ALGORITHMS AND TECHNIQUES FOR DYNAMIC RESOURCE MANAGEMENT ACROSS CLOUD-EDGE RESOURCE SPECTRUM

By

Shashank Shekhar

Dissertation

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

June 30, 2018

Nashville, Tennessee

To my beloved wife, Kasturi for her patience, sacrifices and support

ACKNOWLEDGMENTS

This dissertation would not have been possible without the constant guidance, encouragement and support I received from my advisor, dissertation committee members, colleagues, family, and friends.

First and foremost, I would like to express my sincere gratitude to my advisor Dr. Aniruddha S. