virtual Machine Configuration

For our Linux-based virtualized environment, we had the option to choose between two hypervisor technologies: Xen and kvm. We opted for kvm, where the hypervisor is a kernel module and virtual machines are executing as regular qemu processes. The choice of kvm over Xen is not limiting the applicability of our approach; the paravirtualized I/O path for Xen is quite similar, as I/O still goes through an"I/O proxy". The I/O path goes from a virtual machine to the hypervisor, and then to an "I/O proxy" thread running within qemu. Qemu can use up to 64 I/O threads per VM (in addition to the number of virtual cores assigned to the VM). This is essential for preserving the semantics of the filesystem running inside each of the virtual machines. We assign 4 virtual cores and 3 GB of memory to each VM. Each VM configuration involves a 16GB system-disk image file and a workload data-disk image file.

2.5.4 Settings for I/O Path Slices

For the experiments with gradual load build-up, we configure four I/O path slices, each with the following settings: 6GB of DRAM for filesystem buffers, 32GB of SSD-cache space and 500GB of hard-disk space. The same settings are used for the experiments where we evaluate all combinations of four workloads. Our rationale for configuring less I/O slices

than workloads (4 slices vs 6 workloads) is that with independently submitted workloads we anticipate benefits from statistical multiplexing for resource sharing. The downside is that over-subscription of resources cannot be ruled out; thus, we evaluate *Vanguard* under conditions closer to a real-world deployment.

For the gradual build-up and four-workload combinations experiments, we assign the six workloads to the four available I/O slices by applying the following (empirically derived) rules: (a) BLAST and FIO, the two more I/O-intensive of the six workloads, are assigned to dedicated slices, (b) TPC-E and TPC-W, both OLTP workloads, share a slice, and (c) TPC-H and Apache, both workloads with primarily read I/O activity, also share a slice.

For the excess-load experiments, we configure two I/O path slices, with the following settings: 12GB of DRAM for filesystem buffers, 64GB of SSD-cache space, and 1000GB of hard-disk space. For these experiments, one workload is assigned as the *primary*, and gets to use the first of the two I/O path slices, without competition. At each 300-second interval, a different (randomly selected) combination of four workloads out of the five remaining workloads is activated in the second I/O path slice. For all experiments, the write policy of the SSD-cache is writeback.

2.6 Vanguard Results

We evaluate the effectiveness of *Vanguard* in mitigating performance interference between co-located workloads, by running three types of *timeline* experiments:

1. Excess load, where we run experiments with a *primary* workload competing with background load that changes between time intervals (as explained in Section 5.2).

2. Gradual build-up of system load, where the workload submission sequence is as follows: TPC-E, BLAST, FIO, TPC-W, TPC-H, Apache.

3. Varied load, where we run all possible 4-workload combinations out of six available workloads (total of 15).

We define two metrics, computed from our experimental measurements, to better explain our evaluation results:

The first metric represents the effectiveness of a system for a given workload mix. We have established a performance baseline B_i by running each of the workloads $i = 1, \ldots, W_T$ in a single VM on an unmodified host with the native I/O stack. We use the same baseline for all system configurations (Native, *Vanguard*) and in all the timeline experiments.

To quantify deviations from the performance baseline, we define $PO_{i,T}$ to be the observed performance score for workload i during time interval T. W_T is the number of workloads active during time interval T. The results from our timeline experiments are shown as plots over time of a *combined performance index* defined for time interval T as follows: $PI := \frac{1}{W_T} * \sum_i^{W_T} \frac{PO_{i,T}}{B_i}$. If a workload suffers from performance degradation, then the ratio $\frac{PO_{i,T}}{B_i}$ will be less than 1. The *PI* metric is essentially the average of the normalized performance scores of all running applications on a server.

A crucial clarification for the *PI* metric is that it assumes that for each workload the observed performance score and baseline are higher for better executions. This is indeed the case in our experiments for the TPC-E, TPC-W and FIO workloads, where the performance metric as per Table 2.3 is the rate of completed operations. However, for workloads where the observed performance score and baseline are *lower* for better executions, specifically BLAST, TPC-H and Apache in our experiments where the metric is latency per operation, we need to transform the observed performance score and baseline to a rate, i.e. use $latency^{-1}$ in the corresponding $PO_{i,T}$ fractions.

The second metric E_{SYS} quantifies system efficiency for a workload mix, taking into account both the combined performance index and an indicator of power consumption. Following the work presented in [52] we assume that the total (system) power consumption is a linear function of server CPU utilization: $TotalSystemPower := P_{idle} + (P_{max} - P_{idle}) * Util_{CPU}$, where P_{idle} is the (constant) power consumption of the server when idle, P_{max} is the server's maximum measured power consumption when running workloads, and $Utilization_{CPU}$ is the CPU utilization level (ranging from 0 to 1). We define the following indicator for system power consumption:

$$SYSPWR(Util_{CPU}) := \frac{P_{idle}}{P_{max}} + (1 - \frac{P_{idle}}{P_{max}}) * Util_{CPU}$$

Table 2.3: Absolute performance of *Vanguard* and Native.

Workload	Native	*Vanguard*	Metric
TPC-E	18.00	18.50	transactions/sec
TPC-H	22.45	32.20	execution time (sec)
TPC-W	504.21	547.73	transactions/sec
BLAST	35.15	45.52	execution time (sec)
APACHE	30.46	24.02	50% response time (msec)
	120.16	72.93	95% response time (msec)
FIO	1.5	1.3	GB/sec

Following [35] we set $\frac{P_{idle}}{P_{max}} = \frac{2}{3}$, i.e. we assume that P_{idle} is 67% of the maximum server power consumption. This assumption is validated from our experience with servers with substantial memory and storage resources. Finally, we define system efficiency E_{SYS} during time interval T as follows:

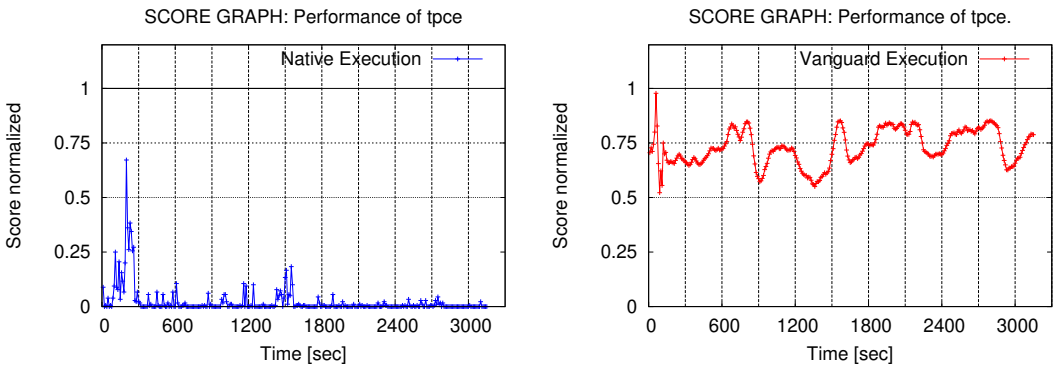
$$E_{SYS} := \frac{W_T * PI}{SYSPWR(Utilization_{CPU})}$$

The *PI* metric allows us to rank execution sequences in terms of performance: higher is better, pointing to lower degrees to performance interference between workloads. The E_{SYS} metric allows us to rank execution sequences in terms of efficiency: higher is better, pointing to more useful units of work being processed per unit of power consumption.

In the following graphs we show execution results normalized to the standalone Native configuration, i.e. we set the baseline for each workload to the performance observed with the unmodified Linux system (*Native*) when there is no performance interference. Along with performance, we also compare Native and *Vanguard* in terms of system efficiency. Table 2.3 summarizes the absolute performance results for each workload, for Native and *Vanguard*.

2.6.1 Excess Load (Primary workload vs combinations of background workloads)

In this subsection, we summarize results under conditions of excess load, when a single workload, that is designated as *primary*, runs concurrently with five *background* workloads.



(a) Performance of Native (b) Performance of *Vanguard*

Figure 2.6: Performance of TPC-E with Native and *Vanguard*, with five VMs as excess load.

In this case, we are interested to identify how affected is the *primary* workload in presence of interference. In each interval we use a different combination of the available workloads. Figures 2.6, 2.7 show the normalized performance of TPC-E, TPC-H, respectively. In the TPC-E case we observe that the performance of the Native starts to drop already in the first interval while in TPC-H the performance drops during the fifth interval. This is due to the fact that the TPC-H database is significantly less (6GB versus 45GB) and the native system manages to keep it in memory for a longer period of time. Eventually though, both workloads fail to make substantial progress. *Vanguard* has the ability to enforce private resources for workloads and therefore achieves a much better average performance. Datacenter operators that intentionally keep their servers underutilized can now add more workloads, increase their efficiency and preserve the satisfaction of their customers.

Another important result of the isolation property is the stability in performance. With *Vanguard* the primary workload maintains its performance level regardless of the background load present in the server. This feature of *Vanguard* makes it a handy tool for cloud infrastructure administrators to offer QoS guarantees.

2.6.2 Gradual Build-up of System Load

In this subsection we show how the server operates when we gradually increase the load by activating more VMs. In each time interval we add one VM. The experiment finishes after

(a) Performance of Native (b) Performance of *Vanguard*

Figure 2.7: Performance of TPC-H with Native and *Vanguard*, with five VMs as excess load.

1800 seconds, with 6 VMs running concurrently. In this experiment we focus on the overall behaviour of the server. We plot the *PI* metric every five seconds. In the timeline graphs shown below, $PI = 1$ corresponds to the maximum observed performance score. The left part of Figure 4.2 shows the combined performance index as a timeline for the gradual load build-up scenario, for Native and *Vanguard*. In the first interval, where only one VM is active, *Vanguard* outperforms the Native system; therefore its combined performance index is above 1. Overall, *Vanguard* outperforms the Native system for three of the workloads (TPC-E, TPC-W, Apache), and lags Native for the rest (by no more than 30%).

In the second interval, the I/O resources of the server are still sufficient and the two running workloads do not interfere with each other, thus the combined performance index of Native is higher than 50%. However, after the third workload in the sequence becomes active, Native cannot sustain the aggregate load, whereas with *Vanguard* the combined performance index is more or less constant at around 0.5 throughout the timeline of this experiment. Native cannot limit the performance interference between workloads, as it allocates resources from severely contented shared pools. *Vanguard* maintains distinct resource pools for each of the I/O path slices, resulting in a more predictable execution for each workload.

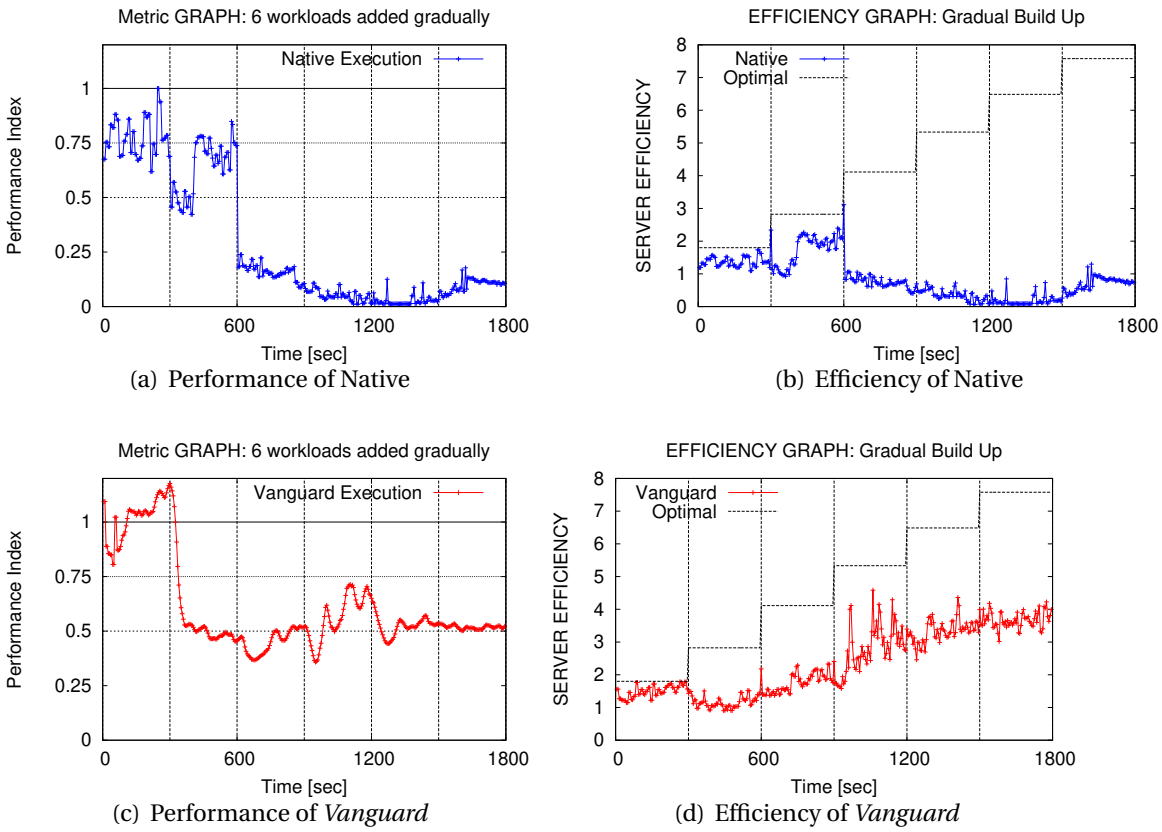
The right part of Figure 4.2 display the server efficiency for each execution (Native and *Vanguard*). We define an *optimal* execution with respect to E_{SYS} as follows: Each added workload consumes only one hyperhread (hence $Utilization_{CPU} = \frac{1}{16}$) and yields the

perfect performance score ($PI = 1$). Therefore, during each interval the optimal execution defines an upper bound for every execution. Clearly, this bound rises as we add more work to the server. We observe that the Native system's trend is descending while the trend with *Vanguard* is ascending: As we add more load to *Vanguard* it becomes more efficient, whereas Native is getting less efficient. Hence, with Native adding load translates to producing much more interference among workloads, leading to a marginal increase in aggregate performance with a disproportionately high resource consumption. On the other hand, with *Vanguard* more load leads to more efficient use of the server's resources, as *Vanguard* limits interference and manages to get better performance with the same or less CPU utilization. This graph indicates that the datacenter operators are justified to operate servers at low utilization levels, as they get less efficient with increasing load. Figure 2.9(a) summarizes per-workload performance for the gradual load build-up scenario. Native outperforms *Vanguard* only for the BLAST workload; however, *Vanguard* maintains a balanced performance envelope for all workloads, with much lower standard deviation.

Based on the data in Figure 2.9(a), a server operator can reason about plausible operating points, using the following parameters: (1) the set of baseline scores B_i, $i = 1, \ldots, W_T$ for the W_T workloads observed during the measurement period T, and (2) an *effectiveness level* E, i.e. the minimum acceptable fraction of baseline performance.

Setting $E = 0$ corresponds to the (unrealistic) expectation that all co-located workloads execute at their stand-alone performance level, without noticeable performance degradation. More realistically, setting $E = 0.25$ corresponds to the requirement that workloads suffer a performance penalty of at most 25% concerning baseline performance. A lower setting, $E = 0.5$, corresponds to a less strict constraint on performance degradation, i.e. describes an operating point that could accommodate more workloads on the same server.

For the particular experiment summarized in Figure 2.9(a), setting $E = 0.25$ means that with *Vanguard* 3 of the 6 co-located workloads operate at acceptable performance levels, as compared to 1 out of 6 with Native. Setting $E = 0.5$ means that 5 out of 6 workloads run at acceptable performance levels with *Vanguard*, as compared to 2 out of 6 with Native. Seen differently, these observations from Figure 2.9(a) imply that with *Vanguard* fewer physical servers would be needed to run all of the workloads at acceptable performance levels. Thus,



(a) Performance of Native (b) Efficiency of Native

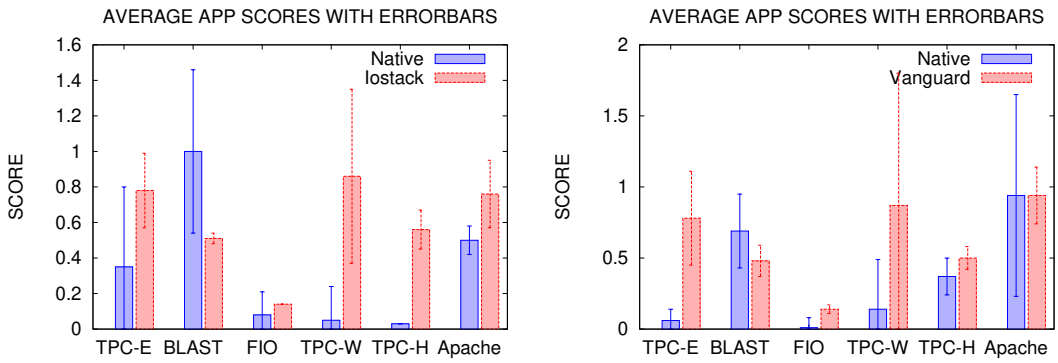
(c) Performance of *Vanguard* (d) Efficiency of *Vanguard*

Figure 2.8: Aggregate performance of Native and *Vanguard*, for the gradual load buildup scenario with up to six workloads.

Vanguard improves effectiveness from the point of view of the server operator, allowing more workloads to be co-located without an uncontrolled collapse of performance.

2.6.3 Varied Load (4-workload combinations)

In this subsection, we complete our evaluation by examining all possible combinations of 4 out of 6 workloads. We choose 4 workloads, because we know by experience that our hardware has not enough capacity to accommodate more, due to resource limitations. These experiments serve as evident proof that *Vanguard* (provided with the necessary information) can make the most out of a consolidated server, whereas the native system fails in most cases. Different combinations of co-located workloads stress resource allocation



(a) Gradual load buildup with up to 6 workload (b) Combinations of 4 out of 6 workloads

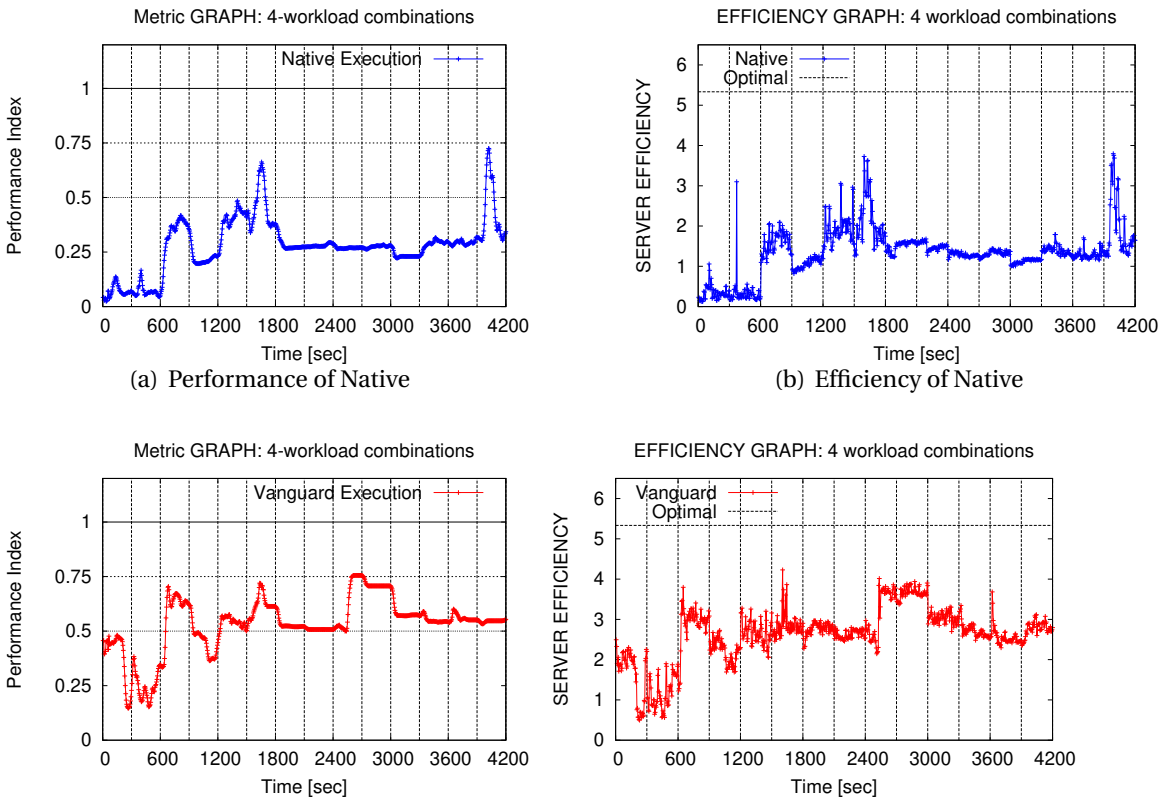
Figure 2.9: Average application score (with error bars) for Native and *Vanguard*

in varying degrees. Figures 2.10(a), 2.10(c) show the combined performance index, while figure 2.9(b) displays the average score for each workload. The per-workload scores show that, for Native 3 of the workloads (TPC-E, FIO, TPC-W) do not produce significant work, TPC-H performs on average at around 30% of its baseline and the rest (BLAST, Apache) perform much closer to their respective baselines. In sharp contrast, with *Vanguard* only one workload (FIO) suffers from severe performance degradation, whereas the other five workloads are within 50% or better of their respective baselines. FIO is being explicitly isolated from the rest of the workloads, following the workload-to-slice assignment described in Section 2.5.4. In Figures 2.10(b) and 2.10(d) we show the E_{SYS} metric for Native and *Vanguard*. The average efficiency of *Vanguard* compared to Native is 2.5x more, which is indicative of the benefits of partitioning in servers' efficiency.

Thus, with *Vanguard* the same hardware resources perform more useful work, exhibiting that way a vast improvement in energy efficiency and cost.

2.6.4 Alternatives for Achieving Isolation

In the Linux kernel the state-of-the-art mechanism for regulating resource use is currently *cgroups*. We should note that although *cgroups* offer various options, it is non-trivial to configure for effective reduction of interference across workloads. We compare *Vanguard* to *cgroups* and four alternative hand-tuned configurations aiming for performance isolation:



(a) Performance of Native (b) Efficiency of Native

(c) Performance of *Vanguard* (d) Efficiency of *Vanguard*

Figure 2.10: Aggregate performance of Native and *Vanguard*, with combinations of 4 out of 6 workloads.

1. Increase the memory budget available for I/O caching in the guest VM, and reduce the size of the hypervisor I/O cache. This ensures that I/O buffers are allocated to each workload rather than shared across workloads. The disadvantage compared to *Vanguard* is that workloads still compete in the shared I/O path in the hypervisor.

2. Create a separate filesystem for each VM, thus eliminating shared use of filesystem structures. However, these filesystems still share the single page-cache of the host. The main disadvantage compared to *Vanguard* is that each modification of the running workloads requires repartitioning the devices and creating new filesystems, which is not practically feasible.

3. Use separate filesystems (as in the previous configuration) and regulate memory use

via *cgroups*.

4. I/O caching in the guest VM, with two filesystem instances (combination of the first two alternatives).

We use an "excess" VM that generates a varying I/O request load concurrently with the performance-sensitive workload. After experimenting with various workloads as excess workloads, we opted for a synthetic workload based on *fio* [24] to allow for controlled experiments.

We define two different interference levels: low and high. We use 64 *fio* threads (as many as the I/O threads in qemu), each sequentially reading a 800-MByte file. We present results with two settings: a 'low-intensity' configuration with request size of 16KB, resulting in a relatively steady I/O throughput rate of 100MB/s, and a 'high-intensity' setting with 256KB requests, that consumes I/O throughput at a rate of 1000MB/s. For both intensity-level settings, the excess VM is allocated one core.

We provision the "primary" VM with disproportionately more memory than the excess VM inside the guest; 13GB of memory versus 3GB. We create two filesystem instances each with a dedicated SSD cache slice, with 3GB allocated for the excess load VM and 62GB for the production VM. The production VM is allocated four cores. Figure 2.11 compares *Vanguard* with these hand-tuned configurations, for the TPC-E workload. TPC-E stresses the effectiveness of both memory and device subsystem. Isolating memory by caching in the guest VM, is sufficient for low levels of excess load but the drop in performance under high excess load is very steep (7% improvement with low excess load, 19x degradation with high excess load). Isolating devices by creating one filesystem instance for each VM improves results, even for the high excess load, with degradation in the 38%-52% range. When we use separate filesystems in combination with *cgroups*, performance degradation is 28% compared to native. When we combine caching in the guest VM with two filesystem instances, we observe 10% improvement for the low setting and 23% degradation for the high setting. However, these configurations are impractical for the system administrators to reserve much more memory and a separate filesystem instance for each guest VM. *Vanguard* achieves better isolation for the high excess load setting, without this configuration.

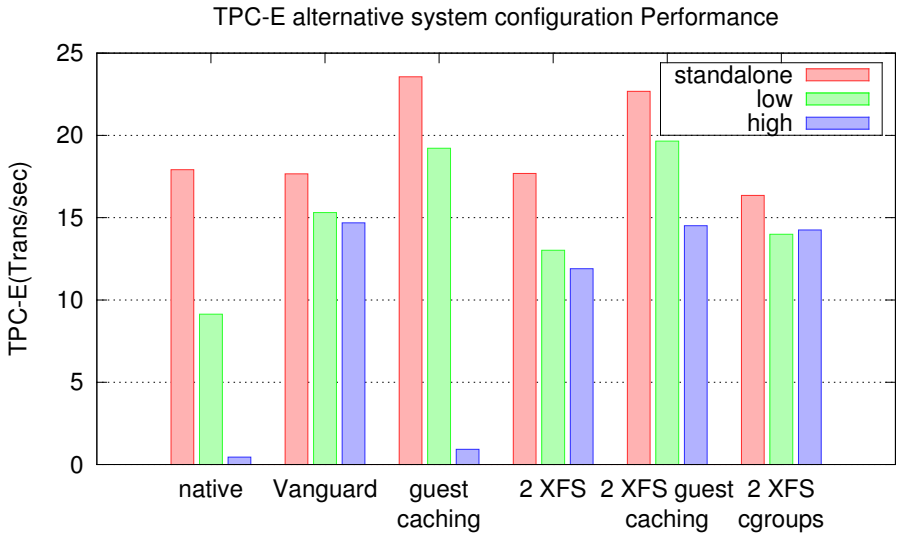


Figure 2.11: Comparison of native partitioning configurations of I/O path and *Vanguard*.

2.7 *QuMan* Evaluation Methodology

2.7.1 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 2.1 GHz and 48GB of DDR-III DRAM. For storage we use four 32 GB (128 GB 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.

2.7.2 Benchmarks

TPC-E [92] 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. 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 [92] 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 twenty consecutive executions of query Q5. We use this query, because it is both CPU and IO intensive.

We use Apache [9] and the *ab* benchmarking tool as our web-content serving application. We issue *ab* requests in batches with a duration of 10 seconds. 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 [24] 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 [22] is an application from the domain of genomics. We use the *blastn* program (v.2.2.27) which 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.

2.7.3 Workload Mix

In our single node evaluation we assume that 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 2.4.1.

2.8 *QuMan* Evaluation

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

2.8.1 *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 2.4 shows the exact hardware configuration that each application receives. Figure 2.12(a) presents the *Performance Index* over time for each application for

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

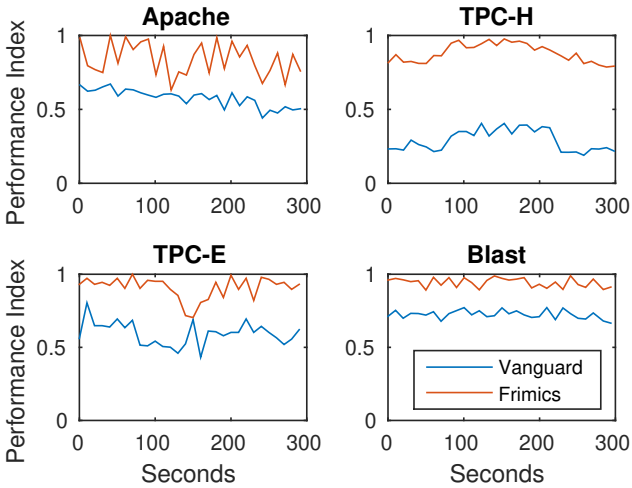
Table 2.4: Slice configurations for TPC-H, Apache, TPC-E, BLAST while they run concurrently and each receives a guaranteed *Performance Index* of 0.8

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%.

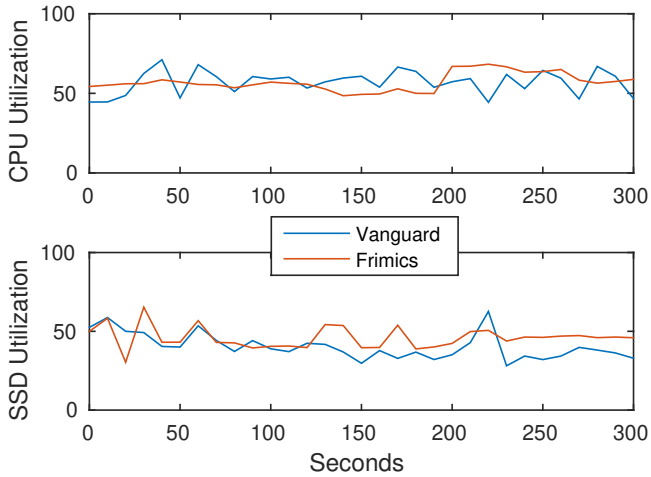
According to Figure 2.12(b), it is the better isolation mechanism of *FRIMICS* that leads to such a performance improvement. Compared to Vanguard, *FRIMICS* mitigates even more application interference because it isolates more resources. Therefore, it further hedges the degradation of performance for consolidated applications which essentially leads to better application performance.

2.8.2 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 (42 GB), and SSD cache (32 GB) 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 2.13 shows the error rate, which we calculate using Function 2.2, 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.



(a) Application *Performance Index*



(b) CPU and I/O utilization

Figure 2.12: Application QoS and server utilization for Vanguard vs. *FRIMICS*.

2.8.3 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

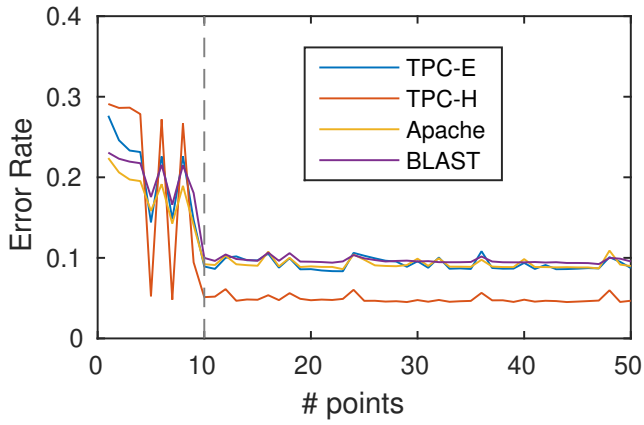


Figure 2.13: Impact on the accuracy of the profiler as we increase the number of training points. The figure displays the first 50 points because the error converges afterwards.

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 2.4.

After 150 seconds of execution, a new TPC-H instance arrives and, to emulate misprediction, we assign to it a slice that consists of 2 cores, 4 GB of memory, and 8 GB of SSD cache. Observe, that this configuration uses 2 GB 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, 44 GB of memory, and 72 GB of SSD cache).

Figure 2.14 plots the *Performance Index* for each application during the first 300 seconds of the execution. We note that although the performance impact on the new application is severe, the rest, properly 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 2.14, 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) *FRIM-*

ICS isolation is not ideal and does not completely eliminate interference with co-located applications. (3) Profiles are not 100% accurate.

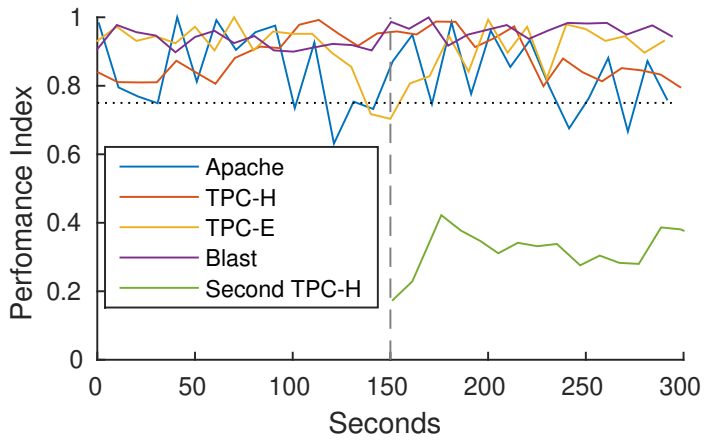


Figure 2.14: Impact of profiler misprediction on application *Performance Index* as the second, mispredicted TPC-H instance is admitted at 150 seconds from the experiment start.

2.8.4 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 four 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 2.4).

Table 2.5 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 similar, overall *Performance Index* for about 16% higher CPU utilization (79% vs. 68%). Table 2.5 also shows

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

Table 2.5: *Performance Index* (PI) and CPU utilization for the user-oriented and provider-oriented policies. The user-oriented policy is configured with thresholds of 1, 0.75, and 0.5.

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.

2.9 *QuMan* 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 2.15 shows the execution history of the same workload we use in Section 2.8.1 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 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 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

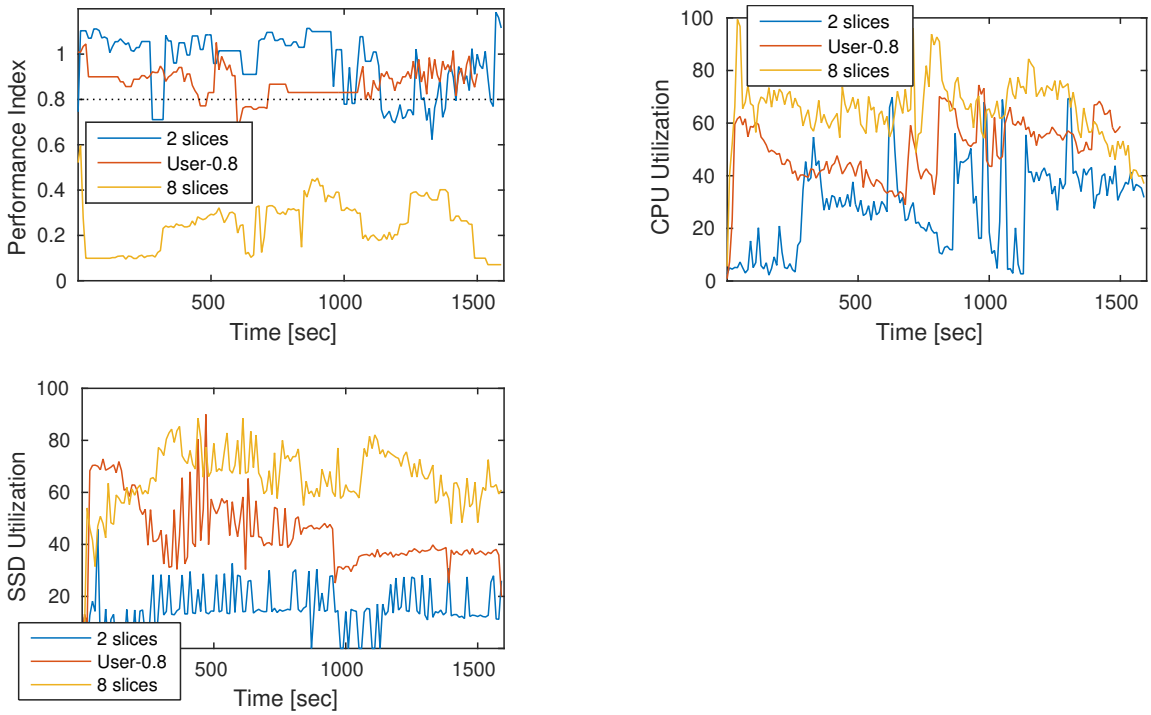


Figure 2.15: Comparison of PI (left), CPU utilization (middle), and SSD Utilization(right) using profile-derived slices vs. fixed-size slices. The profile-derived slices follow the user-oriented policy with threshold 0.8.

of the workload and we run them in isolation (standalone), 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 35 seconds while TPC-E takes 150 seconds, which means that at the 35th BLAST frees 16 cores and Apache will start running because it fits, etc. 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. On the other hand, if we are willing to trade performance for CPU

utilization (provider-oriented policy), we significantly improve utilization.

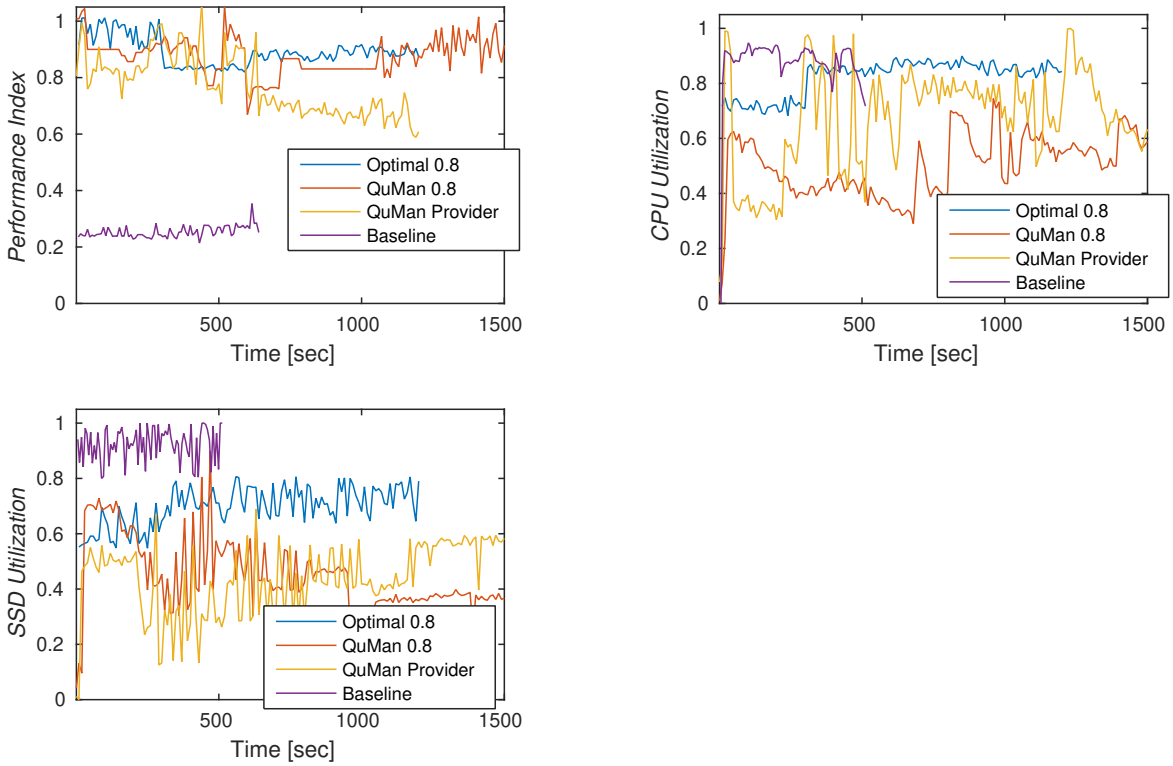


Figure 2.16: Comparison of PI (left), CPU utilization (middle), and SSD Utilization(right) using optimal user 0.8 vs. *QuMan* with user-0.8 mode vs. *QuMan* with provider mode vs. uncoordinated baseline.

2.10 *QuMan* 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/sec for 100 seconds, and the minimum *Performance Index* is set to 0.8. Figure 2.17 shows on the left the average *Performance Index* and on the right the average CPU utilization of active jobs. As we observe from Figure 2.17(b), all 3 systems operate at high load and they all achieve the same levels of 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.

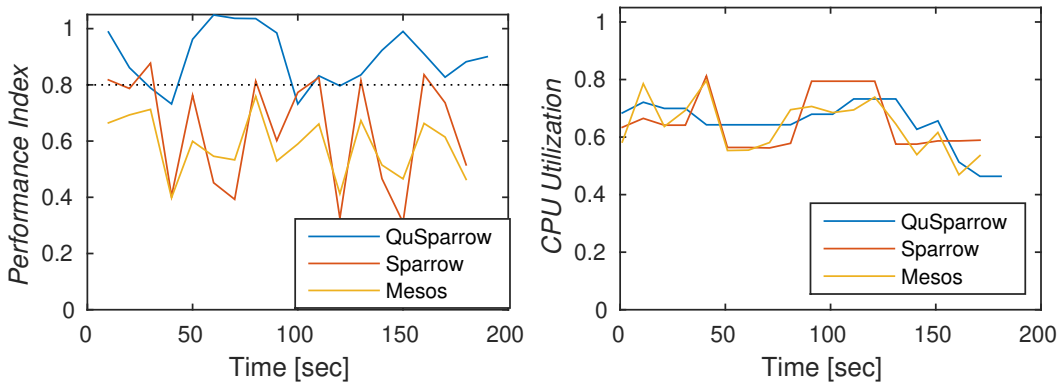


Figure 2.17: Performance Index (left) and average CPU utilization (right) of 5 servers running tasks through QuSparrow, Sparrow, and Mesos.

2.10.1 Cloud Deployment

In our large cluster deployment, a client submits jobs at a rate of 2 jobs per second for 300 seconds 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, to achieve uniform load across all servers, we show a case with a heavy load to keep all servers highly utilized at a steady state. The cluster achieves a steady workload state after 100 seconds, where both systems have similar CPU utilization.

Figure 2.18(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 2.18(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.

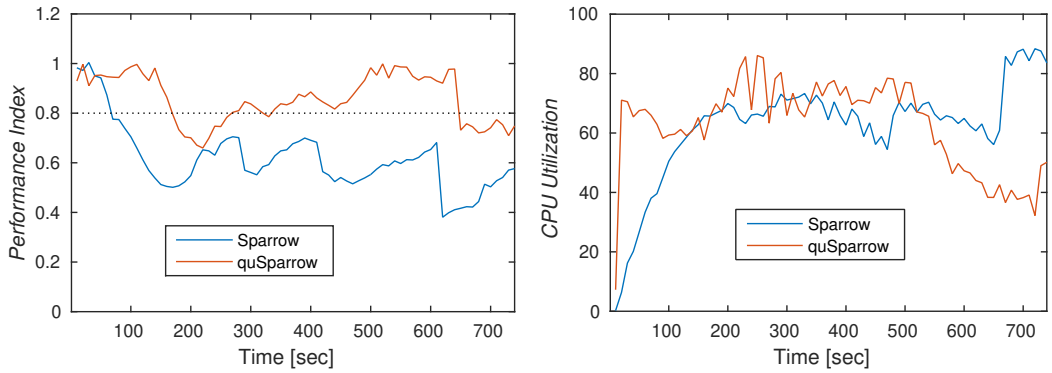


Figure 2.18: Performance Index (left) and average CPU utilization (right) of 100 AWS instances running tasks through QuSparrow and Sparrow.

2.11 Summary

Server utilization in modern data centers has emerged as an important challenge due to both cost, power and technology limitations. In this chapter we propose a new approach that increases server utilization in a cluster, while it keeps under control the degradation of application performance. We present 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 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 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.

Chapter 3

Reactive Resource Adaptation

3.1 Trace-driven Workload Generation and Execution

In this section, we first provide insight into a typical trace; then, we categorize the events it represents and explain how to convert these events into parameters for our model.

3.1.1 Trace Specification

We summarize the notions used in a trace as follows:

- A *task* is an indivisible unit of work that executes on single processing unit. Task duration may vary significantly across tasks, from milliseconds to hours.
- A *job* is a set of tasks. For instance, in a web server, each user request is a job and consists of many tasks. Different Spark jobs in a Spark application appear in the trace as individual jobs.
- An *application* is a set of jobs that execute in batch mode, i.e. we are interested in the completion time of the full application and not individual jobs or tasks.
- A *service* is a set of user-facing jobs, i.e. we are interested in the completion time of individual jobs, as well as the job rate. Typically, services are assumed to run continuously.
- A *workload* is a set of applications and services.

Based on the information available in the traces we examine, Google 2011 [101], Alibaba [82], and Google 2019 [119], we summarize the main events captured by traces as follows:

- *Job events* represent changes in the state of a job, e.g. when a job is submitted or begins execution.
- *Task events* represent changes in the state of a task, similar to jobs. Task events may also contain constraints, e.g. when a task should (not) run on a specific server or task affinity with data.
- *Machine events* represent changes in the hardware or the software of the infrastructure, e.g. when a server is added, a kernel is updated, or a server fails. Machine, events, may also contain machine attributes, e.g. the amount of DRAM available in a server.
- *Resource events* represent the resources reserved or used by jobs and tasks within the interval, e.g. average CPU utilization over 10s. We exclude from this category events that refer to cumulative machine use, and instead, we include these in machine events.

Job, task, machine, and resource reservation events are point events, whereas resource usage events are periodic and refer to intervals. The above categorization, which is the default in traces, mixes inherent workload events with events that depend on the infrastructure and the scheduler. Workload events depend originate exclusively from users, while others depend on the executing environment. In the context of this chapter, such categorization does not help in modeling the generation and execution of trace-driven synthesized workloads. Apart from that, there is information in the trace that does not add value to a workload generator, e.g., the user's username that submitted a job. Therefore, propose a different categorization for the trace that is more suitable for our goal:

- *Workload events*, about *inherent* workload characteristics. They include the submission times of jobs and tasks.
- *Execution events*, with information about the *induced* execution in a specific environment. They include the schedule time and the finish time of a job/task.

3.1.2 Workload and Execution Events

Next, we show how to select the workload and execution events of a workload. We describe our event selection procedure specifically for the Google '11 trace. We follow a similar procedure for all traces we study and collectively show our findings in Table 3.1

Table 3.1: Selected events of Google '11, Alibaba, and Google '19 traces. The terminology among traces differs slightly, however, it is straightforward to map it to the one we use below.

Category	Type	Information
Workload	Job submitted	[time, job-id, sched class]
Execution	Job scheduled, finished	[time, job-id, priority]
	Job usage	[start, end, job-id, task-id, resource usage, system metrics]
	Task submitted, scheduled, finished	[time, job-id, task-id, resources]

Unlike the rest events, job submit events are affected neither by the underlying infrastructure nor by the job/task scheduler. They originate from user requests and are static to recorded workloads. Therefore, we consider the job submit as workload events, while the rest as execution events. We choose only the events in the common path of a workload execution from the rest events, i.e. job "schedule, finish" and task "submit, schedule, finish" events. Additionally, we select information about the event time, and the type of the jobs, either batch or UF, for the submit events. For the schedule and finish events, we select information about the event time, the resource usage, and the system metrics they cause. We omit the events that concern failures or kill events because they are specific to the recorded workload execution. Hence, they do not contribute to the generalization of workload execution.

3.2 *Tracy* Model

In our work, we model a workload as a set of running tasks with certain parameters. To simplify the modeling procedure and without losing generality, we consider that a workload is a mix of independent, batch, and user-facing jobs, originating either from applications or services. We also assume that all tasks in each job are identical, therefore, tasks have

the same duration and execute the same code. Typically jobs today, repetitively perform similar tasks. For instance, a Spark job contains tasks that perform the same computation on different partitions of a Resilient Distributed Dataset (RDD) [134]. Different jobs consist of different tasks. Therefore, our model consists of the following entities:

- A *workload W* is a list of jobs along with their arrival time [Job, J_{AT}].
- A *job J* is a tuple [$J, J_N, J_D, B/UF, (Task, T_{AT})$], where J is the job type, J_N is the number of job tasks, J_D is the duration of the job, B/UF indicates if a job is batch or UF, and [$Task, T_{AT}$] is a list of task instances along with their arrival time.
- A *task T* is a tuple [T, T_D, T_R], where T is the task type, T_D is the duration of the task, and T_R are the resource allocation requests.

Table 3.2 summarizes the parameters we use in our model and their correspondence to trace events. $J_{AT}[UF]$ and $J_{AT}[B]$ parameters correspond to the timestamp of a submit job event of UF and batch jobs. $J_N[UF]$ and $J_N[B]$ parameters correspond to the number of finish events of tasks belonging to the same UF and batch job. $T_{AT}[UF]$ and $T_{AT}[B]$ parameters are the timestamps of a submit UF and batch task event. Finally, $T_D[UF]$, $T_D[B]$, $J_D[UF]$, and $J_D[B]$ are calculated as the difference in the timestamp of schedule and finish events for UF and batch tasks and jobs respectively.

Table 3.2: Parameter definition and PDF estimation for Google 11, Alibaba, and Google 19 traces.

	Parameters	Description	Trace event	Google '11	Alibaba	Google '19
Workload	$J_{AT}[UF]$	UF job arrival time	Timestamp of submit job event	$Chi^2(0.31, -1.5e^{-10}, 1.26e^{15})$	$Norm(1.25e^{12}, 7.18e^{11})$	$B(0.94, 1.09, -8.14e^{-11}, 2.51e^{12})$
	$J_{AT}[B]$	Batch job arrival time		$R(1.98, 1.25e^{12}, 1.25e^{12})$	$R(713, 79.8, 2.42e^{4})$	$B(0.39, 4565, -7.36e^{-11}, 1.71e^{5})$
	$J_N[UF]$	UF job number of tasks	Task finish event of the same job	$T(0.16, -1.45e^{5}, 6.63e^{7})$	$Exp(-2.51e^{12}, 2e^{12})$	$Norm(-5e^{11}, 7.64e^{11})$
	$J_N[B]$	Batch job number of tasks		$T(0.25, 1.0, 1.53e^{-20})$	$KDE(Norm, 1.97)$	$KDE(Norm, 2.03)$
Execution	$T_{AT}[UF]$	UF task arrival time	Timestamp of submit task event	$T(0.48, 0.006, 0.006)$	$B(212, 2.56e^{4}, -4.75e^{15}, 5.84e^{17})$	$Norm(6.75e^{13}, 6.78e^{14})$
	$T_{AT}[B]$	Batch task arrival time		$KDE(Norm, 2.46)$	$KDE(B, 1.56)$	$KDE(Norm, 1.78)$
	$T_D[UF]$	UF task duration	Difference in the timestamp of schedule and finish task events	$KDE(Norm, 1.91)$	$KDE(B, 2.03)$	$KDE(Norm, 1.98)$
	$T_D[B]$	Batch task duration		$KDE(Norm, 2.1)$	$KDE(B, 2.1)$	$KDE(Norm, 2.09)$
	$J_D[UF]$	UF job duration	Difference in the timestamp of schedule and finish job events	$B(0.28, 4.01e^{3}, -2.33e^{-27}, 3.73e^{3})$	$Chi^2(0.24, -1.13e^{-25}, 21.2)$	$Norm(1.28, 11)$
	$J_D[B]$	Batch job duration		$T(0.18, -1.91e^{5}, 6.8e^{6})$	$B(8506, 16.6, -2.19e^{17}, 2.19e^{17})$	$Norm(-1.91e^{13}, 5.11e^{14})$

Next, we use the traces to extract appropriate values for each model parameter. We model each parameter as an independent random variable. For each trace, we extract the histograms of the events that correspond to our model parameters. Then, we identify the PDFs that best fit the histogram in whole or piece-wise and we use these PDFs as the value distributions of our model parameters. Table 3.2 summarizes the PDF that corresponds to each model parameter for each trace.

Depending on the event histogram, we follow two different methods. If the histogram matches a common probability distribution, such as Normal, R, Chi-squared, T, Beta, Log-normal, Gamma, F, Exponential, Cauchy, Laplace, log-gamma, Chi, we apply Parametric Density Estimation (PDE) [122, 50] to calculate its specific parameters, such as mean value and variance. To figure out which PDF to use, we perform multiple PDE tests with different PDF types and select the best fitting PDF that exhibits the minimum distance (least squares) with the given data-set.

If the minimum distance is above 0.5, we consider that the histogram does not match a single probability distribution, and we resort to a Non-parametric Density Estimation (NDE) [69] technique, Kernel Density Estimation (KDE). KDE [60] models random variables as the concatenation of multiple instances of a single PDF kernel. KDE divides the histogram into fixed-size intervals (*bandwidth*), with each interval represented by the same kernel and different kernel parameters. KDE first identifies the kernel and bandwidth by examining the histogram [61] and then identifies the kernel parameters in a second step similar to PDE for each interval.

For the Google 2011 trace, we find that $J_{AT}[UF]$ follows a Chi-squared distribution with parameters $(0.31, -1.5e^{-10}, 1.26e15)$. $J_{AT}[UF]$ follows a R distribution with parameters $(1.98, 1.25e^{12}, 1.25e^{12})$. $J_N[UF], J_N[B], T_{AT}[UF]$, and $J_D[B]$ follow a T-distribution with parameters $(0.16, -1.45e^5, 6.63e^7)$, $(0.25, 1.0, 1.53e^{-20})$, $(0.48, 0.006, 0.006)$, and $(0.18, -1.91e^5, 6.8e^6)$. $J_D[UF]$ follows a beta distribution with parameters $(0.18, -1.91e^5, 6.8e^6)$. Finally, for $T_{AT}[B]$, $T_D[UF]$, and $T_D[B]$, we apply KDE because they do not map well to any of PDF types we use for PDE. We find that these parameters match a Gaussian kernel, with respective bandwidths 2.46, 1.91, and 2.1. This section describes how we generate workloads that can execute on existing systems by addressing two main challenges: (a) Scaling

the generated workload to different infrastructure sizes. (b) Selecting the application types to be used for generating the workload tasks. We present a methodology for validating that the micro-architectural characteristics of synthetic workload executions are close to the ones described in the trace. Thus, we ensure that the application mix we select for the execution of workloads is qualitatively close to the execution captured in the trace.

3.2.1 Workload scaling

We intend to run the generated workloads on different setups and infrastructure sizes. Therefore, there is a need to scale the workloads to match the intended infrastructure. The model parameters and their value distributions as extracted from available traces typically refer to large scale infrastructures, with task and job durations that exceed hours or even days, which is not practical or possible to follow on research prototypes and specific research problems. Running the workload on a different infrastructure requires scaling the workload to adjust the number of jobs, job durations, job arrival times, or data-set sizes.

To achieve this, we scale workload parameters proportionally, as follows. We introduce the following scaling factors:

- The total number of jobs in a workload, W_N. With W_N, we control the number of jobs per server and per time unit.
- A parameter for the scaling of job arrival times, W_{SAT}. With W_{SAT}, we control how loaded the servers will be during execution.
- A parameter for the scaling of the duration of batch and UF jobs, J_{SD}. J_{SD} parameter changes the dynamics of the batch and UF jobs in a workload. This parameter is useful to investigate how the infrastructure and the system software copes with changes in the behavior of the workload. For instance, when throughput is more important than latency, i.e. when the duration of batch jobs is significantly higher than UF jobs and vice versa.

3.2.2 Application Selection

In this step, we select the application types that will be used for executing the trace-based workload. We base the selection of applications on several trace events and the characteristics they represent: 1) maximum and average CPU, 2) memory, disk, and network utilization, 3) cycles per instruction and 4) memory accesses per instruction.

First, we process each trace individually to generate histograms for these parameters. Then we perform dimensionality reduction, using Principal Component Analysis (PCA) [74]. This step is essential for application selection because of the large number of parameters, which complicates application type estimation. After dimensionality reduction, we perform PDF estimation, similar to our model parameters (Section 3.2). Table 3.3 shows the resulting parameters with their corresponding PDF type and parameters for each trace.

Note that the procedure to define the trace application pool requires access to various application types. For the purposes of this work, we select the batch applications from the Rodinia benchmark suite [34] and the TPC family [92]. Also, we select services among the following cloud services: NGINX [94], Redis [13], CouchDB [3], and Memcached [11]. However, it is not in the scope of this work to provide representative UF or batch applications. We assume that users provide suitable application pools depending on their use cases. In addition, we do not further differentiate the importance of some application types over others. Therefore, during workload execution, we uniformly select an application out of the application pool our methodology creates.

Table 3.3: Application selection features extracted for WTs.

Parameters	**Description**	**Google '11**	**Alibaba**	**Google '19**
ACU	Avg. CPU util.	$T(0.79, 2.03e^{-11}, 2.7e^{-11})$	$T(0.97, 1.03e^{-11}, 3.7e^{-11})$	$T(0.29, 3.03, 1.33e^{-9}, 6.60e^{-11})$
MU	Avg. Mem. util.	$Norm(0.01, 0.05)$	$B(0.51, 674, -6.26e^{-30}, 2.15)$	$B(0.053, -3.54e^{-32}, 1.23)$
MM	Max Mem. util.	$Gamma(0.053, -3.54e^{-32}, 1.23)$	$Exp(0.0, 4.44)$	N/A
N_{IN}	Net in	N/A	$Exp(0.026, 0.23)$	$Exp(0.516, 0.83)$
N_{OUT}	Net out	N/A	$B(0.18, 11.3, -1.20e^{-25}, 1.38)$	$B(0.5, 203, -3.53e^{-33}, 5.39)$
IO	IO	$B(0.60, 172, -6.73e^{-33}, 1.56)$	N/A	$B(0.31, 490, -9.85e^{-33}, 0.89)$
PG	Page cache use	$Exp(0.016, 0.03)$	N/A	N/A
PG	Page cache use	N/A	N/A	$Gamma(0.6, -1.07e^{-31}, 0.07)$

3.2.3 Similarity validation

Finally, to validate the similarity of the generated execution-based workloads to the corresponding trace characteristics, we capture system metrics from the actual execution of the generated workload and compare them to the trace events for each model parameter. To compare the two data-sets, trace vs. measured, we use the Pearson correlation coefficient [26].

$$r_{xy} = \frac{\sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \overline{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \overline{y})^2}}, \tag{3.1}$$

where n is the total number of samples for both data-sets, x_i is the i-th sample of the first data-set and $\overline{x}$ its corresponding mean value, y_i is the i-th sample of the second data-set and $\overline{y}$ its corresponding mean value. The range of values that r_{xy} can take is $[-1, 1]$.

For this computation, the two data-sets need to have the same size. Therefore, we randomly divide the data-set of the trace to N subsets with a size equal to the synthetic workload data-set, and calculate the Pearson correlation coefficient for all subsets, with N the quotient of the size of the trace data-set to the size of the synthetic data-set. Then, we calculate the mean value of the resulting r_{xy} coefficients. If r_{xy} is close to 1 (or -1), then the two data-sets are highly correlated (positively or negatively). If r_{xy} is close to 0, the two data-sets are not linearly correlated.

3.3 *Tracy* Implementation

In this section, we discuss the implementation of *Tracy*, which is implemented as a Python script that produces execution-based workloads. The user can generate a workload based on one of the trace profiles we study and built-in *Tracy*. A profile mainly contains the type of PDFs and their parameters for the random variables of Table 3.2. Each profile is stored in a separate directory which becomes a parameter to *Tracy*, e.g. ./wlGenerator.py –profile="Google 2011". We expect that *Tracy* will be augmented over time with additional profiles from new traces, based on our methodology of Section 3.2. After specifying a profile, the user can use the scaling parameters to change the specific setup load. There are three

scaling parameters: 1) the number of jobs, 2) factor for job arrival time, and 3) factor for the job duration of batch and UF jobs. The number of jobs defines how many jobs the generator will create, e.g. ./wlGenerator.py -n=80, will create 80 jobs. The factor for job arrival time is a number that is multiplied by the arrival time of a job, e.g. ./wlGenerator.py –wSat=2, will double the arrival times of jobs. Finally, the factor for the duration of the jobs is a number that changes the number of tasks to change the duration of jobs, e.g. ./wlGenerator.py –jSD=2, will double the number of tasks of each job.

Therefore, *Tracy* allows users to execute diverse workloads of the same load factor and run a single workload at different load factors (scales). The emphasis in the former is that every time the generator will select different tasks and other parameters (based on the specific profile PDFs). In the latter case, a user can fix all other parameters (if helpful) and change the induced load. In both cases, the generated loads will run on the given infrastructure using the sample applications provided along with *Tracy*.

Tracy consists of two main modules: the *workload generator* and the *workload executor*. The workload generator receives as input the workload profile, encoded in custom data type **wlProfile**. Further inputs are J, which is the number of desired jobs in the workload. The workload generator (Pseudocode 1) produces a *job sequence* as output. This output contains a sequence of job instances, specifying the characteristics for each instance. The workload generator is used offline to produce a job trace before a test run is performed.

Although we provide *Tracy* with a specific set of applications to use for job and task generation, this set is not hard-coded in the tool and can be changed by a user. In order to do so, a user has to To change the application and task pool a user needs to provide a directory containing the applications and inputs and a table that specifies tasks durations when executing. To calculate a job duration J_D, we create by $N = J_D/T_D$ tasks. The output of the *workload generator* is two files. The first is a sequence of jobs where each line describes a single job as follows. $J = [J_A T, J_T, T]$. The second file, describes the arrival time of tasks within jobs. Each line is list of timestamps defining the arrival of a task. Next, the workload executor parses the generated job sequence and the task arrival sequence to generate and execute the tasks for each job. The tasks within a job execute the same code on the same data.

ALGORITHM 1
Workload generation pseudocode.

```
1: procedure WLGENERATOR(wlProfile WP, int J, int F)
2:     int jc, N := 0                                   ▷job counter, number of tasks, respectively
3:     time ts, t_total, J_D, A_D := 0                  ▷job timestamp, total time, job duration, application duration, respectively
4:     Boolean S                ▷scheduling class, may either be B, indicating batch job, or UF, indicating user-facing job
5:     appType T                ▷Application name, out of pool of codes available to the tool
6:     jobTypeSeq W                                     ▷A sequence of jobs, the workload to output
7:     while jc < J do
8:         rand := random number between 0 and 100
9:         if rand < P_B then
10:            < T, J_N, ts > := generateJob(B, WP, t_total)
11:        else
12:            < T, J_N, ts > := generateJob(UF, WP, t_total)
13:        end if
14:        t_total := ts
15:        W.append(< jc, T, J_N, F, ts >)
16:    end while
17:    return W
18: end procedure
```

Executing workload

```
19: function EXECUTEWORKLOAD(Boolean S, wlProfile WP, time t_total)
20:     if S = batch then
21:         J_D = WP.J_D[B](WP.λ_B)
22:     else
23:         J_D = WP.J_D[UF](WP.λ_UF)
24:     end if
25:     Select an app A of type T and its configuration out of pool of available apps, determine its
        duration A_D.
26:     J_N = J_D/T_D
27:     ts = t_total + WP.J_AT(WP.λ_AT)
28:     return < J, J_N, ts >
29: end function
```

3.4 *Tracy* Experimental Evaluation

In this section, we first evaluate our methodology for PDE, KDE, and application selection. Afterwards, we use *Tracy* to reproduce synthetic workloads based on Google and Alibaba traces. In our experiments, we use a server with three 16-core AMD Opteron 6200 64-bit processors (48 cores in total), running at 2.1 GHz, and 48GB of DDR-III DRAM. For storage, we use a Samsung EVO 850 128 GB SSD.

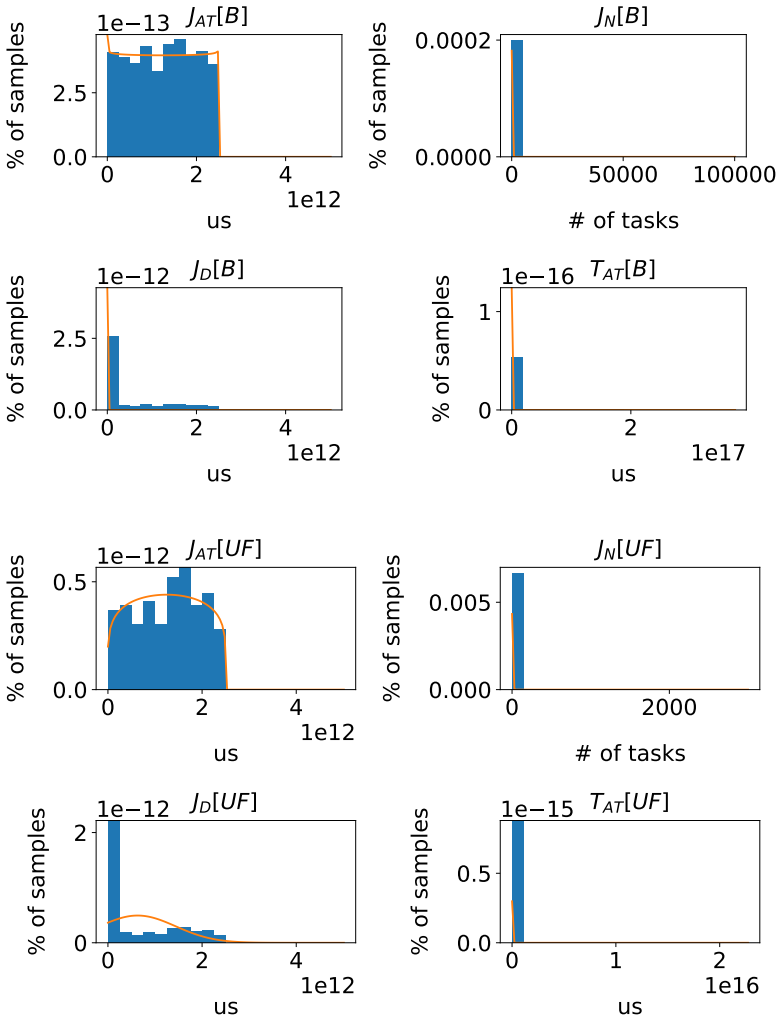


Figure 3.1: Histogram and estimated PDFs using PDE for Google 2011 trace.

3.4.1 Evaluating PDE

This section evaluates the accuracy of the PDFs computed by *Tracy* concerning the histograms of the corresponding parameters extracted from traces. Figure 3.1 compares the histograms of the Google '11 trace parameters with the corresponding PDFs of *Tracy*. We observe that only the $J_D[UF]$ parameter is not very close to the histogram. To measure the similarity between PDFs and histograms, we compute their mean square distance. In Table 3.4, we show the percentage of differences between histograms and PDFs. *Tracy*

achieves the best distance for parameter $J_{AT}[B]$, for which the PDF and the histogram are 94% similar. The worst case is $J_D[UF]$, for which the PDF is only 55% similar to the histogram, and on average, the PDFs are 75% similar to their histograms. Therefore, the characteristics of synthetic workloads of *Tracy* are very close to the ones described in the originating trace.

Table 3.4: Similarity of the workload parameters of PDFs and histograms for the Google 11 trace.

Parameters	**Similarity(%)**
$J_{AT}[B]$	94%
$J_N[B]$	88%
$J_D[B]$	72%
$T_{AT}[B]$	61%
$J_{AT}[UF]$	91%
$J_N[UF]$	78%
$J_D[UF]$	55%
$T_{AT}[UF]$	59%

3.4.2 Dimensionality reduction for application pools

This section shows the results of the dimensionality reduction process of *Tracy* for each trace. *Tracy* reduces a large number of event types in each trace to a smaller number that can be used for application selection using PCA.

Table 3.5 summarizes the importance of each event type as characterized by PCA. Event coefficients in bold indicate the most critical event types for each trace. We show with bold font the events of the traces that we select for application selection. The parameters average CPU and average memory usage are significant for all traces. Apart from that, max memory usage is critical for Google '11 and Alibaba traces while net in and net out for Alibaba and Google '19 trace. Finally, page cache usage and max IO usage are significant for Google '11 trace, while max CPU usage for Google '19. As we observe from the table, at most 6 parameters are sufficient to represent at least 75% of the micro-architectural characteristics of all traces. Cloud applications are diverse and can vary in all of these characteristics significantly. By minimizing the parameter space, we need to search for applications, we

focus only on the trace's critical micro-architectural parameters, which helps in choosing applications more accurately.

Table 3.5: Applying PCA on execution parameters: Importance of each parameter selected for workload execution.

Parameter	**Google '11 (%)**	**Alibaba (%)**	**Google '19 (%)**
avg_cpu	**0.24158018**	**0.17634914**	**0.13583023**
c_mem_usage	**0.16157733**	**0.11970246**	**0.09803796**
page_cache_use	**0.10012959**	0.07177281	N/A
max_mem	**0.16375543**	**0.12004907**	0.05430897
avg_disk_IO	0.07831652	0.05321478	**0.09875985**
avg_disks_space	**0.09512442**	0.0782626	0.03503575
max_cpu	0.02407823	0.0176719	**0.24008818**
max_IO	**0.11280081**	0.07523013	0.04921422
net_in	N/A	**0.12378019**	**0.10734959**
net_out	N/A	**0.14704391**	**0.10299059**
cpi	0.00490664	0.00361279	0.0026702
mai	0.01773085	0.01331022	0.00974135
cpu_distr	N/A	N/A	0.01434824
cpu_tail_distr	N/A	N/A	0.05162486

3.4.3 Emulating the Google and the Alibaba trace

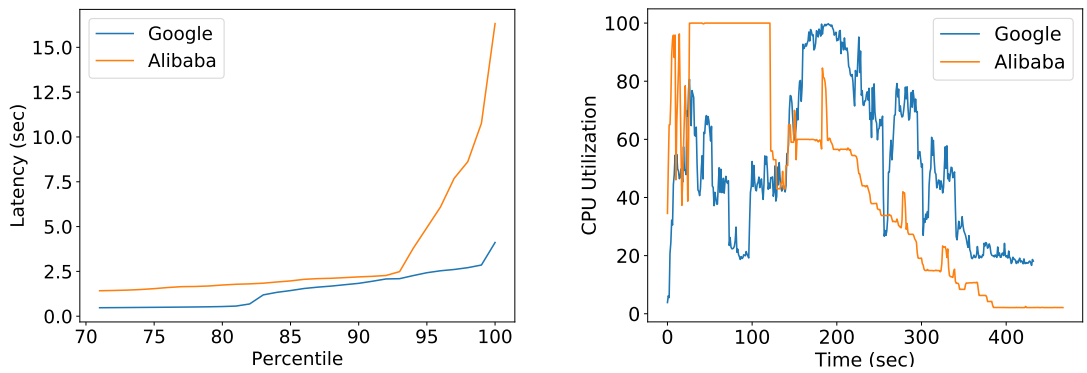
This section generates and executes synthetic workloads of *Tracy*, based on Google and Alibaba. We then validate how representative the execution of these workloads is by comparing the system metrics of the synthetic workload execution to the ones in the trace. Figure 3.2 shows the results of two experiments, where we execute a workload based on the Google trace and a workload based on the Alibaba trace. By applying our validation methodology in both traces, we get a similarity coefficient among all usage parameters above 0.46 and, on average, 0.69. In Table 3.6, we show the correlation coefficients for all parameters for both workloads.

Additionally, comparing the two synthetic workloads, we observe that the Alibaba workload is more bursty than the Google trace, which is why it results in a worse tail latency for the user-facing tasks. In addition, we notice that the workload of the Alibaba trace has 100% load in the first 120 seconds and afterwards it cools down, while on the Google trace

Table 3.6: Validating the micro-architectural parameters of workload executions using Pearson correlation coefficient.

Parameter	Google '11 (r)	Alibaba (r)
avg_cpu	0.79115854	0.69563203
c_mem_usage	0.49827534	0.74002081
page_cache_use	0.62338523	0.87735769
max_mem	0.88640728	0.64538835
avg_disk_IO	0.61440553	0.84108017
avg_disks_space	0.46303910	0.83694352
max_cpu	0.83810822	0.82183377
max_IO	0.48976773	0.83368662
net_in	0.65470367	0.65492767
net_out	0.61298042	0.56783947
cpi	0.70622917	0.75724888
mai	0.53135785	0.63638167
cpu_distr	0.77015809	0.73942622
cpu_tail_distr	0.73790012	0.82886921

the load is more balanced over time.



(a) Tail latency (b) CPU utilization

Figure 3.2: Two Workloads inspired by the Google and the Alibaba trace. They emulate the events of the average server of each datacenter trace. On the left we see the tail latency of the tasks, while on the right we plot the observed CPU utilization.

Google According to the Google trace analysis [100], the average server in the datacenter is 50% utilized, the batch to user-facing job duration ratio is 0.5, and it contains 38% batch jobs. We create a workload using *Tracy* that follows the statistics of the Google trace for a

single server and select 35s for the average batch job duration. To scale down the workload, we estimate the scaling factors of *Tracy*, i.e. number of jobs, job arrival time, duration ratio of batch versus UF jobs, by calculating the corresponding values of Google servers in average. The workload that *Tracy* produces causes 52% CPU utilization and average job arrival time 10s.

The trace released by Google has been studied extensively in several publications e.g. [79, 100, 80, 47, 17, 101, 38, 91]). The trace has over 670, 000 jobs and 25 million tasks executed over 12, 500 hosts during 1-month time period [48]. Around 40% of submissions recorded are less than 10 milliseconds after the previous submission even though the median arrival period is 900 ms. The tail of the arrival times distribution is power-law-like, though the maximum job arrival period is only 11 minutes. Jobs shorter than two hours account represent more than 95% of the jobs, and half of the jobs run for less than 3 minutes. The majority of jobs runs for less than 15 minutes [80].

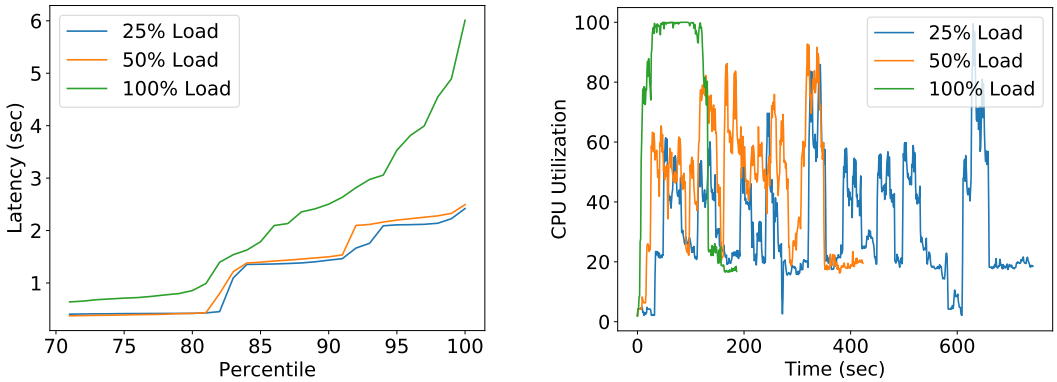
Alibaba Correspondingly, in Alibaba trace, the average server is 40% utilized, the batch to user-facing job duration ratio is 0.05, and it contains 53% batch jobs. Similarly, we create a workload with *Tracy*, selecting 4 seconds for the average batch job duration. The workload results in 45% CPU utilization and average job arrival time 680 ms. In both cases, the workload generated by *Tracy* is very close to the average statistics of servers as per the original traces.

The Alibaba trace is analyzed in [82]. It contains 11089 user-facing jobs and 12951 batch jobs, which run over a time period of 12 hours. This places the batch job versus the user-facing job ratio at 53.9% to 46.1%.

User-facing jobs in the Alibaba trace are long-running service jobs, in this particular case spanning the entire duration of the trace. On the contrary, batch jobs in the trace are predominantly short-running, with about 90% of batch jobs running in less than 0.19 of an hour, while a total of 98.1% of batch jobs runs in less than an hour. Overall, out of the total of jobs in the trace, 47.2% are long jobs, whether batch or user-facing.

3.4.4 Investigating how scaling affects the tail latency

Our goal in this experiment is to show that with *Tracy*, we can easily explore how the workload is affected when we change one of its scaling factors while keeping the other the same. In this experiment, we produce a workload that emulates the workload of a Google server according the Google trace on average. We examine how scaling the job arrival time affects the tail latency of jobs by running the workload 3 times with a different value scaling factor of J_{AT}.



(a) Tail latency (b) CPU utilization

Figure 3.3: The trade-off between CPU utilization and tail latency. The left figure shows the impact on the tail latency of user-facing applications as we increase the load in the system (25%, 50%, and 100%) load. The right figure shows the corresponding CPU utilization achieved.

In Figure 3.3, we plot two graphs that summarize the three runs described above. On the left graph, we show the tail latency of all the user-facing tasks starting from 70th to the 100th percentile, while on the right graph, we see the corresponding CPU utilization of the system. In the first experiment (green line), we target a load of 25% on average in the system. In the second (orange line), we target 50% load, and in the third (blue line), we target 100%. The average utilization of the system is 31%, 46%, and 95% respectively, which indicates that we can successfully control the utilization of the system with *Tracy*, by just changing the J_{AT} scaling factor. In addition, we observe that the total execution time of the experiment changes almost in reverse proportion to the load of the system. For 25%

load in the system the experiment finishes in 700 seconds, for 50% load in 400 seconds, and 200 seconds for 100% load.

We observe that tail latency is not affected for the runs with 25% and 50% load. However, for the case with 100% load, tail latency suffers a 2x deterioration as we approach the 100-percentile of the tasks. We conclude that it is not straightforward how to increase the utilization of the system while still achieving low tail latency for the user-facing tasks.

3.5 Performance-driven Resource Management for Dynamic Workloads

At a high level, a resource management system today has a straight-forward goal: Given a fixed set of resources (e.g. a datacenter), assign the minimum amount of resources to each application that results in acceptable performance and execute as many applications as possible. At the same time, resource managers need to deal with diverse applications, varying application load, and changing behavior during execution. Modern resource managers involve complex and interacting mechanisms. To categorize different approaches used by state-of-the-art systems, we consider the following dimensions:

Resource estimation refers to performing automatic estimations of application resource requirements.

Online refers to adjusting application resources during execution. Offline systems are more accurate than online ones they cannot cope with dynamic applications.

Self-adaptive refers to adapting the parameters that affect resource management based on application feedback.

Resource types refers to the types of resources, such as CPU, memory, the system can handle. Increasing resource scope comes at a significant increase in the configuration space for resource estimation.

Application sizing and placement refers to how resources for each application are divided into containers and how containers are placed on servers. Typically, this is done using some load-oriented metric and less often considers more sophisticated targets such as interference with other co-located applications or monetary cost.

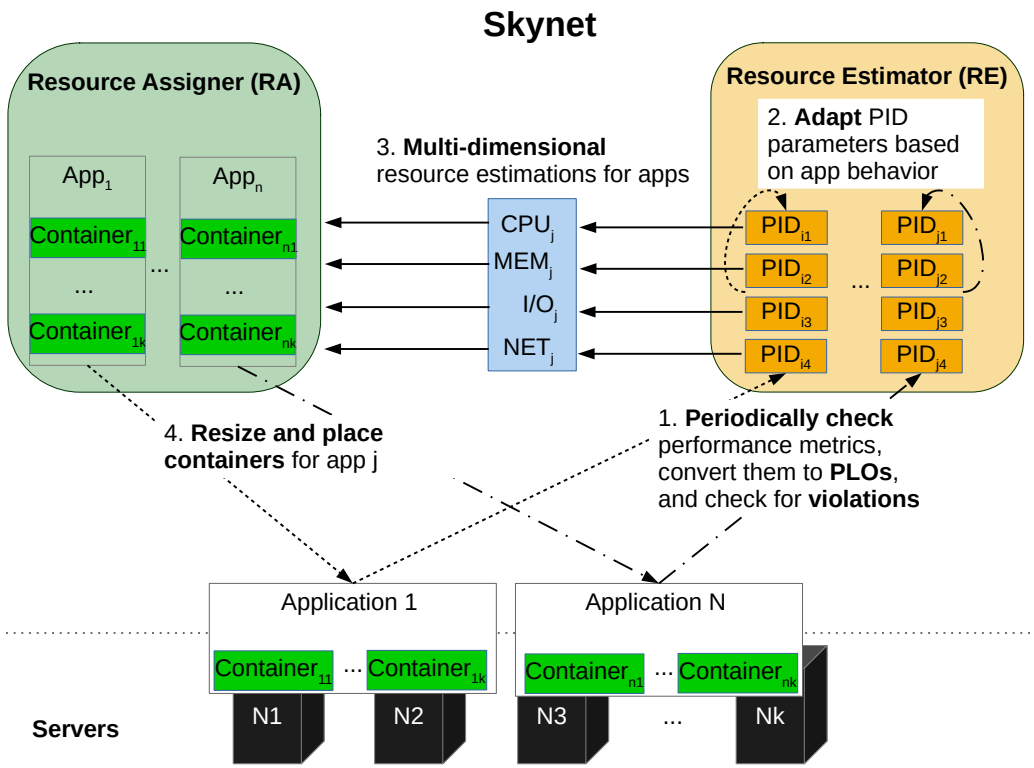


Figure 3.4: Overview of *Skynet* design

Skynet follows a design that estimates resources based on user-defined PLOs, employs online and self-adaptive allocation for multiple resources, and adjusts container sizes by choosing the servers with the maximum load.

3.6 *Skynet* Design

3.6.1 Overview

Figure 4.3 shows the high-level view of the design of *Skynet*. *Skynet* accepts user requests about the deployment of new applications, accompanied by the target PLO. For example, a webserver with a PLO of 1000 requests/s. *Skynet* allocates a predefined container for each new application. During execution, *Skynet* uses its two main components, the Resource Estimator (RE) and the Resource Assigner (RA), to periodically adjust application resources and meet the target PLO, as follows:

1. *Skynet* periodically monitors performance metrics of applications, and in case of a PLO violation, it triggers the RE.
2. The RE adjusts the parameters of the application PIDs based on the PLO.
3. Then, the RE estimates new resource requirements for the application, based on the target PLO.
4. When the resource estimation is available, the RA, which adjusts application containers to match the new resource estimation.

Skynet solves two optimization problems: (1) Which are the minimum aggregate resources required to achieve a performance target (PLO) for a specific application

$$\begin{aligned} \underset{\mathbf{i}}{\text{minimize}} \quad & \mathbf{R_i},\ i = 1, \ldots, m \\ \text{subject to} \quad & P(\mathbf{R_i}) \geq PLO. \end{aligned}$$

(2) Given the aggregate resources required by an application and the current state of the infrastructure, how do we minimize the number of servers required to host the application.

$$\begin{aligned} \underset{s}{\text{minimize}} \quad & \sum N(s),\ N(s) = \begin{cases} 1 & \textit{if } s \textit{ is selected} \\ 0 & \textit{otherwise} \end{cases} \\ \text{subject to} \quad & \sum \mathbf{AR}(s) \geq \mathbf{R_i}, \\ & \sum N(s) \leq S \end{aligned}$$

where $N(s)$ denotes if server s is selected, $\mathbf{AR}(s)$ are the available resources of server s, and S is the total number of servers.

3.6.2 Removing users from the loop

Cloud operators typically defer resource allocation to users, which results in a large percentage of idle resources. Instead, *Skynet* takes over allocation decisions, starting from user-provided PLOs. Given that applications use different metrics, there is a need to support support multiple PLO types. Control loop mechanisms and models, on the other hand, are more efficient when they target a single goal. For this reason, most related work

assumes that all users are interested in the same performance metric, e.g., throughput [114], tail latency [75], or minimizing interference [44]. To bridge different PLO types, *Skynet* optimizes either towards a higher value, e.g., throughput, or a lower value, e.g., latency. To capture both cases, we transform the observed performance of applications as ratios, either *PLO/op* or *op/PLO*, depending on the PLO type. In addition, the target value for each PLO is typically within some feasibility range for each infrastructure. In our work, we assume that providers specify a feasible PLO range for each application (cf. Section 5.2) available to *Skynet*.

3.6.3 Supporting dynamic workloads

To manage dynamic workloads, *Skynet* needs to quickly adjust resources for each application, as its behavior and input change during execution. At the same time, *Skynet* needs to minimize idle resources to host as many applications as possible. This is particularly difficult because:

- Elaborate techniques are not practical because they do not respond quickly to workload changes.
- Applications input changes in unpredictable manners.
- Applications go through different phases, where different resources affect performance.

To adjust resources quickly for diverse applications, *Skynet* uses a PID controller [103] as its main component. PID is generic and lightweight: It can adapt (estimate) functions of different forms and can therefore be suitable for diverse workloads. by tuning its parameters appropriately. In addition, the estimations of PID require a simple calculation (see Equation 3.2), thus, a single CPU core can handle thousands of PID instances.

PID implements a single-input single-output (SISO) feedback control loop. It periodically calculates the difference between the target and the actual values and applies a correction to a knob with a new value $n(t)$, as follows:

$$n(t) = K_p e(t) + K_i \int_0^t e(t)dt + K_d \frac{de(t)}{dt} \tag{3.2}$$

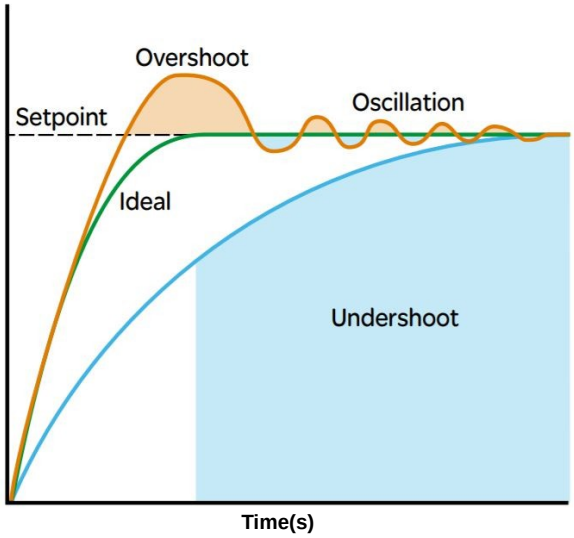


Figure 3.5: Example of a PID execution [8]. The PID controller always achieves the target (setpoint) that we set. Depending on how well we tune its parameters, there are three cases for the PID: (1) it overshoots the setpoint and afterwards oscillates around it, (2) it undershoots and requires multiple steps to reach the setpoint, (3) it is ideal and reaches the point quickly.

where t is the time, $n(t)$ is the new value of the knob, $e(t) = r(t) - y(t)$ is the difference between the target $r(t)$ and observed $y(t)$ values, and K_p, K_i, K_d, are the proportional, integral, and derivative parameters of PID.

Figure 3.5 shows a typical example of a PID execution as a time-series. PID controllers always reach their targets, independent of the use-case and its parameters. However, the steps that PID requires to converge might decrease dramatically, depending on how well its parameters match the use-case. More specifically, there are three cases with PID: (1) Overshoot the target and afterwards oscillate around it if PID parameters are aggressively tuned; (2) Undershoot, resulting in taking many parameter adjustment steps with a conservative parameter tuning to reach the target; and (3) Ideal, with proper tuning of PID parameters. Therefore, an important challenge is how to calculate the PID parameters.

A variety of auto-tuning techniques are available for the PID parameters, including genetic algorithms [77, 51] and machine learning [30, 31]. We choose the augmented Ziegler–Nichols (ZN) tuned (AZNPID) method, an improved and online variation [45] of the lightweight auto-tuning algorithm proposed by Ziegler and Nichols [144]. AZNPID

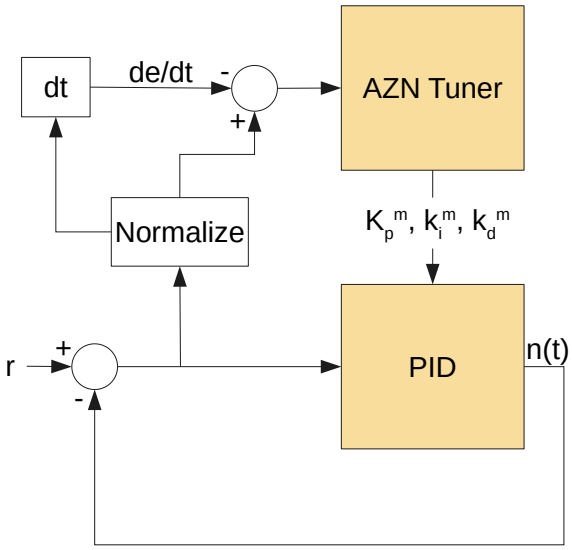


Figure 3.6: Block diagram of AZNPID. It consists of two control loops, the AZN tuner and the PID, that each is affected by the difference of performance with the PLO.

includes an additional control loop for the auto-tuning of the PID parameters K_p, K_i, K_d, which depends on the normalized difference of the observed performance to the target (error) and its derivative. Therefore, PID parameters adapt to the current error and the changes in the error. AZNPID calculates the new knob value $n^m(t)$ as follows:

$$e_N(t) = \frac{e(t)}{|r(t)|} \tag{3.3}$$

$$a(t) = e_N(t)\frac{de_N(t)}{dt} \tag{3.4}$$

$$K_p^m(t) = K_p(1 + |a(t)|) \tag{3.5}$$

$$K_i^m(t) = K_i(1 + a(t)) \tag{3.6}$$

$$K_d^m(t) = K_d(1 + 12|a(t)|) \tag{3.7}$$

$$n^m(t) = K_p^m e(t) + K_i^m \int_0^t e(t)dt + K_d^m \frac{de(t)}{dt} \tag{3.8}$$

where $e_N(t)$ is the normalized error, $a(t)$ is the updating factor of the modified parameters $K_p^m(t)$, $K_i^m(t)$, $K_d^m(t)$, and $n^m(t)$ is the modified knob. Figure 3.6 shows the block diagram of the modified PID.

Note that *Skynet* optimizes the resource usage of applications. It does not optimize

the execution of applications per se. Therefore, *Skynet* does not address the performance bottlenecks that applications might include.

3.6.4 Supporting diverse applications

Cloud applications can exhibit vastly different resource requirements. To cope with such applications, *Skynet* manages CPU, memory, I/O bandwidth, and network bandwidth. To achieve this, *Skynet* addresses the single-output limitation of PID. A single output is only sufficient for applications depending on a single resource, e.g. CPU. *Skynet* extends PID to output multiple values, one for each resource type. However, increasing the dimensions of the control loop the search space of allocations exponentially, making PID less accurate, slower, and less stable. On top of that, storage and network traffic are more challenging to handle because they are affected by request patterns. For this reason, most related work focuses either on CPU or memory only [85, 58, 114, 124, 44] or I/O and network [71, 111, 62, 89].

To reduce the dimensionality of the problem, *Skynet* treats each resource individually. For each application, it instantiates four different PID controllers that each manages the CPU, memory size, I/O bandwidth, and network bandwidth correspondingly. With this approach, we divide the problem into multiple smaller and simpler ones. *Skynet* simultaneously calculates the new estimations for each resource and applies them sequentially in a greedy manner. It sorts the resource estimations in descending order, depending on their expected impact on the performance of the application. We do this by comparing the parameters of each PID of the application. *Skynet* stops applying all the suggested resource modifications in case the application achieves the PLO middle way.

Furthermore, the dedication of resource types to an individual PID indirectly improves the stability of the estimations. Multi-dimensional resource estimations are sensitive to request patterns because of the I/O and the network. However, *Skynet* limits the sensitivity of estimations specifically to I/O and network. Additionally, to improve the robustness and speed of estimations, it considers acceptable resource allocations if they exhibit performance above the PLO and up to 20% more. That way, *Skynet* favors performance over idle

resources.

3.6.5 Placement of applications

After determining the amount of resources an application will use, *Skynet* decides on the size and placement of the respective container. Similar to prior work, *Skynet* deals with this as a variation of the bin packing problem [86]. *Skynet* considers containers as the items to pack, nodes as bins, and resources as volumes. Container sizing and placement are more complicated than bin packing because volumes are multi-dimensional and items do not necessarily fit in the bins.

A typical policy for sizing and placing containers is to maximize the container sizes and spread the load evenly across servers [96, 57, 43, 44, 42, 41]. This is a Least-Load (LL) policy, as it places container instances to the least loaded servers. The goal of such a policy is twofold: avoid co-location interference, and improve application performance. Neither of these goals is directly relevant for *Skynet* because: (1) it uses Linux containers (LXC) [7] on many resource types to mitigate interference, and (2) it performs tight resource estimation based on PLOs. *Skynet* tries to "pack" load instead of "spreading" it evenly. For this purpose, *Skynet* uses a Maximum-Load (ML) policy, that similar to LL, maximizes the container size, however, chooses the maximum loaded servers to host the containers of an application.

The LL policy results in relatively equal resource assignment among servers. Therefore, it typically keeps all the servers of the infrastructure moderately utilized. On the other hand, the ML policy results in an unbalanced resource assignment, where some servers are highly utilized and others remain idle. Both policies reduce network traffic within an application by minimizing the number of spawned containers. However, ML additionally offers the following: (1) reduces fragmentation of resources across servers; hence can admit more applications with high resource demands and (2) allows more room for energy management, e.g. by switching to low-power mode for idle servers.

Skynet implements ML using a best-fit heuristic algorithm to place containers as follows: First, *Skynet* decides the number and size of the containers of an application, based on the estimated resources at each adaptation step. For practical purposes, *Skynet* decides to use

the same size for each resource within an application and tries to minimize the need for moving containers.

Placing new applications: When a new application arrives, *Skynet* scans the servers of the infrastructure to fit all available resources in a single server. In case no server has adequate resources according to the application requirements, we iteratively perform the following steps:

- Increase the number of containers by one.
- Divide the resources equally among the containers.
- Find the highest load servers that can accommodate the containers.
- If we do not find enough servers, repeat the steps.

After *Skynet* determines the servers, the size, and the number of containers, it places the application in the servers for execution.

Resizing applications: Each time there is a request to change the resources of an application, there are three possibilities: (1) fewer resources, (2) more resources that are available on the servers already assigned to the application, (3) more resources that are available in different nodes. In the first case, *Skynet* determines if there are excess containers that can be removed. Then, it sorts the servers in ascending order based on their load and removes the excess containers. In the second case, *Skynet* increases the resources of applications equally among containers to match the new request. *Skynet* handles the third case as a new application placement.

Note that if we keep servers close to 100% utilized, small allocation changes result in migrations. For this reason, *Skynet* allows a slack of unallocated resources (10% of total) on each server and uses this only to increase the allocation of running applications, but not for migrations.

3.6.6 Implementation on Kubernetes

We implement *Skynet* in approximately one thousand LOC in Golang as a Kubernetes custom scheduler. *Skynet* employs an extension of the apiserver (as a pod) to support custom

ALGORITHM 2
Assign resources to applications in nodes.

```
1:  procedure APPLICATIONTONODES(allocs, nodes)
2:      candidateNodes = {}
3:      For each alloc in allocs do
4:          For each node in nodes do //Find the nodes
5:              if fits_in_node(alloc, node) then
6:                  candidateNodes.append(node)
7:              end if
8:          end for
9:          if notEmpty(candidateNodes) then
10:             bestFit(alloc, candidateNodes)
11:             subtractResources(node, alloc)
12:         else
13:             return False
14:         end if
15:     end for
16: end procedure
```

metrics in applications and the Prometheus monitoring system [2]: a database pod to store the metrics and a driver pod that collects metrics. *Skynet* requires applications to include a REST server that exposes all performance metrics of interest to an endpoint (e.g., /metrics) to Prometheus. For our evaluation, we change the container of each application to include a REST server in Node.js, with approximately 50 LOC on average. It is straightforward to deploy *Skynet* for managing dynamic workloads. *Skynet* can either function as an alternative to the default Kubernetes scheduler or in parallel, managing a subset of the workload. The codebase of *Skynet* is publicly available on GitHub [15].

Figure 3.7 illustrates the steps, executed once every second, to capture application metrics: (1) Prometheus collects and stores the metrics of all running pods. (2) In parallel, the metric server gets the container metrics, and stores it locally. (3) Finally, *Skynet* requests the pod metrics from the metric server.

3.7 *Skynet* Evaluation Methodology

We use a local cluster for exploration purposes, and AWS for selected larger-scale experiments.

Private cluster: We use 5 identical servers, each with two 8-core/16-thread Intel Xeon CPU E5-2630 v3 processors (running at 2.10GHz), for 32 hyperthreads in total and 256GB of

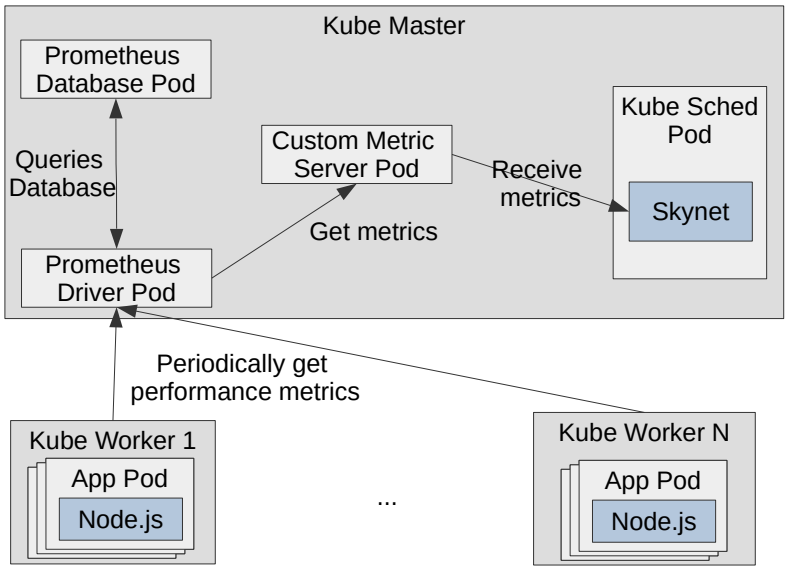


Figure 3.7: We implement *Skynet* as an external pod scheduler, monitoring metrics using Prometheus and a custom metric server. The custom metric server inserts metrics into the metrics database of Prometheus.

memory. All servers have a Samsung 860 Evo SSD installed as their primary storage (and docker container use the same storage device). The interconnect of the servers is a 40Gb/s Ethernet switch.

AWS cloud: We use 50 *c5.metal* machines that constitute a Kubernetes cluster with a single master, and 10 c5.metal machines that generate the workload. These are custom 2nd generation Intel Xeon Scalable Processors (Cascade Lake) with: (1) a sustained all-core turbo frequency of 3.6GHz, (2) 96 cores, (3) 192GB of memory, (4) capability of at least 100, (5) 19,000Mbps of I/O, (6) and up to 25Gb/s network bandwidth.

Native: We use the Kubernetes scheduler unmodified and with no size restrictions in the scheduled pods as our baseline. In the case of Native, the applications run uncontrolled, leading to significant performance variability and PLO violations during execution.

We use six popular applications in our evaluation.

***Redis [13]*:** A key-value memory database. It supports multiple data structures such as strings, hash tables, lists, and sets. This application is CPU and I/O intensive.

***Memcached [11]*:** A distributed memory object caching system, intended to speed up

dynamic web applications by alleviating database query load. This application is memory intensive.

***Spark [135]*:** A general-purpose data processing framework. Spark applications are very diverse and any resource could be important for their performance. We use TPC-H on Spark that is I/O and network heavy.

***CouchDB [3]*:** A distributed document database optimized for handling heavy workloads typical of large fast-growing web and mobile apps. In our evaluation, CouchDB is CPU and I/O intensive.

***Elasticsearch [10]*:** A distributed, RESTful search and analytics engine. This application is CPU and memory intensive.

***Nginx [12]*:** A web server, load balancer and HTTP cache. This application is CPU intensive.

Skynet requires information about the available ranges PLOs of applications to ensure that estimations always converge. We run controlled experiments for each application to determine the feasible PLO ranges on both the AWS and the local cluster. Table 3.7 shows these ranges.

Table 3.7: Feasible value ranges for the PLOs of applications, on our private cluster and AWS.

App	Metric	PLO ranges	
		Private cluster	AWS
Nginx	Latency (ms)	4 - ∞	2 - ∞
Memcached	Ops/s	0 - 200,000	0 - 110,000
TPC-H Spark	Exec Time (s)	50 - ∞	35 - ∞
Elastic Search	Queries/s	0 - 100,000	0 - 550,000
Redis	Ops/s	0 - 15,000	0 - 110,000
CouchDB	Queries/s	0 - 300,000	0 - 2,000,000

We develop a workload generator that produces workloads using as parameters the total number of applications and their feasible PLO ranges. We emulate dynamic workloads as follows: (1) select an application type randomly, (2) select a random PLO, (3) repeat steps 1 and 2 until we create the requested number of applications. All random numbers follow a uniform distribution.

In Figure 3.8, if the application performs better than the PLO, the relative performance

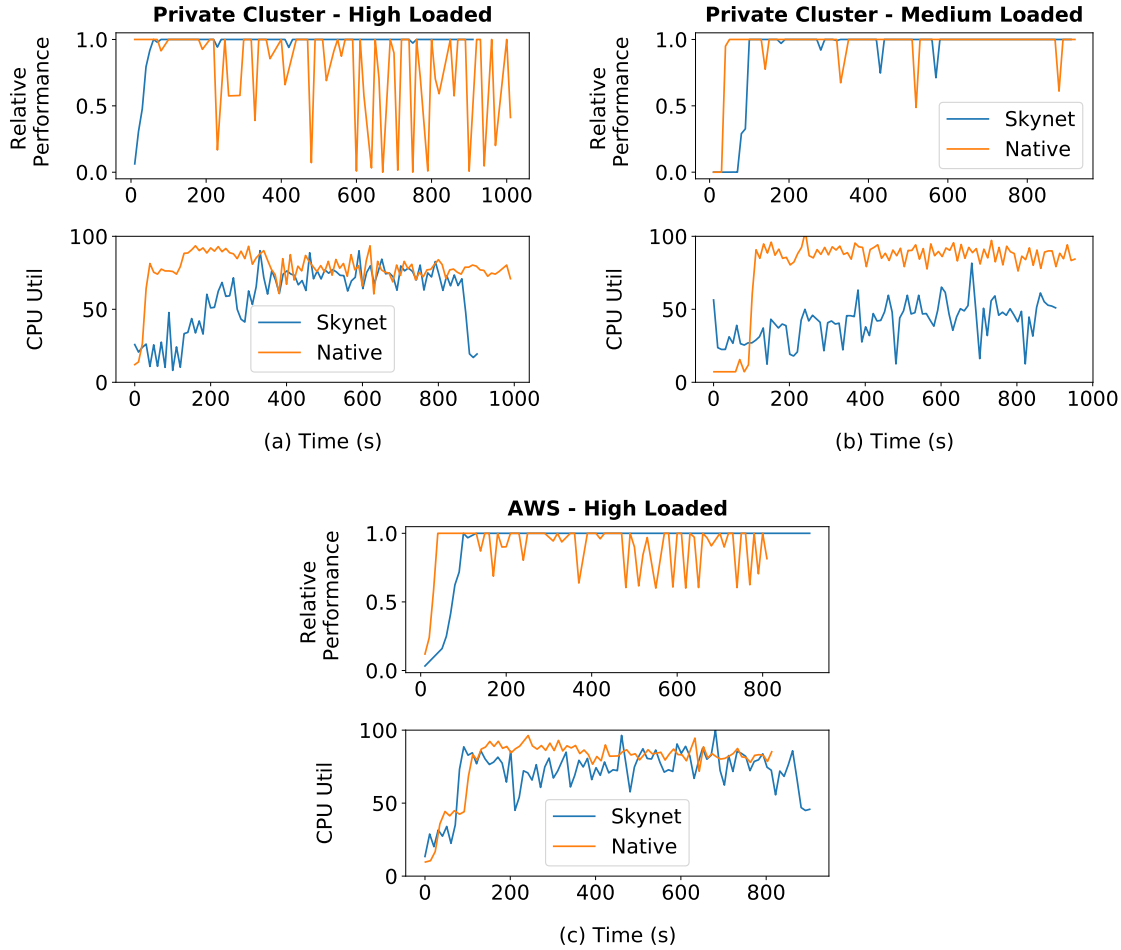


Figure 3.8: Improvement of *Skynet* compared to Native on a private cluster, using (a) a workload that can meet PLOs at high load (about 80% CPU utilization) (left), (b) a workload that can meet PLOs at medium load (about 40% CPU utilization) (middle), and (c) a cluster consisting of 60 bare-metal servers at AWS using a workload that produces high load similar to (a) (right).

equals to 1. In all other cases, we compute the relative performance as described in Section 3.6.2. For each experiment, we generate a random workload, run it 10 times, and report the average of the results. We consider that the workload generates high/medium load when the deployed applications drive infrastructure utilization to 80%/40%, correspondingly.

Applications are either pending or running, depending on whether there are sufficient resources in the cluster. For each running application, we run a corresponding client application (either YCSB [14] using one of the default workloads, or Apache benchmark ab [1]),

with bursty behavior and for a fixed number of total requests. Each client application performs requests in all running applications of the same type. The requests of all YCSB applications follow a Zipfian distribution, while the ab processes constantly issue a uniformly random number of concurrent requests in batches. When all client applications finish their execution, the corresponding running applications terminate; then *Skynet* deploys some of the pending applications in their place. This procedure continues until there are no more pending applications.

Finally, we emulate an *oracle* allocator that allocates resources for applications with almost 100% accuracy. We use a brute-force offline profiling approach, by gathering $48 \times 64 \times 50 \times 100 = 15,360,000$ datapoints for each application (48 for the CPU, 64 for the memory, 50 for the I/O, and 100 for the network). The difference of the observed performance and the PLOs is negligible (less than 2% in all cases). Periodically, every 180s, we change the target for each application and compare *Skynet* to the *oracle*.

3.8 *Skynet* Experimental Evaluation

3.8.1 Meeting PLOs under High Utilization

The goal of this experiment is to show that *Skynet* reduces PLO violations at high load. We compare *Skynet* with Native (unmodified Kubernetes). Figure 3.8(a) shows the results averaged over 10 runs. Both Native and *Skynet* highly utilize the cluster with comparable CPU utilization; however, Native violates PLOs 7.4× more often than *Skynet*. *Skynet* reduces PLO violations from 32.6% to 4.4% for the entire duration of the experiment. The PLO violations of Native are 330s out of 1010s (32.6%), whereas with *Skynet* the violations are 40s out of 910s (4.4%). In addition, with *Skynet*, applications exhibit variance in performance, because *Skynet* adapts their resources according to the workload. On the contrary, Native divides resources equally among running applications, which results in high variations to application performance based on the aggregate workload.

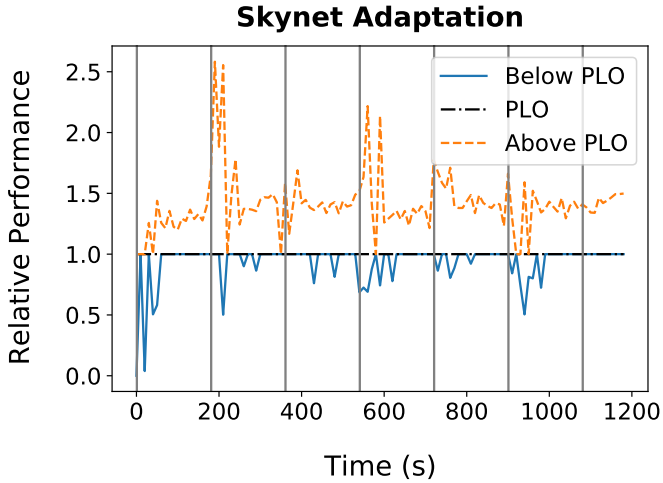


Figure 3.9: Speed of *Skynet* adaptation to changes in the workload. We spawn 80 application instances that serve bursty requests. Every 180s, the number of concurrent requests changes randomly. Despite the dynamic load, *Skynet* minimizes PLO violations for all applications.

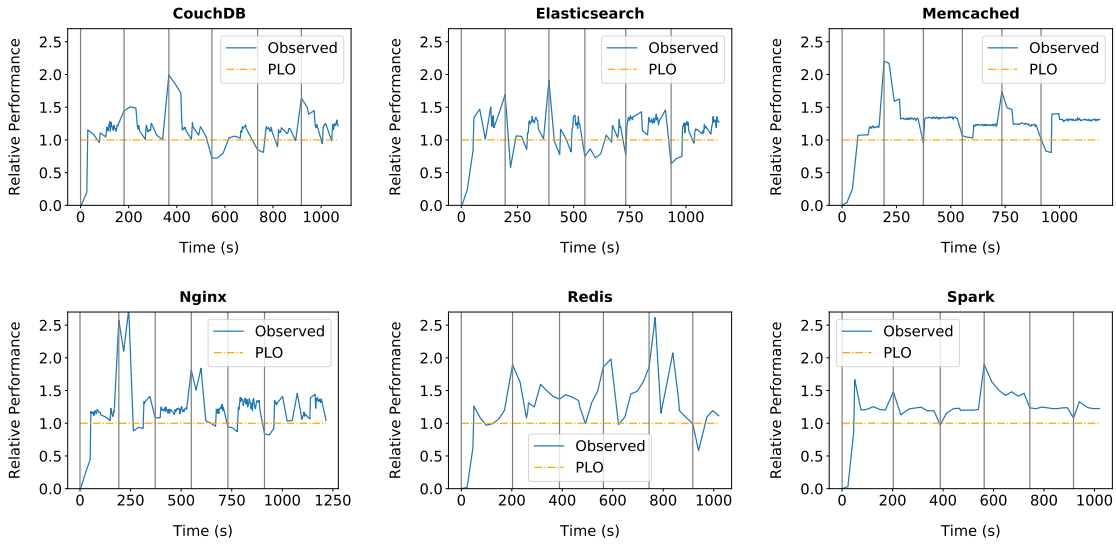
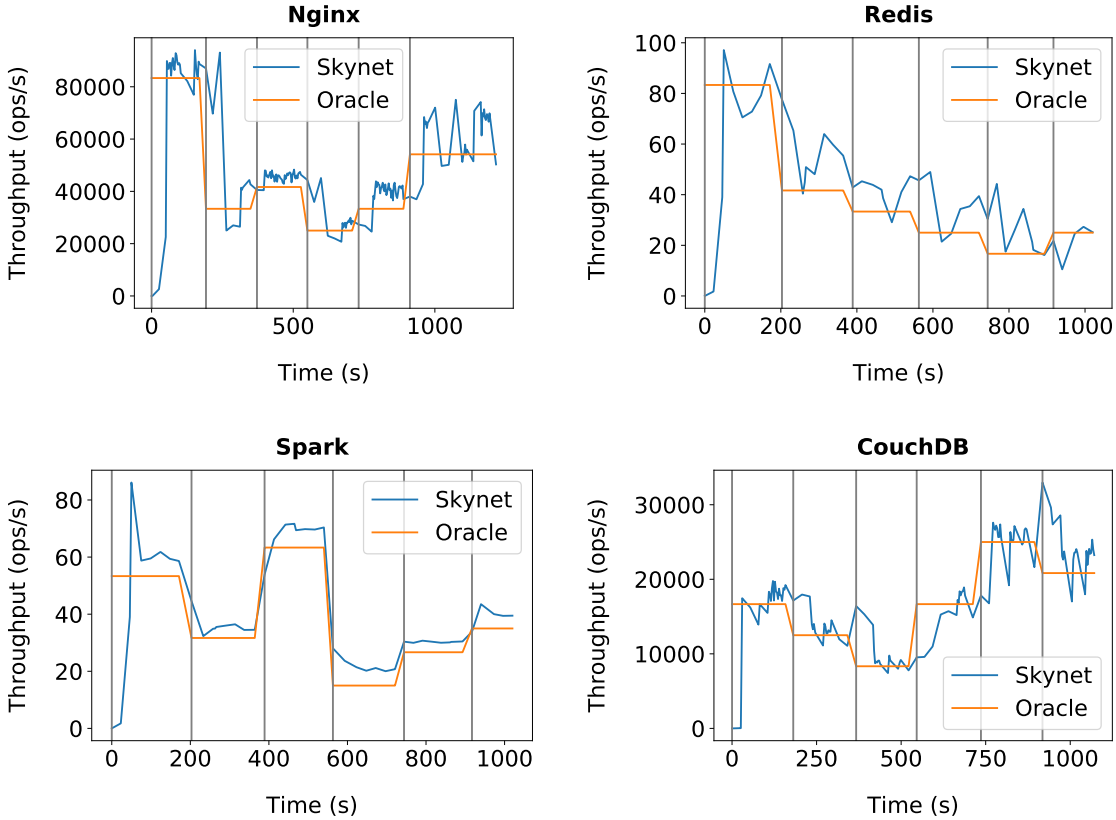


Figure 3.10: Per-application performance relative to their requested PLO over time. We spawn 80 application instances that serve bursty requests. Every 180s, the number of concurrent requests changes randomly. *Skynet* manages to sustain the performance above the PLO for 93.7% of the time.

3.8.2 Minimizing Workload Resources

In this experiment, we show that *Skynet* minimizes the resources allocated to workloads on lowly utilized infrastructure. We compare *Skynet* to Native for a workload that induces



medium load. We show the corresponding average results in Figure 3.8(b). We observe that *Skynet* uses 50% fewer resources to execute such workloads without hurting performance, compared to Native. *Skynet* manages to pack all applications in two and a half nodes and leave room for additional workload. In contrast, Native uniformly distributes the applications on all 5 nodes of the cluster and uses all available resources. Therefore, with Native, the utilization of the infrastructure is more than 90%. Kubernetes does not limit the applications' resources by default; hence, they consume as many resources as possible, performing significantly above their PLOs. Despite utilizing many more resources with Native, we still notice PLO violations for 4.2% (40s out of 950s) of the time, due to the dynamics of the workload mix. *Skynet* increases PLO violations to 6.7% of the time (60s out of 900s), however at significantly lower resource usage. Finally, using *Skynet*, we still over-provision, on average, 13% of the resources to applications. *Skynet* mitigates the over-provisioning in comparison to what is currently typical in the cloud, where users

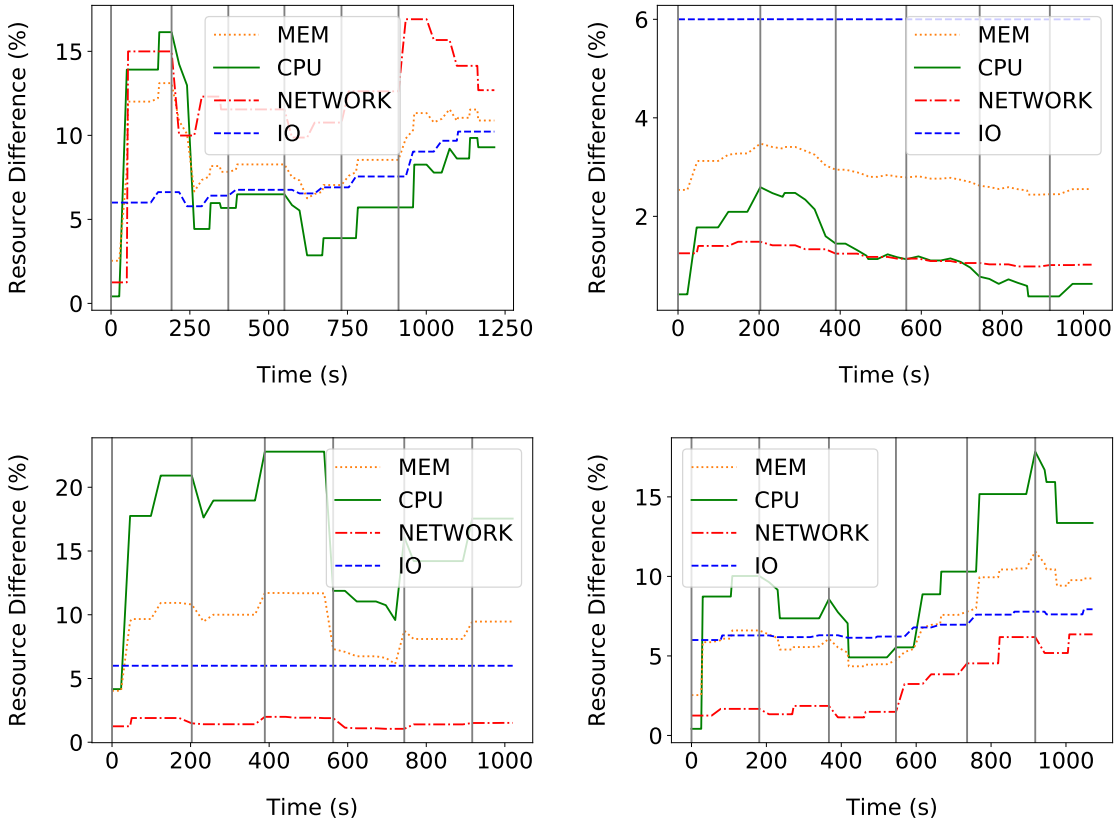


Figure 3.11: Resource adaptation by *Skynet* for Nginx, Spark TPC-H, Redis, and CouchDB. We indicate the resource consumption in the server as a percentage of its total resources. Each application runs in isolation with a single instance.

over-provision applications by 100% [102, 83, 40].

3.8.3 *Skynet* on Public Cloud

We now examine how *Skynet* can estimate and allocate resources in production-level environments using AWS. Figure 3.8(c) shows what happens for a relatively heavy workload that can meet PLOs at about 80% CPU utilization. Regarding CPU utilization, we see that Native results drive it to 85% on average, whereas *Skynet* at 75%. Although Native exhibits 10% higher CPU utilization compared to *Skynet*, it still violates PLOs more than one-third of the experiment duration, 290s out of 800s (36.25%). In comparison, *Skynet* improves violations by more than 5x and results in PLO violations for only 6.6% of the time (60s out

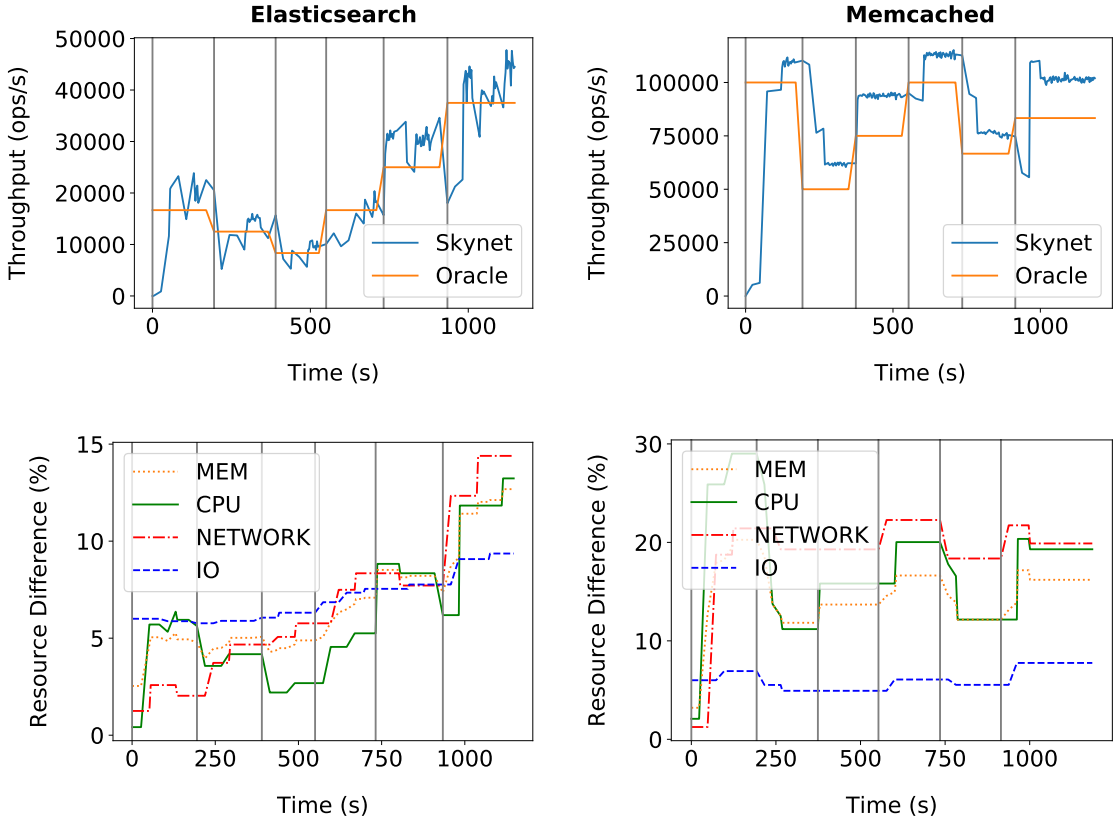


Figure 3.12: Resource adaptation by *Skynet* for Elastic Search and Memcached. We indicate the resource consumption in the server as a percentage of its total resources. Each application runs in isolation with a single container.

Table 3.8: Evaluation of PID. In the left: percentage of time *Skynet* misses the PLOs of applications. In the right: percentage of overprovisioned resources to applications compared to oracle.

App	*Error*	*Below PLO*	CPU, MEM, I/O, NET
Nginx	12%	6.9%	5%, 9%, 30%, 16%
Memcached	10.5%	7.8%	8%, 15%, 20%, 9%
TPC-H Spark	9.5%	3.5%	12%, 9%, 20%, 13%
Elastic Search	13.7%	7.6%	20%, 8%, 23%, 12%
Redis	15%	4.4%	25%, 3%, 14%, 10%
CouchDB	10.5%	7.6%	2%, 7%, 15%, 8%

of 910s). Applications on Native are unrestricted and thus, they are allowed to utilize as many resources as available. Therefore, when consolidating, Native increases utilization by over-achieving PLOs.

3.8.4 Speed of Adaptation

Next, we discuss how resilient *Skynet* is to changes in the workload on a highly loaded infrastructure. We use a workload that changes periodically every 180s (vertical lines in the figure), while maintaining a fixed PLO target. Figure 3.9 displays a time series of the average performance of each deployed application. The dashed line denotes the PLO target and is the achieved relative performance of each application. We observe that, in most cases, *Skynet* requires around 10s to adjust resource allocations and keeps satisfying all PLOs. There are some exceptions, for example, in the time interval 540-900, where a significant amount of containers of the running applications need to migrate, which causes additional overhead. Overall, *Skynet* manages to preserve PLOs for 86.3% of the time, despite the bursty nature of incoming requests.

We examine in more detail what happens to each application in Figure 3.10 as we vary the workload. We observe that, in most cases, *Skynet* reacts to changes quickly and adjusts the resources of each application efficiently (applications perform close to their PLOs). In other cases, such as in CouchDB, around 500s, it does not adjust as well. In the left part of Table 3.8, we summarize the average absolute difference of the PLO and the observed performance for each application. In addition, we show the percentage of time where applications performed below the PLO.

Finally, we evaluate the overhead of PID to estimate new resource allocations and also determine the upper bound of the total time *Skynet* requires to adapt to changes. The total time for the estimation of new resource allocations is less than 250ms. However, the bottleneck for a single step is that we collect new values for the performance metrics every second. Furthermore, *Skynet* requires 2 to 5 steps (new allocation estimates) on average before it converges after a workload change. Therefore, the system can handle workloads that change no more frequently than every few seconds, as it takes on average 3s and 5s in the worst case to adapt.

Note that *Skynet* slightly overprovisions applications to minimize PLO violations (it is visible in both Figures 3.9 and 3.10). In the next section, we evaluate the percentage of the resources that *Skynet* overprovisions compared to an oracle allocator.

3.8.5 Efficient allocation of resources

We now examine how much *Skynet* overprovisions resources compared to an *oracle* allocator, as described in Section 5.2. Figures 3.11 and 3.12 show the results for each application. The top row of Figures 3.11 and 3.12 the difference in the application performance achieved under *Skynet* versus *oracle*. The bottom row of the figures shows the corresponding differences in resource allocation as a percentage between *Skynet* and *oracle*. The average absolute difference in the performance of applications with their corresponding PLOs is 11.9%, while the applications perform below their PLOs for 6.3% of the experiment time on average. We observe that compared to *oracle*, *Skynet* overprovisions by 12% for CPU, 8.5% for memory, 20% for I/O, and 11% for the network. Table 3.8 summarizes in the last column the average difference (resource overprovisioning) of *Skynet* from the *oracle*.

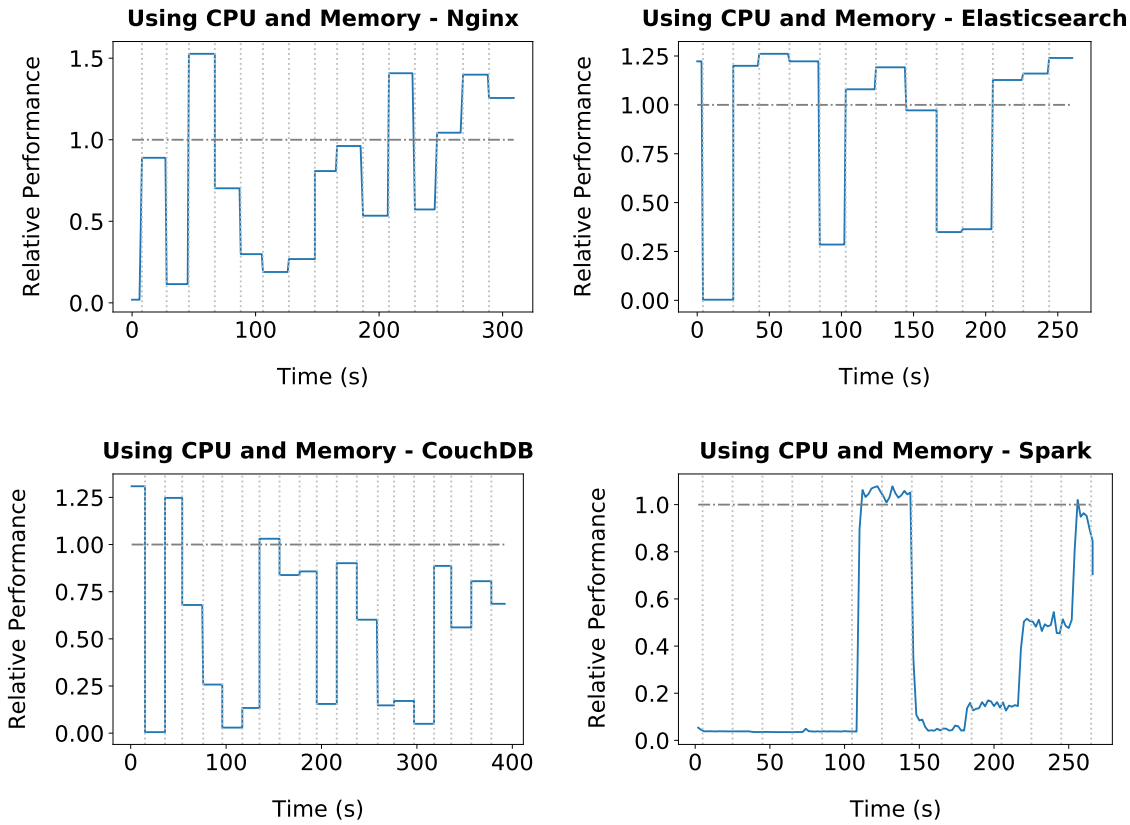


Figure 3.13: Performance variability and degradation when *Skynet* handles only Memory and CPU. The applications are deployed on single server, together with background load that consumes a random percentage of the resources and changes every 20s.

3.8.6 Multi-resource provisioning

We investigate the effects on the performance of applications when *Skynet* only manages CPU and memory using a workload that produces high load. Figure 3.13 shows how performance is affected when we do not manage I/O and network for Nginx, Elastic Search, and CouchDB. Although we allocate sufficient CPU and memory for each application in each case, applications violate PLOs most of the time. The actual performance of each application varies depending on how many I/O and network resources are available. With high consumption of I/O or network bandwidth, then application performance degrades significantly and, even worse, becomes unpredictable.

3.9 Summary

This chapter presents *Skynet*, an adaptive resource allocation and assignment controller for dynamic workloads. *Skynet* removes the user from the resource estimation loop by only requiring a performance-level objective for each application and by building on-the-fly a model of the performance achievable with given amounts of resources. Building this model online allows *Skynet* to adapt to varying input loads and different application phases for each application. We implement *Skynet* as an extension to Kubernetes, making it easy to deploy in existing environments and present an evaluation using dynamic workload mixes. We find that on a cluster operating at high utilization (approximately 80%), *Skynet* reduces PLO violations by up to 5× compared to Kubernetes. Moreover, it saves up to 2× the server resources to run dynamic workloads when the cluster operates at moderate utilization (approximately 40%). Our evaluation shows that *Skynet* is effective for resource management in the data center for dynamic workloads without requiring prior per-workload parameterization.

Chapter 4

Application to Serverless Resource Allocation

4.1 Motivation

Typically, the sequence of steps to execute serverless functions is the following: (1) the user triggers a function invocation. (2) Next, the serverless framework receives the request, spawns new function instances if necessary, and assigns the request to a running instance to execute the serverless function. (3) Afterwards, the serverless function starts executing, and at the end, it notifies the serverless framework that the computation is complete. (4) Finally, the framework responds to the initial request of the users (Figure 4.1).

The main operation of serverless frameworks is to perform autoscaling to match the incoming load towards function instances. Serverless frameworks monitor system metrics such as the concurrent requests served by a function instance or total requests per second. Therefore, the operations to support the autoscaling mechanism do not add significant overhead. In the case of a sudden burst of requests, the framework spawns multiple new function instances to cope with the additional load. However, the new instances require hundreds of milliseconds to be active, which causes a substantial increase in the execution latency. This work minimizes the need to spawn new instances with vertical elasticity and resource estimation.

Quantifying overheads: While the system scales out, the function instances due to an

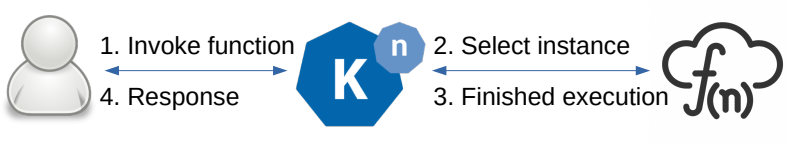


Figure 4.1: The life cycle of a serverless function request, starting from the function invocation caused by users and ending with the response from the serverless framework.

Table 4.1: Overheads of inactive function execution.

	Firecracker	**vHive**
Autoscaler	400us	400us
Instance Creation	**250000us**	**60000us**
Load Balancer	200us	200us
Runtime	2000us	2000us
Total	252600us	62600us

increase in the load, the overall overhead of the framework to spawn a new function instance is roughly two hundred milliseconds. Considering that 90% of serverless functions execute in less than a second [116], at least 20% of the time to run functions is wasted on the serverless framework. Table 4.1 summarizes the time spent on the framework's critical components when running an inactive "no processing" function (cold start). As we can see, for functions that require seconds to execute, the overhead of the serverless framework is negligible. However, for functions that run for tens of milliseconds or less, the dominant factor for the latency is instance creation. Therefore, we cannot tolerate the overhead to spawn new instances for relatively short running functions. Additionally, we observe that the overhead of system-level actions excluding scale-out amounts to less than one millisecond. Therefore, it is negligible for most functions that execute for more than a few milliseconds.

Benefits of vertical elasticity: To reduce the overhead in the function execution caused by horizontal elasticity, we can use vertical elasticity. Figure 4.2 shows the benefits of using vertical elasticity in the typical case to adjust the resources to increase the incoming load compared to horizontal elasticity. We compare two modes of vhive [121] that rely on horizontal elasticity and one for the native Linux container [7] resize mechanism for vertical elasticity. We observe that with horizontal elasticity, the resizing of instances requires at

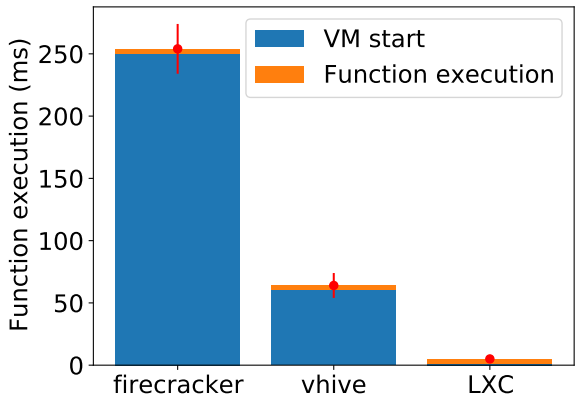


Figure 4.2: Overhead of horizontal elasticity with snapshots compared to the overhead of vertical elasticity.

least 60ms to complete, whereas vertical scaling is less than 50us.

4.2 Design

This section describes the components of *LatEst* (see Figure 4.3) and the technical challenges we need to address. It consists of two main parts: (1) Resource Autoscaler (RA), a mechanism to scale-up resources, and (2) the Resource Estimator (RE), computing estimates of the required resources of serverless functions according to the incoming load. The RE monitors the incoming requests for functions and calculates the required resources to minimize the resulting latency. Next, the RE instructs RA to scale up and/or scale out the corresponding function. In the case of vertical scaling, the RA uses the cgroups [98] Linux kernel feature to increase the resource limits of containers within the VM. In the case of horizontal scaling, the RA uses the Knative autoscaler mechanism to spawn new instances with sufficient resources.

The RA maintains in memory the available resources of each server and the number of allocated resources per running function. During changes in the incoming load of serverless functions, the RE calculates the resources necessary for that specific function. The RE forwards the estimations to the RA, and the RA vertically scales the resources to the running instances. The RE serializes concurrent requests to scale resources. If there are not enough resources in the servers hosting the function, the RA spawns a new function

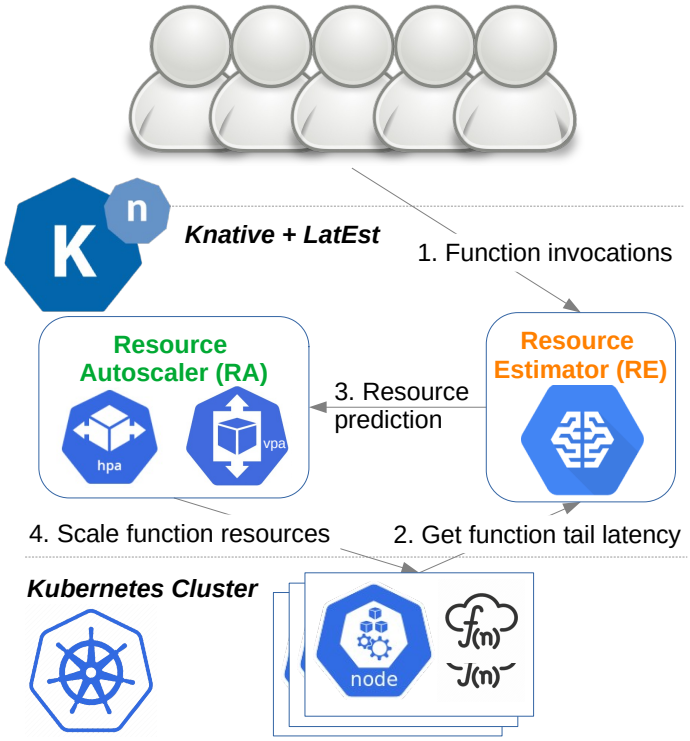


Figure 4.3: Flow diagram of *LatEst*. It consists of two main components: Resource Autoscaler (RA) and Resource Estimator (RE).

instance and places it in the least loaded server.

4.2.1 Resource elasticity

The RA improves the Knative Pod Autoscaler (KPA) that adjusts the resources according to the incoming load. KPA monitors the incoming resources in time windows of constant duration and scales out the resources of serverless functions (horizontal elasticity) based on current requests per second or request concurrency per instance. If the number of incoming requests measured in a time window exceeds a user-defined threshold of total requests/concurrency, KPA spawns a proportional number of new instances. Similarly, if the number of incoming requests is less than the threshold, KPA terminates a proportional number of running instances. The RA makes two optimizations to KPA. First, it does not rely on user input to trigger the autoscaling mechanism. Instead, the RE triggers the scaling mechanism based on its estimations. Second, it scales up the resources of serverless

ALGORITHM 3
Resource Autoscaler algorithm.

```
1: procedure AUTOSCALING(string funcName, Resources estRes, Resources[] availRes,
   Node[] nodes)
2:     int i
3:     int totalAvailCPU := 0                                   ▹the aggregate avail CPU
4:     int totalAvailMem := 0                                   ▹the aggregate avail mem
5:     for i = 0; i < len(availRes); i++ do
6:         totalAvailCPU+ = availRes[i]["CPU"]
7:         totalAvailMem+ = availRes[i]["Mem"]
8:     end for
9:     cpuPerNode = (totalAvailCPU − estRes["CPU"])/len(nodes)
10:    memPerNode = (totalAvailMem − estRes["Mem"])/len(nodes)
11:    if cpuPerNode < 0||memPerNode < 0 then                   ▹Horizontal elasticity
12:        for i = 0; i < len(availRes); i++ do
13:            allocateAllRemainingResources(nodes[i])
14:        end for
15:        node := spawnNewFunctionInstance()
16:        allocateResources(−cpuPerNode ∗ len(nodes), −memPerNode ∗ len(nodes), node)
17:    else                                                     ▹Vertical elasticity
18:        for i = 0; i < len(availRes); i++ do
19:            allocateResources(availRes[i]["CPU"] − cpuPerNode, availRes[i]["Mem"] −
   memPerNode, nodes[i])
20:        end for
21:    end if
22: end procedure
```

ALGORITHM 4
Resource Estimator algorithm.

```
1: procedure ESTIMATERESOURCES(string funcName, int load)
2:     targetLatency := getTargetLatency(funcName)
3:     latency := getCurrentLatency(funcName)
4:     resources := PID(targetLatency, latency, load)
5:     return resources
6: end procedure
```

functions (vertical elasticity) when possible and minimizes the serverless framework's overheads. The Algorithm 3 is the implementation of the RA of *LatEst*.

Horizontal vs vertical elasticity: The accounting of resources with horizontal elasticity is pretty simple. The function instance has a static resource configuration; hence, the number of active instances is sufficient to calculate the total provisioned resources. However, with

LatEst, function instances have dynamic resource configuration due to vertical elasticity. Therefore, with potentially millions of running function instances, it can be a significant overhead to calculate the amount of provisioned resources. To avoid this overhead, we maintain two lists: one for the currently allocated resources per running function and another for the allocated resources per server. Thus, we have flexibility in managing the resources of running function instances.

Additionally, typically container and VM resource configurations are identical for serverless frameworks. Therefore, to vertically scale the resources of a function instance, we need to change the resources of the corresponding container and VM simultaneously. Linux cgroups allow the user to change the resources of a running container instance; however, this is not the case for the firecracker hypervisor, i.e., the resources of a running VM cannot change. Therefore, to achieve vertical elasticity with *LatEst*, we assign all the available cores to VM instances with all cores but one disabled. Consequently, we use the CPU online/offline kernel feature for each function and predefined resources for their corresponding containers (128 millicores and 256MB of memory). By disabling most cores in newly spawned VMs, we avoid burdening the CPU scheduler of the host.

4.2.2 Resource estimation

The RE estimates the least resources that minimize the function tail latency. The number of required resources depends on the current incoming load. *LatEst* uses a feedback loop control, the PID controller, for resource estimations. The PID controller is a single-input single-output controller (SISO) that is lightweight and can accurately estimate the required resources that achieve a specific performance target. It periodically calculates the difference between a targeted value and the actual value of a system it controls and applies a correction to a knob of that system. In our case, the PID controller monitors the tail latency of the serverless function, compares it to a target tail latency, and suggests changes in the allocated resources of the function accordingly. Therefore, a prerequisite for PID is the target latency that is unknown before the actual execution. Initially, the PID remains inactive until we identify the targeted tail latency of the serverless function. Algorithm 4 illustrates how the

RE works.

Finding the tail latency target: The RE must find the minimum feasible tail latency for each function. To achieve that, the RE profiles online the function's execution time as follows: Initially, it starts with a container of 128 millicores and 256MB of memory. Then, it gradually increments the container resources by doubling, tripling, and so forth until we reach a point of diminishing returns. At this point, the RE has defined the tail latency target and proceeds to activate the PID controller to estimate the required resources periodically. This procedure must be repeated for all possible incoming loads since the incoming load can change the possible tail latency. Therefore, we cannot omit this initialization step by the PID controller, as the RE will be continuously observing different mappings of incoming loads and feasible tail latency targets.

4.3 Evaluation

4.3.1 Methodology

Platform: We evaluate *LatEst* on a server with a 24x thread Intel Xeon CPU E5-2620 v2 processor and 128GB of memory running at 2.40GHz and a cluster of 4x such servers. The system disk of each server is a Samsung SSD 950 Pro 128GB PCIe, and its operating system is Ubuntu 18.04.

Serverless functions: We use a no-processing serverless function that prints a message in our experiments.

Request generation: We generate requests for serverless function invocations with a constant rate for periods of 100 seconds. However, the actual rate among different periods might differ.

Tail latency: We consider tail latency the max latency of the 99th percentile (p99) of the incoming requests. In the figures, we show the tail latency as a time-series, and hence, we calculate it in windows of 1 second.

4.3.2 Maintaining latency with vertical elasticity

In this experiment, we show how spawning new function instances can increase the execution latency, whereas increasing the resources of active instances does not affect the latency. We perform 3 experiments as time-series, where we measure the tail latency, the number of active function instances (VMs), and the utilization on a single server of our infrastructure. The first has a static incoming load. In the second, the load incrementally increases by 200 requests every 100 seconds and then decreases (incremental). The load oscillates between 200 requests per second and 1200 requests per second (bursty) in the third. We compare *LatEst*, which uses vertical elasticity (orange and black lines), against vHive, which uses horizontal elasticity (blue and yellow lines).

For the static workload, we observe that *LatEst* and vHive behave similarly in terms of utilization and latency, but vHive has 2 active function instances, whereas *LatEst* has only 1. We experiment with the static workload to validate that *LatEst* does not add additional overhead to the function execution, which is the case. We also observe that when vHive scales-out resources in 2 instances, it does not add additional overhead and is equivalent to scaling-up resources in the same instance. That happens because a single instance is sufficient to sustain the incoming load. Therefore, the time required for the second instance to be active does not affect the latency.

For the incremental workload, we observe that each time the load increases (every 200 seconds), there is an increase up to 8x in the tail latency for vHive when the number of running instances changes. More specifically, in the second 100, the requests change from 200 reqs/s to 400 reqs/s, but the tail latency remains the same because the number of active instances remains the same. However, in the second 200, the number of requests increase from 400 reqs/s to 600 reqs/s, which triggers a scale-out and spawns 2 additional instances, which results in 40ms tail latency that is 8x more compared to the previous period. After 20 seconds, the tail latency goes back to 5ms. Another observation about the behavior of vHive is that in the period between 300 and 400 seconds, where the system spawns a single instance, the tail latency is 10ms, which is only 2x compared to the beginning of that period. On the other hand, the tail latency is not affected by *LatEst*.

Finally, we observe that the increase in the tail latency of function execution is more profound with vHive, up to 20x, during a scale-out for the bursty workload. The reason for that is that the autoscaler spawns up to 6 new instances compared to the 3 instances in the case of the incremental workload. Similarly, like in the case of the incremental workload, the tail latency of the function execution is not affected by *LatEst*. Therefore, we summarize that *LatEst* does not add overhead to the execution. If vertical scaling is feasible in an active function, it can speed up the execution up to 20x compared to horizontal scaling.

4.3.3 Improving latency by predicting the required resources

This experiment evaluates how predictions about the required resources to execute a function improve the resulting tail latency. We compare vHive against *LatEst* without predictions and *LatEst* with predictions using a bursty workload oscillating between 200 req/s and 14400 reqs/s. We measure the number of instances, the allocated resources, and the resulting tail latency as a time-series on the 4-node cluster of our infrastructure. We observe that vHive spawns up to 120 function instances during the burst, which accounts for reserving all the available cores of the 4 nodes and results in 1000ms tail latency. *LatEst*, without predictions, utilizes the same number of cores on only 4 function instances. Therefore, the resulting tail latency is 90ms which is more than 10x better than vHive. However, vHive and *LatEst* without predictions allocate 96 cores (4×24), while only 40 are required. *LatEst* with predictions, allocate only 42 cores, which is very close to the required ones. Hence, it spawns only 2 function instances on 2 out of the 4 nodes and achieves 40ms latency. This latency is still 8x worse than warm function execution, which is 5ms, but it is 25x times better than vHive, which is state-of-the-art.

4.3.4 *LatEst* against the related work

In this section, we summarize the benefits of running serverless functions on top of *LatEst* compared to the related work. Table 4.2 shows the latency of functions, with Firecracker, vHive, and *LatEst*. The table shows 3 columns: (1) the overhead of the frameworks, (2) the latency of a cold start, which includes the time to spawn a new function instance, and (3)

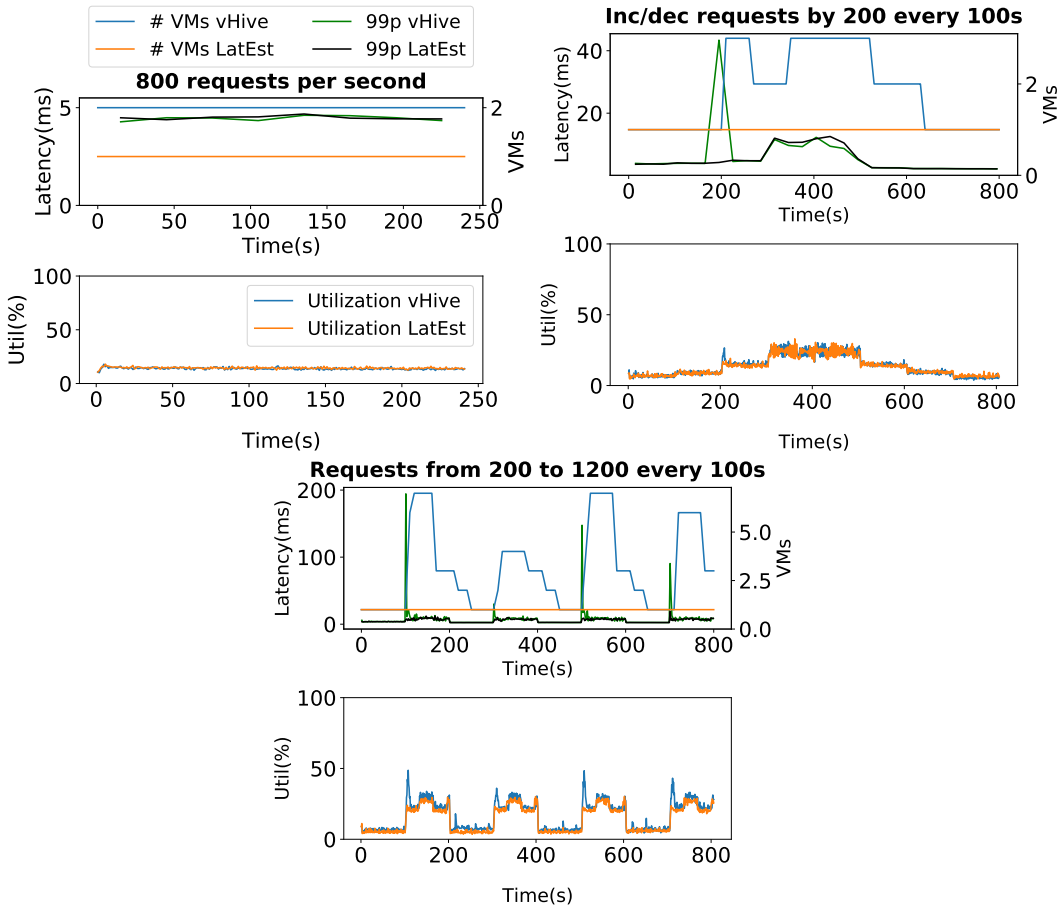


Figure 4.4: Comparing the horizontal elasticity of Native against the vertical elasticity of *LatEst*. The left figure concerns a static workload, and incoming requests are constant at 800 reqs/s. The central figure concerns an incremental workload, and incoming requests increase by 200 every 100 seconds and go from 200 reqs/s to 800 reqs/s. The right figure concerns a bursty workload, and incoming requests oscillate from 200 reqs/s to 1200 reqs/s.

Table 4.2: Serverless function latency on different serverless frameworks. VE means vertical elasticity and HE horizontal elasticity.

	Overhead	Cold start	Warm start
Firecracker	650us	250ms	2.5ms
vHive	850us	60ms	2.7ms
LatEst	550us	2.4ms (VE) / 60ms (HE)	2.4ms

the latency of a warm start. First, we observe that the overheads of the frameworks are similar. Second, for warm starts, all frameworks achieve the same latency. Third, there is a significant difference in cold start function latency. In case vertical elasticity is possible,

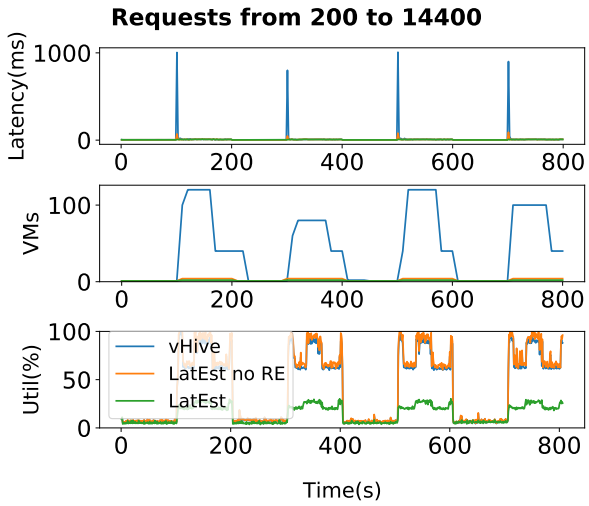


Figure 4.5: The resulting number of instances of vHive against *LatEst* without and with resource predictions, when vertical elasticity is not possible.

LatEst improves the latency of cold starts by 2 orders of magnitude compared to Firecracker and an order of magnitude compared to vHive. In the case of horizontal elasticity, *LatEst* is achieving the same latency as vHive.

4.4 Summary

This chapter proposes using vertical next-to-horizontal elasticity for serverless functions. We limit the spawning of new function instances only to cases where the server resources are depleted. The resulting tail latency of the serverless framework is a few milliseconds. Additionally, we estimate the resources required to maintain functions' execution latency depending on the incoming load. That way, *LatEst* adapts to workload changes quickly and in fewer steps compared to using predefined resource configurations for scaling. This approach minimizes the need for horizontal elasticity, which further reduces the tail latency. Our results show that *LatEst* bursts do not significantly affect the tail latency of function execution. Compared to the state-of-the-art, *LatEst* can reduce the tail latency by up to 25x.

Chapter 5

Cost-aware Cloud Deployment

5.1 Cost-aware Deployment for Dynamic Cloud Services

The factors we in this work to optimize the cost of deployments in the cloud are:

- VM type for each instance.
- VM resource configuration for each instance.
- Cloud provider.
- Number of VM instances.

Figure 5.2 illustrates the high-level design of our *DyRAC*. The system monitors the application to identify changes in the workload and it re-configures the service if that action reduces the cost. More specifically, the RA decides the number of instances of the service, assigns several resources to each instance, selects a provider to host the service, and spawns one VM with the appropriate size for each application instance. The details of the re-configuration procedure are described in Sections 5.1.1, 5.1.2, and 5.1.3.

Services: A *Service* is a distributed and interactive application consisting of multiple instances that cooperate to maintain a user-defined performance level. For our purposes, *DyRAC* expects as input the total number of resources per VM type, across all providers, that satisfy the service performance requirements depending on the current workload.

Dynamic Workloads: Papers in the literature commonly assume, for simplicity, that the workload is either periodic or static [107, 73]. However, the actual challenge is handling

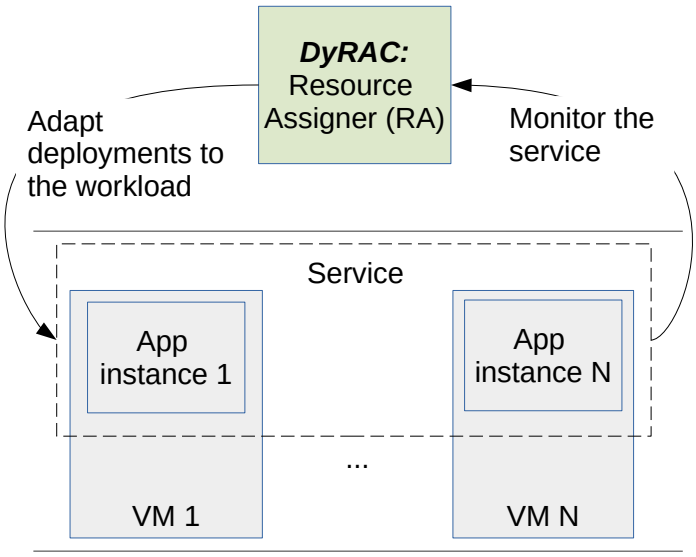


Figure 5.1: Overview of *DyRAC*. A Service consists of multiple application instances. *DyRAC* places each application instance to a single VM. *DyRAC* monitors the application to adapt the service deployment in response to workload changes, deciding the number of VMs to spawn and how to distribute resources among VMs to minimize deployment cost.

bursty and stochastic workloads [32], which is the most common case for services in the cloud. Therefore, *DyRAC* makes no assumptions about the workload.

Cost model: We model the problem of minimizing the cost at a given time, as a variation of the multidimensional knapsack problem (MDKP) [76]. Given a set of item types with a weight and a value, we select a number for each type (possibly zero) so that the total weight remains above the demand of the knapsack and the total value is minimized. In our context, application instances are the items. The weights are the resources assigned to each instance. The value is the cost of the VM hosting the instance. Finally, the total number of the requested resources for a service is the weight demand of the knapsack. We configure the service deployment such that: the total assigned resources are at least what was requested and using the VM sizes that minimize the total cost. Resource assignment includes two additional challenges: (1) the weight demand of knapsack changes when we change the number of allocated resources and (2) there is a potential for waste when we remove items before their charging period ends. We formulate a variation of MDKP is

formulated as follows:

$$\begin{aligned} \min \mathrm{C} &= \sum_{n=1}^{N} c_n x_n \\ \text{subject to} \sum_{n=1}^{N} r_{mn} x_m &\geq l_m, m = 1, \ldots, M \\ 0 &\leq x_n, n = 1, \ldots, N, \end{aligned} \tag{5.1}$$

where N is the number of VM types, M is the number of the different resource types, i.e. CPU and memory, x_n is the number of VM instances per VM type, c_n is the cost of each VM, r_{mn} is the number of resources per resource type per VM type, and l_m is the weight demand. MDKP is NP-hard, hence, we propose four policies described in Section 5.2, that implement simplified greedy approaches [19].

5.1.1 Resource Assignment

RA is a critical and often neglected component when managing the resources of applications. Most related work explores how to place tasks/jobs on the underlying hardware, intending to avoid interference or increase server utilization [64, 84, 39]. Although these are significant issues to address, they partially contribute to minimizing the cost of deploying services in the cloud. We also need to include other fundamental criteria for cost optimization: (1) how to choose the type of VM that hosts a service and (2) how to divide the resources among VM instances.

We explore the impact of different service configurations on the cost of executing a workload. *DyRAC* takes into account that providers charge VMs for quantized periods, e.g. one hour. Therefore, when there is a request for reducing resources in a service, *DyRAC* does not shrink VMs until the minimum charging period expires. In the case of a request for increased resources and depending on the RA policy (see Section 5.2), *DyRAC* decides to resize the running application instances or spawn additional ones. RA determines the configuration that minimizes cost each time there is a demand for additional resources

using the following equation:

$$c(t) = c(t-1) + min(cost(cd), cost(ar)), \tag{5.2}$$

where $c(t)$ is the cost at time t, $cost(cd)$ is the cost to change the configuration, i.e. change VM types or provider, and $cost(ar)$ is the cost to add resources to the service, i.e. spawn additional VMs of the same type.

When *DyRAC* chooses to re-configure a service, it essentially spawns new VMs and, at the same time, keeps the old VMs inactive until their charging interval completes. For example, suppose that a VM has been running for 20 minutes, the VM charging period is one hour, and *DyRAC* chooses to resize it to support an increase in the resource demand of a workload. We add both the old and the new VM for the next 40 minutes in the service's total cost. Therefore, $cost(cd)$ is the cost of the new VMs that result from the service re-configuration plus the old inactive VMs until their charging period is over. To calculate $cost(ar)$, we compute how many additional VMs *DyRAC* spawns.

5.1.2 Provider Selection

Apart from choosing the most cost-efficient deployment, RA also considers changing cloud providers at runtime. This approach has been considered unrealistic due to the slow data migration and the complexity of the software stack. For this reason, most related work about provider selection considers only capacity planning for batch jobs before their execution and not for interactive services [138, 137, 127]. However, with the current pace that storage and network technologies are advancing (e.g. 100Gb/s network throughput, 3GB/s I/O throughput in a single NVMe device) as well as the current trends of modern hypervisors, migrating massive amount of data becomes feasible. It is interesting to explore the possible benefits of migrating a service among multiple providers at runtime. We develop this feature in *DyRAC* to the deployment cost of services while also considering providers. *DyRAC* first calculates the best service configuration for each provider individually and afterwards calculates the cost to change providers, as shown in Equation 5.2, by spawning all currently running active VMs.

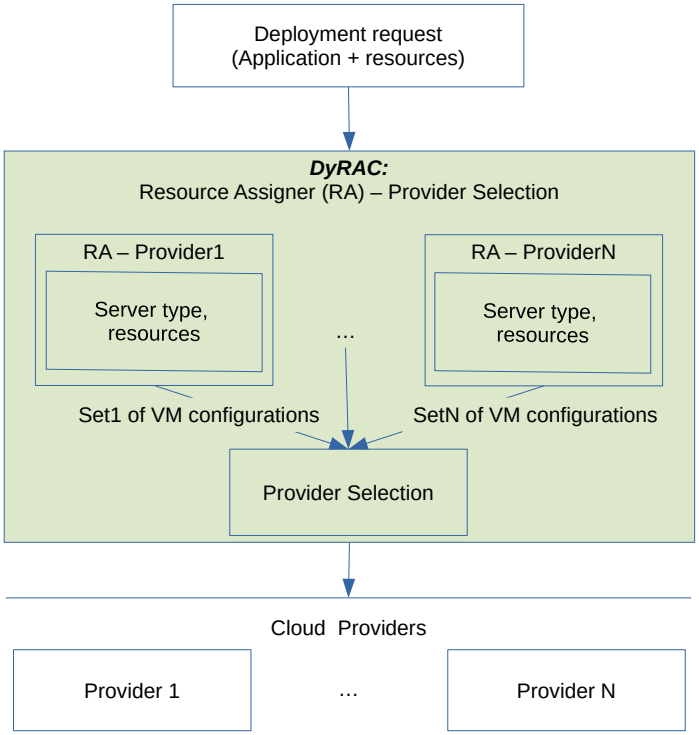


Figure 5.2: *DyRAC* performs its decisions in two steps. In the first step multiple instances of the RA select the most cost-efficient list of VM instances and their configuration for each provider. In the second step, *DyRAC* selects the best list of VM instances among providers.

5.1.3 Reconfiguring Services

DyRAC periodically checks, every thirty seconds, whether there is a more cost-efficient service deployment among providers depending on the current service requirements. *DyRAC* maintains multiple independent RA threads, one for each available cloud provider. Each RA thread periodically calculates the VM configurations with the most benefit in each provider for all deployed services, according to *DyRAC*'s current policy. Afterwards, the RA threads store the selected service deployment as a list of VMs along with their resource configuration. Next, *DyRAC* selects the provider's VM list with the least cost and finally decides whether or not to perform a service re-configuration according to Equation 5.2.

In case *DyRAC* estimates that a selected VM configuration will reduce the total cost, it triggers the procedure of service re-configurations. First, *DyRAC* spawns the necessary additional VMs according to the selected configurations and waits until they all become

ready. Afterwards, *DyRAC* starts service instances in each of the new VMs and transfers the service's current state along with the necessary data for the already running instances. Finally, VMs that are not included in the new deployment remain active until their charging period is over. By not terminating the previous VM instances immediately, we better handle workloads with rapid increase in their resource requirements. During re-configurations the services will have extra resources to handle load transients if necessary.

5.2 Methodology

Our evaluation uses a trace-driven simulation that employs two tools written in Python, the *Tracer*, and the *Simulator*. The Tracer generates traces with resource estimations for a service, matching the output of a dynamic workload that requires changing its resources over-time. The Simulator reads such traces to calculate the most cost-efficient deployment according to a policy. ***Dynamic workloads:*** The Tracer can generate a variety of workloads based on four parameters:

- The resource requirements of the workload: increasing and then decreasing, with oscillations, random.
- The timing of resource changes: periodic, aperiodic.
- The increment of each resource: static, random.
- The total duration of the workload, in seconds.

With the parameters described above, the Tracer can cover a vast variety of datacenter traces [102, 83, 40]. For example, we can create periodic, bursty, or random workloads. Furthermore, we can produce incrementally varying workloads that allow the evaluation of specific RA features in a controlled manner.

For our evaluation, we generate three workload types:

- *Gradual*: It gradually increases the resource demand of the workload statically and periodically.

Table 5.1: Summary of the available policies implemented in *DyRAC*.

Policy	**VM Types**	**VM Count**	**Use-case**
Single VM	Single active	One	Baseline
Adaptive	Multiple	One	Provider B
Fixed	Single	Many	Provider A and C
Type-adaptive	Single active	Many	*DyRAC*

- *Oscillating*: It periodically changes the resource demands of the workload by a random quantity.
- *Random*: It aperiodically changes the resource demand of the workload by a random quantity.

Configuration Policies: The Simulator is extensible with custom configuration policies. We implement three baseline policies and one for *DyRAC* (see Table 5.1):

- *Single VM*. It always maintains one VM with enough resources per service. This policy emulates what a novice cloud user would do.
- *Adaptive*. It maintains multiple VMs per service. Each time RE increases the service's resources, it creates a new VM with the additional resources. This is an example of how an advanced cloud user could manage resources on Provider B, which offers VMs with custom resource configurations.
- *Fixed*. It maintains multiple VMs of one type per service. Each time the RE requests for more resources, it calculates the additional VMs to spawn. This matches Provider A and C, as they offer VMs with fixed resource configurations. We choose a VM type with 1CPU and 1GB memory, resulting in fewer unused resources than larger VMs.
- *Type-adaptive*. It is a combination of the Fixed and Adaptive policies, using multiple VMs of the same type. However, the type can change at runtime if such a decision reduces deployment cost.

When RE indicates to decrease the resources of a service, the RA changes the configuration of services only after the charging period of their VMs is over.

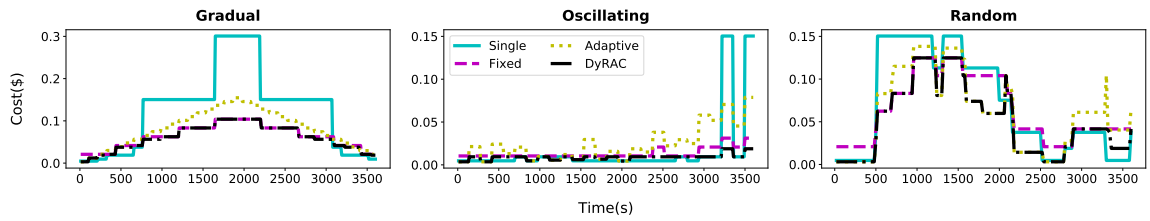


Figure 5.3: Comparing *DyRAC* against 3 baseline policies. The cost reductions with *DyRAC* are up to 33% on average.

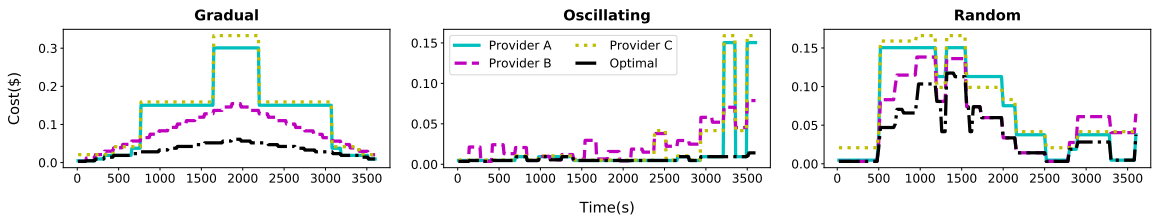


Figure 5.4: Exploring the potential benefits of an optimal (oracle-style) policy, using the Single policy against the best policy of *DyRAC* at each time, for all 3 providers. The cost reductions with *DyRAC* are up to 70% on average.

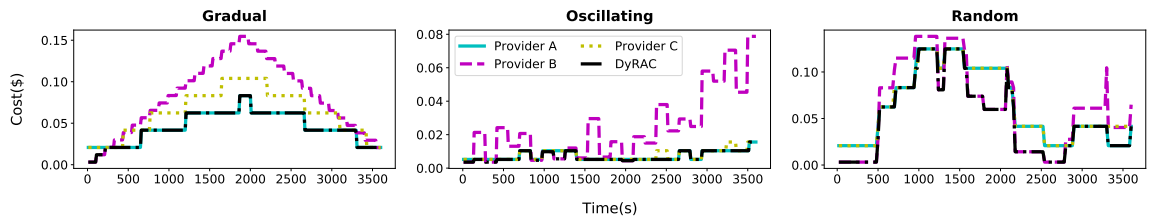


Figure 5.5: Comparing *DyRAC* against provider A, provider B, and provider C. The cost reductions with *DyRAC* are up to 25% on average.

5.3 Exploration of Service Deployment Alternatives

5.3.1 Comparison of *DyRAC* to Baseline Policies

We compare the resulting cost of deploying the same cloud services using *DyRAC* against the three baseline policies (Single, Adaptive, and Fixed) in Figure 5.3. In most cases, *DyRAC* results in the lowest deployment cost. *DyRAC* manages to keep unused resources to a minimum compared to other policies by choosing the VM types that best suit the workload. *DyRAC* reduces costs from 25% to 33% on average. The next best is the Fixed policy. This

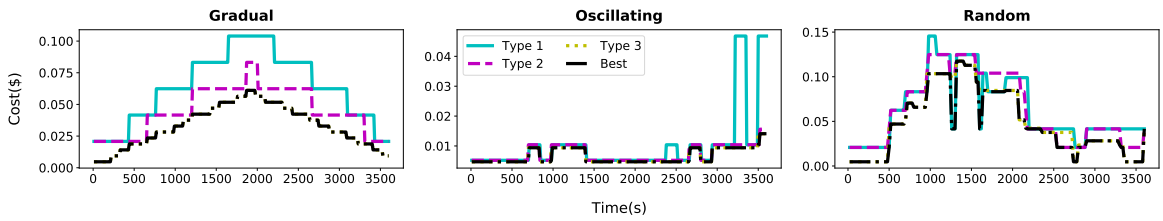


Figure 5.6: Comparing *DyRAC* using 3 typical VM types from provider B, which are present in the other providers as well. The cost reductions with *DyRAC* are up to 8% on average.

happens since provider B is more expensive than providers A and C, and we have selected a small-sized VM type. However, the limitation of Fixed is that if the service requires a lot of resources, it will cause network bandwidth issues as this policy will then spawn many VMs; this scenario is to be evaluated in future work. Finally, the Adaptive policy benefits from using all the allocated resources. However, it assumes that the VMs of a service are load-balanced, which may not be applicable in all cases.

Additionally, we explore the potential cost benefits of an optimal (oracle-style) RA. We compare this with the baseline Single VM policy with all providers and workloads. We define *Optimal* as the best choice among policies and providers for each timestamp of the experiment. Therefore, *Optimal* includes the optimizations in all three critical dimensions, i.e. choosing the configuration, VM type, and provider that minimizes cost. *Optimal* serves only as an upper bound of the feasible reduction of the deployment cost. We cannot realistically implement this policy, as it assumes that we know the entire timeline of workload changes beforehand. However, considering this comparison with *DyRAC*, together with the prior comparison against the three baseline policies, we establish a useful envelope of performance and dynamic behavior for the evaluation of *DyRAC* under real-world workload conditions. Figure 5.4 shows that *DyRAC* decreases the cost from 50% to 70% on average. In other words, we conclude that *DyRAC* is capable of serving the same dynamic workload at roughly half the cost as compared to other cost optimization approaches that are typical today.

5.3.2 Impact of changing the cloud provider

We now compare the cost difference of each provider using the same policy. This comparison provides insight into how relevant the "correct" choice of provider is for a service deployment. As a high-level conclusion, we observe that if the workload is bursty with random changes in its resource requirements, provider selection is significant to cost reduction. *DyRAC* manages to decrease costs up to 25% on average. Providers A and C have a similar model for charging; however, there are fewer options available in provider C. As shown in Figure 5.5, provider A is usually cheaper than provider C because, with provider A, we have more flexibility in choosing the resources of VMs. Therefore, we have more leeway to select a VM configuration closer to actual user requirements. provider B is more expensive than providers A and C. However, provider B allows us to choose an arbitrary amount of resources for each VM. Therefore, with provider B, we do not waste resources in the configuration of a service. It depends on the workload's resource requirements if an application hosted by provider B eventually achieves a cost reduction compared to other providers. If the workload requires only a few resources, then it is likely that we will waste resources with provider A and C (due to the fixed nature of the VM configurations available). On the other hand, if a workload requires many resources, we are more likely to find a VM configuration with few unused resources with providers A and C, and hence the deployment will be cheaper.

5.3.3 Configuring VM instances

Next, we compare the benefit of *DyRAC* drawn from choosing the most cost-efficient VM type to gain insight into the importance of this selection dimension. VM type selection adds a small benefit to reducing cost and is not as important as selecting the provider. Figure 5.6 shows the corresponding results. *DyRAC*, with all VM types available, decreases the cost up to 8% on average. This is because we include only three types of VMs available on all of the providers considered in this evaluation, and moreover, we consider VM type and provider selection independently. In future work, we plan to consider VM type and provider selection in a combined manner, which could further reduce costs.

Finally, we explore the difference in cost depending on how we divide service resources to VM instances. Figure 5.3 includes the Single, Fixed, and Adaptive policies that each sizes VMs differently. Single creates mostly large VMs. Fixed creates many small VMs, while Adaptive creates either large or small VMs based on workload demands. There is no guaranteed best policy for sizing, thus justifying an adaptive approach.

5.4 Summary

This chapter shows the potential cost benefits for cloud users by optimizing service deployment in the cloud. We approach the cost optimization of cloud services as a combination of resource assignment and provider selection. We argue that resource assignment is a significant aspect of the cost, despite being often neglected for simplicity. Therefore, we present and evaluate an approach (*DyRAC*) that adapts VM resources and their assignment during service execution. *DyRAC* adjusts the type, amount of assigned resources, and the provider according to the service workload changes. Our experiments illustrate that three concerns are important for reducing costs: (1) provider selection, (2) VM type selection, and (3) VM sizing. While an *Optimal* (oracle-style) policy can reduce the cost by up to 70%, *DyRAC* achieves a reduction of deployment costs from 25% to 33% on average.

Chapter 6

Related Work

We classify related work into the following categories: (a) offline resource estimation, (b) reactive resource adaptation, (c) minimizing latency in serverless frameworks, (d) efficient deployment in the cloud, and (e) workload traces and trace generators.

In the first category, the user predicts the resource requirements of an application before it is executed. The challenge they face is to choose accurate models of application behavior and resource usage. Additionally, it is challenging to estimate the resource requirements, as the execution environment changes which can lead to mispredicting the actual behavior of the application during execution.

In the second category, the user reactively adapts the assigned resource of an application while it is executing. Like offline estimation, it is challenging to choose accurate models of application behavior and resource usage. Unlike offline estimation, the behavior of the application is expected to change over time, so it is challenging to adapt the resources quickly when such changes happen.

The third category is similar to the second, with the difference that the user proactively adapts the assigned resource of an application while it is executing.

In the fourth category, the user selects the most appropriate cloud resources to use for an application and configuring them to optimize performance and minimize costs.

In the fifth category, the user faces two main challenges: 1) How to choose the parameters of the workload mix and 2) How to scale workloads to the available infrastructure in a reasonable manner. Although cloud benchmark suites help users by offering representative

cloud application, the user must still resolve challenging issues, such as how to determine a mix, how many instances of an application to spawn, how to generate a reasonable dataset for each application, how to deploy applications in the available infrastructure.

6.1 Offline resource estimation

Ernest [124] and CherryPick [21] are examples of this approach. Ernest finds the VM configuration with the least resources that achieve good performance. CherryPick implements a similar approach to Ernest but is customized for SQL queries. PARIS [130] has similar goals to Ernest; it finds the VM configuration that minimizes the deployment cost while preserving application performance. It improves over Ernest by minimizing the number of online samples required for identifying the VM configuration. The limitation of these approaches is that they cannot choose an arbitrary amount of resources for the running applications but rather allocate resources at the level of VMs, with static resource configurations. In contrast, this dissertation can choose the resources that match achieve the desired performance, resulting in fewer allocated resources. In principle, we can deploy many VMs with meager resources for each application to avoid the problem of having unused resources due to quantization. However, this results in more network traffic among VMs and more pressure on the hypervisor. In comparison, this dissertation is more straightforward to the user, spawns containers with many resources when possible, and puts less pressure on the infrastructure.

Paragon [43] identifies applications compatible with colocation. It performs small-scale interference tests between each application and controlled levels of background applications for each resource. Then, it uses complicated multi-variable statistical classifiers to predict the expected interference among applications at the time of colocation. Quasar [44] 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 with 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 re-

quirements. On the other hand this dissertation has a higher degree of freedom over the type of applications that colocates because of its isolation mechanism. Thus, in this dissertation (1) statistical modeling is simpler as it uses fewer variables and (2) mispredictions do not affect running applications.

Morpheus [72] operates on production environments that mostly execute similar tasks at the same time every day. This recurring pattern enables it to prepare offline resource allocations that are optimized for the jobs it expects. this dissertation 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 of the non-recurrent tasks.

Carbyne [59] 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 [95] 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 [6]. Instead, the isolation mechanism of this dissertation mitigates the impact of mispredictions and hence does not need VM migrations.

6.2 Reactive resource adaptation

dCat [129] and Ubik [75] are dynamic systems that manage the CPU LLC use of co-located workloads. These systems focus on managing the LLC cache only, limiting their applicability to memory-intensive applications. dCat collects the system metrics concerning the LLC to reserve a proper amount of LLC cache space for applications. Ubik dynamically partitions the LLC to improve the tail latency of latency-critical workloads. Both systems estimate the

required resources of applications with high accuracy, using relatively fast and elaborate techniques. This is possible because they manage only the LLC. Similar to dCat and Ubik, ResQ [120] manages the amount of LLC space required to avoid the "noisy" neighbor problem for software Network Functions (NF). In contrast, this dissertation implements a faster and simpler technique to manage multiple types of resources.

MemGuard [133] 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 this dissertation, instead it requires an external user-level daemon to configure it.

ARIA [125] and Jockey [53] implement a profiler-based resource allocation technique for MapReduce environments. Each job comes with a completion deadline which 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. this dissertation 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. [136] also propose profile-based scheduling for live-streaming video queries. They base their scheduling on the quality and lag goals of videos. They highlight the main challenge in video streaming which 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. this dissertation faces a similar challenge because multiple allocations can lead to the same performance. Haoyu Zhang et. al. finds a good allocation and subsequently search for mathematically equivalent ones, whereas this dissertation uses a heuristic to find equivalent allocations.

Autopilot [104] is the scheduler in Borg [126] that automatically allocates CPU and

memory for containers to avoid execution errors due to insufficient resources. this dissertation manages more resources in comparison and its goal is to guarantee performance. PRESS [58] and CloudScale [114] are online predictive systems that estimate the CPU resources of VM applications needed to achieve PLOs. Q-Clouds [93] addresses the performance interference of consolidated VMs by dynamically adjusting CPU and memory allocation. In comparison, this dissertation manages more resources and applies to both containers and VMs.

Bubble-Up [85] uses profiling via "bubble" to predict precisely performance degradation across workloads. Bubble-Flux [132] enhances this approach to online interference and QoS management and uses a dynamic bubble technique to probe servers for resource pressure, before taking colocation decisions. Bubble-Up and Bubble-Flux target latency-sensitive workloads that have a specific QoS target. this dissertation uses a more general the notion of QoS defined relative to standalone execution. Bubble-Up and Bubble-flux relies on detecting (offline or online) and avoiding interference via probes and appropriate colocation decisions, whereas this dissertation avoids interference by creating dedicated, properly-sized slices for each container, as estimated with the application profiling.

Many papers adopt a resource isolation approach to prevent or mitigate the adverse effects of interference. Heracles [81] and PARTIES [36] provide isolated hardware resources to workloads and monitor application performance at runtime. They isolate hardware concerning cores, memory, network, and L1 CPU cache. Additionally, they treat performance-critical workloads with high priority and offer them enough resources to meet their deadlines, while it allocates remaining server resources to batch workloads. They assume certain workload characteristics, typical for Google workloads, and shine at workloads that consist of both latency-critical and batch applications.

In contrast, this dissertation ensures a user-defined performance level for all running applications, rather than just the latency-critical ones. Additionally, while prior works such as Vanguard [111] partition the I/O path of data-intensive applications to provide predictable performance, they require the user to provide static information about the resource requirements of applications. In contrast, this dissertation estimates resources dynamically using a control loop. By doing so, this this dissertation provides more flexibility

and adaptability in managing resources for optimal performance.

6.3 Minimizing latency in serverless frameworks

Firecracker [18] is a hypervisor optimized for serverless functions that induce minimal memory overhead and can boot application code within a few hundred milliseconds. To achieve that, it implements a snapshot mechanism for spawning new MicroVM instances. vhive [121] further reduces the latency of new instances by exploiting the recurrence in the memory working set of serverless functions to reduce cold-start latency. gVisor [5] is an alternative approach to Firecracker for serverless functions. Instead of running containers on top of a virtual machine, it sandboxes containers to provide isolation. gVisor provides an additional layer that acts as a guest OS kernel and intercepts system calls for isolation and security. Thus gVisor achieves the spawning of new serverless functions within hundreds of milliseconds. However, this approach reduces application compatibility and causes high system call overhead. Catalyzer [49] improves upon gVisor to further reduce the start-up time of function instances to a few milliseconds in the best case. To achieve that, it introduces a new OS primitive, *sandboxed fork*, that reuses the state of already running sandboxes to spawn new function instances by restoring well-formed checkpoints. However, Catalyzer induces significant memory overhead to maintain the checkpoints for each function type in memory. A similar approach to Catalyzer is described in [115]. The key idea is the use of checkpoint/restore (CRIU) technique to replace the standard fork-exec procedure to spawn new function instances. Although the CRIU technique can significantly reduce the cold start of function instances, it still requires several tens of milliseconds.

Barista [27] is a system that manages the resources of pre-trained deep-learning prediction services. It assumes that the incoming load is variable and that latency constraints accompany services. To achieve these latency constraints, Barista: (1) forecasts the incoming requests based on historical data, (2) estimates offline the required resources by profiling the execution time on different resource configurations, and (3) chooses the cheapest VM configuration that achieves the required latency for provisioning. In case the forecasting overestimates the load, Barista assigns the unused resources to other low-

priority jobs. Similar to Barista, this dissertation predicts the resources required to achieve latency constraints; however, this dissertation predicts the resources online based on the actual incoming load. Furthermore, Barista collocates batch jobs to utilize idle cores in case of over-provisioning due to mispredictions, whereas this dissertation vertically scales the resources. Archipelago [116] is a serverless platform that expects a DAG of functions that are associated with a latency deadline. Archipelago partitions a given cluster into several smaller worker pools and assigns DAGs to each pool using a load-balancing approach. Additionally, Archipelago predicts the number of function invocations towards a worker pool, assuming a Poisson distribution. Like Archipelago, Fifer [63] is an adaptive resource management framework that manages function chains to reduce the number of containers spawned. Moreover, Fifer uses workload forecasting to spawn containers proactively. Compared to Archipelago and Fifer, this dissertation does not require information about function chains or DAGs to mitigate the cold-start latency. Additionally, similarly to Barista, Archipelago and Fifer might suffer from mispredictions in workload forecasting, which is not the case for this dissertation. An approach with a similar goal as Archipelago and Fifer is described in [68]. The authors propose a new primitive for serverless functions called freshen. It allows developers to perform actions within running functions for future use (e.g., spawn a new function instance to avoid the cold start of its invocations). However, the limitation is that it relies on the user for proactive actions. A further limitation is that users must change the executable within the serverless function to invoke this new primitive. Batch [20] is a framework for ML inference serving on serverless platforms. Batch minimizes the tail latency of inference requests by batching them into a single function invocation. To achieve that, it employs a profiler that predicts the characteristics of the workload. Compared to Batch, this dissertation does not require offline profiling and is not limited to ML-serving functions.

6.4 Efficient deployment in the cloud

Stratus [39] maintains a queue of running jobs/tasks and places tasks to increase infrastructure utilization as much as possible while at the same time achieving performance goals. It

packs tasks of a job together and places them in VMs to execute as many tasks as possible. Once the tasks are scheduled, they run to completion. Instead, this dissertation manages interactive cloud services, a significantly more complex and dynamic resource assignment problem.

Mao and Humphrey in [84] propose a system that decides at runtime how to minimize the resources of a running application, hence attempting to reduce cost as well. It explores how to schedule incoming requests to cloud services and auto-scale the resources of services to support the varying load with the minimum resources possible. This approach is dynamic and can adapt to workload changes; however, it is less cost-efficient compared to this dissertation since it targets to minimize the allocated resources and not directly the cost per se.

HotSpot [112] automatically decides when to migrate to new spot instances if their price changes favorably. However, this type of approach needs also to handle the risk of revocation and the overhead of migration, which currently makes it impractical.

Xinqian et. al. [139] explore the low energy efficiency of physical hosts for cloud providers. Their paper presents a VM placement approach that improves a cloud data center's energy efficiency, thus increasing the profit margins for cloud providers. Unlike this dissertation, this approach is directed to providers and not to users. In addition, this approach intervenes in the VM placement policy of a provider without considering the performance interference of collocated VMs. This can lead to significant performance degradation for applications.

6.5 Workload traces and trace generators

BigDataBench [54] is a benchmark suite that provides benchmarks focused on online service, offline analytics, graph analytics, AI, data warehouse, NoSQL, and streaming. Contrary to our work, which focuses on using kernels as computation units, BigDataBench analyses data motifs, which are commonly occurring data characteristics, and correlates them with a set of micro-benchmarks to form computational building blocks. Subsequently, common sequences of data motifs are identified and used to compose more complex

workload building blocks, based on these data motif sequences and the previously constructed micro-benchmarks. Finally, the resulting workload building blocks contrast with user characteristics and real-life data center application characteristics to come up with a workload based on a real-life software stack. BigDataBench aims to provide variety and scalability in terms of benchmark generation, and it provides a dynamically created collection of computation units. Our work intends to add the extra step of generating a synthetic workload based on the kernels, which are, in our case, the units of computation.

In DCMIX [128], the authors collect a set of characteristic computation units, both concerning offline analytics and online services, to come up with a collection of benchmarks. Although DCMIX is a benchmark suite, their work partially parallels our own in that they provide the functionality to execute a workload, as specified in a workload configuration file, which comprises characteristics such as the number of requests, server threads, job threads, and others. Our work also provides the capability to create arbitrarily synthetic workloads and goes one step further: we profile real-life data center traces to offer the user a collection of parameter values based on realistic scenarios.

HiBench [67] is a benchmark suite for Hadoop. It consists of a collection of programs that can be executed over the Hadoop framework, aiming to evaluate deployment options.

Several synthetic workload generators serve to provide benchmarks that target specific operation aspects of a data center. Such a generator is BigBench [37, 55]. BigBench is a work that, similarly to ours, deals with the extraction of characteristics of realistic workloads and their use to generate synthetic workloads. However, BigBench focuses on a particular use case, a system handling the transactions of a big retailer, both physical and online. As such, the result is rather restricted to workloads that consist of database queries, contrary to our work, which aims at producing synthetic workloads for a more generic application domain.

Some more generic efforts to produce synthetic workloads with realistic characteristics appear elsewhere in the literature. Due to the wide-spread use of this particular computing paradigm, two prominent synthetic workload generation examples, [23] and [4], deal with the synthesis of MapReduce workloads.

GridMix [23] is a benchmarking tool by Apache Hadoop, which creates synthetic workloads suitable for testing Hadoop clusters. To do so, it relies on MapReduce job traces

extracted from some production clusters. Rumen, a tool in the Hadoop suite, generates such traces from the job history files of the cluster. These traces are used as input to GridMix, which in turn outputs a synthetic workload. The load can be scaled up or down by adjusting the intervals between job submissions. Several inherent and run-time characteristics of the jobs, such as memory usage, input data size, or job type, can be modeled.

Similarly, SWIM [4] generates synthetic workloads based on the traces released by Facebook in 2009 and 2010. SWIM aims to produce workloads of shorter duration than the original traces while still maintaining their essential characteristics. As in our case, SWIM workloads are suitable for execution on real-life cluster setups. Contrary to this dissertation, this workload consists exclusively of MapReduce-related tasks. We aim to enable having a broader range of job types in synthetic workloads.

GloudSim [46] is a simulation toolkit that aims at reproducing the behavior of a real cloud system, either on a desktop or on a cluster that uses virtual machines. This toolkit includes a trace analyzer that characterizes traces and a sample job generator that generates synthetic workloads. Contrary to our work, which provides a framework for describing workloads using any public trace, GloudSim aims at closely reproducing the characteristics specific to the Google trace. Given that the sample job generator follows the job arrival times of the Google trace, it is not clear how to scale up or down a synthetic workload. Furthermore, it is not clear how to vary parameters such as the load to the system under test or the type of applications. Overall, workload generation with GloudSim is meant for simulation rather than the actual execution.

In [66], CloudMix, a benchmarking tool that synthesizes cloud workloads with realistic characteristics, is presented. Similar to our use of a repository of kernels, CloudMix relies on a repository of *reducible workload blocks* (RWBs). RWBs represent different mixes of assembly-level computations. The collection of RWBs is used to create synthetic workloads that can be executed on real clusters. They are designed to mimic micro-architectural usage characteristics of tasks found in the Google cluster trace. Workload scaling down is possible by reducing job and trace durations. Unlike our work, [66] is more concerned with reproducing the run-time behavior (in terms of computational characteristics and memory accesses) observed in the Google trace. We are rather concerned with realistically

synthesizing various sequences of the influx of jobs, independently of the substrate they will be executed on.

Chapter 7

Conclusions and Future Work

7.1 Future Work

This dissertation provides solutions to increasing the utilization of data center infrastructure which is surprisingly low in the average case. Next, we describe some directions for future work.

Offline profiling limitations: Our current profiling technique is generic and can accurately profile most cloud applications. However, it does not capture well applications with peculiar performance behaviour. For example, consider video encoding can have bitrates of 1080P, 720P, and 480P. The function that correlates performance to offered resources is essentially a step function. The performance has three values low, mid, and best quality, while the offered resource affects the performance only when it crosses certain thresholds. Logistic regression will not generate accurate performance profiles for such cases. Therefore, an interesting direction for future work would be to explore other machine-learning techniques that can handle special cases for applications. We would first identify such applications and then choose the most appropriate technique to profile their performance.

Managing hardware heterogeneity: In our evaluation, we use machines of the same generation with similar hardware. However, typically, data centers include three different server generations [43] and a variety of different peripherals. We do not consider this orthogonal dimension in this dissertation. Other works consider heterogeneity [44] as

part of their performance modeling or in their online estimations. Although making a thorough investigation of the possibilities concerning the hardware combinations, it would be interesting to first explore how the accuracy of the performance profiles is affected on different hardware. Similarly, we can also compare how the accuracy is affected when we use different CPU vendors such as ARM, AMD, or Intel. Future work should address this issue to drive the system closer to the real world.

Support of multiple metrics in the reactive approach: In our reactive approach, we allow users to define a single performance target for each application. For instance, we cannot choose to manage the resources of a web server with a throughput of one thousand transactions per second and tail latency under ten milliseconds. Currently, we can choose to either choose to target tail latency or throughput, but, in some cases, a single performance goal is not sufficient for users. It is a very interesting direction for future work to explore the challenges to support multiple performance metrics for a single at the same time.

7.2 Conclusions

Cloud computing has revolutionized how businesses operate by removing the need for infrastructure management and offering seamless scalability of resources. However, the ever-increasing demand for cloud resources puts pressure on providers to continuously increase the offered resources. This pressure is driving providers to explore alternatives to meet the demand, such as increasing the utilization of the existing infrastructure. The proposed intelligent resource management solution offers a practical solution for providers to optimize resource utilization and address the challenges posed by the increasing demand for cloud resources.

The dissertation proposes techniques to use data center resources more efficiently focusing, on two main directions. First, it proposes a performance profiling approach based on logistic regression that utilizes multiple application runs to generate accurate performance models. The performance models are then used to correlate the resources offered to the application with the expected performance, enabling more efficient resource

allocation and assignment. Although offline profiling is highly accurate, it cannot efficiently handle dynamic workloads, leading to the proposal of a reactive approach as the second direction. The reactive approach generates performance models for applications on-the-fly, allowing for dynamic workload adaptation, albeit with less accuracy. We base our reactive approach on a feedback loop control, specifically the Proportional Integral Derivative controller (PID).

Additionally, we show how to apply our techniques to serverless computing and cloud deployments. First, we develop a controller based on the reactive approach that adapts resources of serverless functions to ensure low tail latency for execution. Second, we show how to minimize the cost of deploying cloud services. We define a function that correlates the cost of deployment to the assigned resources and the provider selected. This function adapts the resources of cloud services and providers according to the workload.

Overall, we can manage the resources for various workloads and efficiently use cloud resources for different hardware and applications. Our results show that with machine learning, we can achieve high resource utilization and good quality of service for the applications.

Bibliography

[1] ab - Apache HTTP server benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.ht

[2] An open-source monitoring solution. https://prometheus.io/.

[3] Apache CouchDB. https://couchdb.apache.org/.

[4] Facebook MapReduce traces.

[5] gVisor is an application kernel for containers that provides efficient defense-in-depth anywhere. https://gvisor.dev.

[6] Launching a spark/shark cluster on ec2. http://ampcamp.berkeley.edu/exercises-strata-conf-2013/launching-a-cluster.html.

[7] Linux containers (LXC). https://linuxcontainers.org/lxc/introduction/.

[8] PID tutorial. https://oscarliang.com/quadcopter-pid-explained-tuning/.

[9] The Apache HTTP Server. http://httpd.apache.org.

[10] The Elasticsearch Analytics Engine. https://www.elastic.co/products/elasticsearch.

[11] The Memcached the Distributed Memory Object Caching System. https://memcached.org.

[12] The NGINX web server. http://nginx.org.

[13] The Redis in-memory Data Structure Store. https://redis.io.

[14] Yahoo! Cloud Serving Benchmark (YCSB). https://github.com/brianfrankcooper/YCSB/wiki.

[15] Skynet adaptive k8s pod scheduler: code repository, 2020. Reference to repository redacted to preserve double-blind submission and review rules.

[16] Omar Arif Abdul-Rahman and Kento Aida. Towards understanding the usage behavior of google cloud users: the mice and elephants phenomenon. In *Cloud Computing Technology and Science (CloudCom), 2014 IEEE 6th International Conference on*, pages 272–277. IEEE, 2014.

[17] Omar Arif Abdul-Rahman and Kento Aida. Towards understanding the usage behavior of Google cloud users: the mice and elephants phenomenon. In *IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 272–277, Singapore, December 2014.

[18] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX symposium on networked systems design and implementation (NSDI 20)*, pages 419–434, 2020.

[19] Yalçın Akçay, Haijun Li, and Susan H Xu. Greedy algorithm for the general multidimensional knapsack problem. *Annals of Operations Research*, 150(1):17, 2007.

[20] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. Batch: machine learning inference serving on serverless platforms with adaptive batching. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE, 2020.

[21] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *NSDI*, volume 2, pages 4–2, 2017.

[22] Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Myers, and David J. Lipman. Basic Local Alignment Search Tool. *Journal of Molecular Biology*, 215:403–410, 1990.

[23] Apache Hadoop. *GridMix*.

[24] Jens Axboe. Flexible I/O Tester. https://github.com/axboe.

[25] Anton Beloglazov, Rajkumar Buyya, Young Choon Lee, and Albert Zomaya. A taxonomy and survey of energy-efficient data centers and cloud computing systems. *Advances in computers*, 82:47–111, 2011.

[26] Jacob Benesty, Jingdong Chen, Yiteng Huang, and Israel Cohen. Pearson correlation coefficient. In *Noise reduction in speech processing*, pages 1–4. Springer, 2009.

[27] Anirban Bhattacharjee, Ajay Dev Chhokra, Zhuangwei Kang, Hongyang Sun, Aniruddha Gokhale, and Gabor Karsai. Barista: Efficient and scalable serverless serving system for deep learning prediction services. In *2019 IEEE International Conference on Cloud Engineering (IC2E)*, pages 23–33. IEEE, 2019.

[28] Robert Birke, Lydia Y Chen, and Evgenia Smirni. Data centers in the wild: A large performance study.

[29] Leon Bottou and Olivier Bousquet. The tradeoffs of large scale learning. In *Proceedings of the 20th International Conference on Neural Information Processing Systems*, NIPS'07, pages 161–168, USA, 2007. Curran Associates Inc.

[30] Soufiene Bouallègue, Joseph Haggège, Mounir Ayadi, and Mohamed Benrejeb. Pid-type fuzzy logic controller tuning based on particle swarm optimization. *Engineering Applications of Artificial Intelligence*, 25(3):484–493, 2012.

[31] Hamid Boubertakh, Mohamed Tadjine, Pierre-Yves Glorennec, and Salim Labiod. Tuning fuzzy pd and pi controllers using reinforcement learning. *ISA transactions*, 49(4):543–551, 2010.

[32] Nicolò Maria Calcavecchia, Ofer Biran, Erez Hadad, and Yosef Moatti. Vm placement strategies for cloud scenarios. In *2012 IEEE Fifth International Conference on Cloud Computing*, pages 852–859. IEEE, 2012.

[33] Jeffrey S Chase, Darrell C Anderson, Prachi N Thakar, Amin M Vahdat, and Ronald P Doyle. Managing energy and server resources in hosting centers. *ACM SIGOPS operating systems review*, 35(5):103–116, 2001.

[34] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pages 44–54, Washington, DC, USA, 2009. IEEE Computer Society.

[35] Gong Chen, Wenbo He, Jie Liu, Suman Nath, Leonidas Rigas, Lin Xiao, and Feng Zhao. Energy-Aware Server Provisioning and Load Dispatching for Connection-Intensive Internet Services. In *NSDI*, volume 8, pages 337–350, 2008.

[36] Shuang Chen, Christina Delimitrou, and José F Martínez. Parties: Qos-aware resource partitioning for multiple interactive services. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 107–120. ACM, 2019.

[37] Yanpei Chen, Sara Alspaugh, and Randy Katz. Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads. *Proceedings of the VLDB Endowment*, 5(12):1802–1813, 2012.

[38] Yanpei Chen, Archana Sulochana Ganapathi, Rean Griffith, and Randy H. Katz. Analysis and lessons from a publicly available google cluster trace. Technical Report UCB/EECS-2010-95, EECS Department, University of California, Berkeley, Jun 2010.

[39] Andrew Chung, Jun Woo Park, and Gregory R Ganger. Stratus: Cost-aware container scheduling in the public cloud. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 121–134, 2018.

[40] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 153–167. ACM, 2017.

[41] Pamela Delgado, Diego Didona, Florin Dinu, and Willy Zwaenepoel. Job-aware scheduling in eagle: Divide and stick to your probes. In *Proceedings of the Seventh*

ACM Symposium on Cloud Computing, SoCC '16, pages 497–509, New York, NY, USA, 2016. ACM.

[42] Pamela Delgado, Florin Dinu, Anne-Marie Kermarrec, and Willy Zwaenepoel. Hawk: Hybrid datacenter scheduling. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, pages 499–510, Berkeley, CA, USA, 2015. USENIX Association.

[43] Christina Delimitrou and Christos Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'13, pages 77–88, New York, NY, USA, 2013. ACM.

[44] Christina Delimitrou and Christos Kozyrakis. 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, pages 127–144, New York, NY, USA, 2014. ACM.

[45] Chanchal Dey and Rajani K Mudi. An improved auto-tuning scheme for pid controllers. *ISA transactions*, 48(4):396–409, 2009.

[46] Sheng Di and Franck Cappello. Gloudsim: Google trace based cloud simulator with virtual machines. *Softw., Pract. Exper.*, 45(11):1571–1590, 2015.

[47] Sheng Di, Derrick Kondo, and F. Cappello. Characterizing and modeling cloud applications/jobs on a google data center. *Journal of Supercomputing*, 69:139–160, 01/2014 2014.

[48] Sheng Di, Derrick Kondo, and Walfredo Cirne. Characterization and comparison of cloud versus grid workloads. In *IEEE Cluster 2012*, 2012.

[49] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the Twenty-Fifth International*

Conference on Architectural Support for Programming Languages and Operating Systems, pages 467–481, 2020.

[50] Bradley Efron, Robert Tibshirani, et al. Using specially designed exponential families for density estimation. *The Annals of Statistics*, 24(6):2431–2461, 1996.

[51] Liu Fan and Er Meng Joo. Design for auto-tuning pid controller based on genetic algorithms. In *2009 4th IEEE Conference on Industrial Electronics and Applications*, pages 1924–1928. IEEE, 2009.

[52] Xiaobo Fan, Wolf-Dietrich Weber, and Luiz André Barroso. Power Provisioning for a Warehouse-sized Computer. In *ISCA*, 2007.

[53] Andrew D. Ferguson, Peter Bodik, Srikanth Kandula, Eric Boutin, and Rodrigo Fonseca. Jockey: Guaranteed job latency in data parallel clusters. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 99–112, New York, NY, USA, 2012. ACM.

[54] Wanling Gao, Jianfeng Zhan, Lei Wang, Chunjie Luo, Daoyi Zheng, Xu Wen, Rui Ren, Chen Zheng, Xiwen He, Hainan Ye, et al. Bigdatabench: A scalable and unified big data and ai benchmark suite. *arXiv preprint arXiv:1802.08254*, 2018.

[55] Ahmad Ghazal, Tilmann Rabl, Minqing Hu, Francois Raab, Meikel Poess, Alain Crolotte, and Hans-Arno Jacobsen. Bigbench: towards an industry standard benchmark for big data analytics. In *Proceedings of the 2013 ACM SIGMOD international conference on Management of data*, pages 1197–1208, 2013.

[56] Daniel Gmach, Jerry Rolia, Ludmila Cherkasova, and Alfons Kemper. Workload analysis and demand prediction of enterprise data center applications. In *Proceedings of the 2007 IEEE 10th International Symposium on Workload Characterization*, IISWC'07, pages 171–180, Washington, DC, USA, 2007. IEEE Computer Society.

[57] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert NM Watson, and Steven Hand. Firmament: fast, centralized cluster scheduling at scale. In *Proceedings of OSDI'16:*

12th USENIX Symposium on Operating Systems Design and Implementation, page 99, 2016.

[58] Zhenhuan Gong, Xiaohui Gu, and J. Wilkes. Press: Predictive elastic resource scaling for cloud systems. In *2010 International Conference on Network and Service Management*, pages 9–16, Oct 2010.

[59] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. Altruistic scheduling in multi-resource clusters. In *Proceedings of OSDI'16: 12th USENIX Symposium on Operating Systems Design and Implementation*, page 65, 2016.

[60] Alexander G Gray and Andrew W Moore. Nonparametric density estimation: Toward computational tractability. In *Proceedings of the 2003 SIAM International Conference on Data Mining*, pages 203–211. SIAM, 2003.

[61] Arsalane Chouaib Guidoum. Kernel estimator and bandwidth selection for density and its derivatives. *The kedd package, version*, 1, 2015.

[62] Ajay Gulati, Ganesha Shanmuganathan, Irfan Ahmad, Carl Waldspurger, and Mustafa Uysal. Pesto: online storage performance management in virtualized datacenters. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 19. ACM, 2011.

[63] Jashwant Raj Gunasekaran, Prashanth Thinakaran, Nachiappan C Nachiappan, Mahmut Taylan Kandemir, and Chita R Das. Fifer: Tackling resource underutilization in the serverless era. In *Proceedings of the 21st International Middleware Conference*, pages 280–295, 2020.

[64] Tian Guo, Upendra Sharma, Prashant Shenoy, Timothy Wood, and Sambit Sahu. Cost-aware cloud bursting for enterprise applications. *ACM Transactions on Internet Technology (TOIT)*, 13(3):1–24, 2014.

[65] James Hamilton. Overall data center costs. http://perspectives.mvdirona.com/2010/09/overall-data-center-costs, 2010.

[66] R. Han, Z. Zong, F. Zhang, J. L. Vazquez-Poletti, Z. Jia, and L. Wang. Cloudmix: Generating diverse and reducible workloads for cloud systems. In *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*, pages 496–503, June 2017.

[67] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010)*, pages 41–51. IEEE, 2010.

[68] Erika Hunhoff, Shazal Irshad, Vijay Thurimella, Ali Tariq, and Eric Rozner. Proactive serverless function resource management. In *Proceedings of the 2020 Sixth International Workshop on Serverless Computing*, pages 61–66, 2020.

[69] Alan Julian Izenman. Review papers: Recent developments in nonparametric density estimation. *Journal of the American Statistical Association*, 86(413):205–224, 1991.

[70] Seyyed Ahmad Javadi, Amoghavarsha Suresh, Muhammad Wajahat, and Anshul Gandhi. Scavenger: A black-box batch workload resource manager for improving utilization in cloud environments. 2019.

[71] Vimalkumar Jeyakumar, Mohammad Alizadeh, David Mazières, Balaji Prabhakar, Changhoon Kim, and Albert Greenberg. Eyeq: Practical network performance isolation at the edge. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 297–312, Berkeley, CA, USA, 2013. USENIX Association.

[72] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Íñigo Goiri, Subru Krishnan, Janardhan Kulkarni, and Sriram Rao. Morpheus: towards automated slos for enterprise clusters. In *Proceedings of OSDI'16: 12th USENIX Symposium on Operating Systems Design and Implementation*, page 117, 2016.

[73] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Íñigo Goiri, Subru Krish-

nan, Janardhan Kulkarni, et al. Morpheus: Towards automated slos for enterprise clusters. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 117–134, 2016.

[74] Sasan Karamizadeh, Shahidan M Abdullah, Azizah A Manaf, Mazdak Zamani, and Alireza Hooman. An overview of principal component analysis. *Journal of Signal and Information Processing*, 4(3B):173, 2013.

[75] Harshad Kasture and Daniel Sanchez. 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, pages 729–742, New York, NY, USA, 2014. ACM.

[76] Hans Kellerer, Ulrich Pferschy, and David Pisinger. Multidimensional knapsack problems. In *Knapsack problems*, pages 235–283. Springer, 2004.

[77] Jin-Sung Kim, Jin-Hwan Kim, Ji-Mo Park, Sung-Man Park, Won-Yong Choe, and Hoon Heo. Auto tuning pid controller based on improved genetic algorithm for reverse osmosis plant. *World Academy of Science, Engineering and Technology*, 47(2):384–389, 2008.

[78] Y. Klonatos, T. Makatos, M. Marazakis, M.D. Flouris, and A. Bilas. Azor: Using two-level block selection to improve SSD-based I/O caches. In *Proc. IEEE International Conference on Networking, Architecture, and Storage (NAS)*, 2011.

[79] Bingwei Liu, Yinan Lin, and Yu Chen. Quantitative workload analysis and prediction using google cluster traces. *2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 935–940, 2016.

[80] Zitao Liu and Sangyeun Cho. Characterizing machines and workloads on a google cluster. In *Proceedings of the 2012 41st International Conference on Parallel Processing Workshops*, ICPPW '12, pages 397–403, Washington, DC, USA, 2012. IEEE Computer Society.

[81] 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, pages 450–462, New York, NY, USA, 2015. ACM.

[82] C. Lu, K. Ye, G. Xu, C. Z. Xu, and T. Bai. Imbalance in the cloud: An analysis on alibaba cluster trace. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 2884–2892, Dec 2017.

[83] Chengzhi Lu, Kejiang Ye, Guoyao Xu, Cheng-Zhong Xu, and Tongxin Bai. Imbalance in the cloud: An analysis on alibaba cluster trace. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 2884–2892. IEEE, 2017.

[84] Ming Mao and Marty Humphrey. Auto-scaling to minimize cost and meet application deadlines in cloud workflows. In *SC'11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2011.

[85] 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 Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pages 248–259, New York, NY, USA, 2011. ACM.

[86] Silvano Martello and Paolo Toth. Bin-packing problem. *Knapsack problems: Algorithms and computer implementations*, pages 221–245, 1990.

[87] Eric Masanet, Arman Shehabi, Nuoa Lei, Sarah Smith, and Jonathan Koomey. Recalibrating global data center energy-use estimates. *Science*, 367(6481):984–986, 2020.

[88] S. Mavridis, Y. Sfakianakis, A Papagiannis, M. Marazakis, and A Bilas. Jericho: Achieving scalability through optimal data placement on multicore systems. In *30th Symposium on Mass Storage Systems and Technologies (MSST)*, 2014.

[89] S. Mavridis, Y. Sfakianakis, A. Papagiannis, M. Marazakis, and A. Bilas. Jericho: Achieving scalability through optimal data placement on multicore systems. In *2014 30th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, June 2014.

[90] Asit K Mishra, Joseph L Hellerstein, Walfredo Cirne, and Chita R Das. Towards characterizing cloud backend workloads: insights from google compute clusters. *ACM SIGMETRICS Performance Evaluation Review*, 37(4):34–41, 2010.

[91] Ismael Solís Moreno, Peter Garraghan, Paul Townend, and Jie Xu. An approach for characterizing workloads in google cloud to derive realistic resource utilization models. In *SOSE*, pages 49–60. IEEE Computer Society, 2013.

[92] Raghunath Nambiar, Nicholas Wakou, Forrest Carman, and Michael Majdalany. Transaction processing performance council (tpc): State of the council 2010, 2011.

[93] Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. Q-clouds: Managing performance interference effects for qos-aware clouds. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys'10, pages 237–250, New York, NY, USA, 2010. ACM.

[94] Clément Nedelcu. *Nginx http server*. Packt Publishing, 2013.

[95] Dejan Novaković, Nedeljko Vasić, Stanko Novaković, Dejan Kostić, and Ricardo Bianchini. Deepdive: Transparently identifying and managing performance interference in virtualized environments. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 219–230, Berkeley, CA, USA, 2013. USENIX Association.

[96] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 69–84. ACM, 2013.

[97] 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, AM Novak, F Nothaft, L Pachter, D Patterson, M Smuga-Otto, JM Stuard, L Van't Veer, B Wold, and D Haussler. The NIH BD2K center for big data in translational genomics. *Journal of the American Medical Informatics Association*, 22(6):1143–1147, November 2015.

[98] Paul Menage. cgroups features, including cpusets and memory controller. In *Linux Kernel cgroups Documentation*. The Linux Kernel Archives, 2006.

[99] Jeff Rasley, Konstantinos Karanasos, Srikanth Kandula, Rodrigo Fonseca, Milan Vojnovic, and Sriram Rao. Efficient queue management for cluster scheduling. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 36. ACM, 2016.

[100] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 7:1–7:13, New York, NY, USA, 2012. ACM.

[101] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. Towards understanding heterogeneous clouds at scale: Google trace analysis. 2012.

[102] Charles Reiss, Alexey Tumanov, Gregory R Ganger, Randy H Katz, and Michael A Kozuch. Towards understanding heterogeneous clouds at scale: Google trace analysis. *Intel Science and Technology Center for Cloud Computing, Tech. Rep*, 84, 2012.

[103] Daniel E Rivera, Manfred Morari, and Sigurd Skogestad. Internal model control: Pid controller design. *Industrial & engineering chemistry process design and development*, 25(1):252–265, 1986.

[104] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, et al. Autopilot: workload autoscaling at google. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[105] Johann Schleier-Smith, Vikram Sreekanti, Anurag Khandelwal, Joao Carreira, Neeraja J Yadwadkar, Raluca Ada Popa, Joseph E Gonzalez, Ion Stoica, and David A Patterson. What serverless computing is and should become: The next phase of cloud computing. *Communications of the ACM*, 64(5):76–84, 2021.

[106] Yannis Sfakianakis, Eleni Kanellou, Manolis Marazakis, and Angelos Bilas. Trace-based workload generation and execution. In *Euro-Par 2021: Parallel Processing: 27th International Conference on Parallel and Distributed Computing, Lisbon, Portugal, September 1–3, 2021, Proceedings 27*, pages 37–54. Springer, 2021.

[107] Yannis Sfakianakis, Christos Kozanitis, Christos Kozyrakis, and Angelos Bilas. Quman: Profile-based improvement of cluster utilization. *ACM Transactions on Architecture and Code Optimization (TACO)*, 15(3):27, 2018.

[108] Yannis Sfakianakis, Manolis Marazakis, and Angelos Bilas. Dyrac: Cost-aware resource assignment and provider selection for dynamic cloud workloads. In *2020 IEEE 26th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 502–509. IEEE, 2020.

[109] Yannis Sfakianakis, Manolis Marazakis, and Angelos Bilas. Skynet: Performance-driven resource management for dynamic workloads. In *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*. IEEE, 2021.

[110] Yannis Sfakianakis, Manolis Marazakis, Christos Kozanitis, and Angelos Bilas. Latest: Vertical elasticity for millisecond serverless execution. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 879–885. IEEE, 2022.

[111] Yannis Sfakianakis, Stelios Mavridis, Anastasios Papagiannis, Spyridon Papageorgiou, Markos Fountoulakis, Manolis Marazakis, and Angelos Bilas. Vanguard: Increasing server efficiency via workload isolation in the storage i/o path. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC'14, pages 19:1–19:13, New York, NY, USA, 2014. ACM.

[112] Supreeth Shastri and David Irwin. Hotspot: automated server hopping in cloud spot markets. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 493–505, 2017.

[113] Arman Shehabi, Sarah Smith, Dale Sartor, Richard Brown, Magnus Herrlin, Jonathan Koomey, Eric Masanet, Nathaniel Horner, Inês Azevedo, and William Lintner. United states data center energy usage report. 2016.

[114] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. Cloudscale: Elastic resource scaling for multi-tenant cloud systems. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, SOCC'11, pages 5:1–5:14, New York, NY, USA, 2011. ACM.

[115] Paulo Silva, Daniel Fireman, and Thiago Emmanuel Pereira. Prebaking functions to warm the serverless cold start. In *Proceedings of the 21st International Middleware Conference*, pages 1–13, 2020.

[116] Arjun Singhvi, Kevin Houck, Arjun Balasubramanian, Mohammed Danish Shaikh, Shivaram Venkataraman, and Aditya Akella. Archipelago: A scalable low-latency serverless platform. *arXiv preprint arXiv:1911.09849*, 2019.

[117] Ibrahim Takouna, Wesam Dawoud, and Christoph Meinel. Accurate mutlicore processor power models for power-aware resource management. In *2011 IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing*, pages 419–426. IEEE, 2011.

[118] Ali Tariq, Austin Pahl, Sharat Nimmagadda, Eric Rozner, and Siddharth Lanka. Sequoia: Enabling quality-of-service in serverless computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 311–327, 2020.

[119] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. Borg: the next generation. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–14, 2020.

[120] Amin Tootoonchian, Aurojit Panda, Chang Lan, Melvin Walls, Katerina Argyraki, Sylvia Ratnasamy, and Scott Shenker. Resq: Enabling slos in network function virtualization. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, pages 283–297, 2018.

[121] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. Benchmarking, analysis, and optimization of serverless function snapshots. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 559–572, 2021.

[122] Mahesh K Varanasi and Behnaam Aazhang. Parametric generalized gaussian density estimation. *The Journal of the Acoustical Society of America*, 86(4):1404–1415, 1989.

[123] Amir Varasteh and Maziar Goudarzi. Server consolidation techniques in virtualized data centers: A survey. *IEEE Systems Journal*, 11(2):772–783, 2015.

[124] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. Ernest: Efficient performance prediction for large-scale advanced analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 363–378, Santa Clara, CA, March 2016. USENIX Association.

[125] Abhishek Verma, Ludmila Cherkasova, and Roy H. Campbell. Aria: Automatic resource inference and allocation for mapreduce environments. In *Proceedings of the 8th ACM International Conference on Autonomic Computing*, ICAC '11, pages 235–244, New York, NY, USA, 2011. ACM.

[126] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015.

[127] Zhihao Wang, Junfang Wang, Bowen Li, Yijing Liu, and Jinlong Ma. Online cloud provider selection for qos-sensitive users: Learning with competition. *IAENG International Journal of Computer Science*, 43(3):310–317, 2016.

[128] Xingwang Xiong, Lei Wang, Wanling Gao, Rui Ren, Ke Liu, Chen Zheng, Yu Wen, and Yi Liang. Dcmix: generating mixed workloads for the cloud data center. In *International Symposium on Benchmarking, Measuring and Optimization*, pages 105–117. Springer, 2018.

[129] Cong Xu, Karthick Rajamani, Alexandre Ferreira, Wesley Felter, Juan Rubio, and Yang Li. dcat: Dynamic cache management for efficient, performance-sensitive infrastructure-as-a-service. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 14:1–14:13, New York, NY, USA, 2018. ACM.

[130] Neeraja J Yadwadkar, Bharath Hariharan, Joseph E Gonzalez, Burton Smith, and Randy H Katz. Selecting the best vm across multiple public clouds: A data-driven performance modeling approach. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 452–465. ACM, 2017.

[131] 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, pages 607–618, New York, NY, USA, 2013. ACM.

[132] 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, pages 607–618, New York, NY, USA, 2013. ACM.

[133] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*, pages 55–64, April 2013.

[134] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Pre-*

sented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12), pages 15–28, 2012.

[135] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache spark: A unified engine for big data processing. *Commun. ACM*, 59(11):56–65, October 2016.

[136] Haoyu Zhang, Ganesh Ananthanarayanan, Peter Bodik, Matthai Philipose, Paramvir Bahl, and Michael J. Freedman. Live video analytics at scale with approximation and delay-tolerance. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 377–392, Boston, MA, 2017. USENIX Association.

[137] Miranda Zhang, Rajiv Ranjan, Armin Haller, Dimitrios Georgakopoulos, and Peter Strazdins. Investigating decision support techniques for automating cloud service selection. In *4th IEEE International Conference on Cloud Computing Technology and Science Proceedings*, pages 759–764. IEEE, 2012.

[138] Miranda Zhang, Rajiv Ranjan, Surya Nepal, Michael Menzel, and Armin Haller. A declarative recommender system for cloud infrastructure services selection. In *International Conference on Grid Economics and Business Models*, pages 102–113. Springer, 2012.

[139] Xinqian Zhang, Tingming Wu, Mingsong Chen, Tongquan Wei, Junlong Zhou, Shiyan Hu, and Rajkumar Buyya. Energy-aware virtual machine allocation for cloud with resource reservation. *Journal of Systems and Software*, 147:147–161, 2019.

[140] Yunqi Zhang, Michael A Laurenzano, Jason Mars, and Lingjia Tang. 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*, pages 406–418. IEEE Computer Society, 2014.

[141] Z. Zhang, K. Barbary, F. A. Nothaft, E. Sparks, O. Zahn, M. J. Franklin, D. A. Patterson, and S. Perlmutter. Scientific computing meets big data technology: An astronomy use

case. In *Big Data (Big Data), 2015 IEEE International Conference on*, pages 918–927, Oct 2015.

[142] Da Zheng, Randal Burns, and Alexander S Szalay. Toward millions of file system iops on low-cost, commodity hardware. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 69–71. ACM, 2013.

[143] Haishan Zhu and Mattan Erez. Dirigent: Enforcing qos for latency-critical tasks on shared multicore systems. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 33–47, 2016.

[144] John G Ziegler and Nathaniel B Nichols. Optimum settings for automatic controllers. *trans. ASME*, 64(11), 1942.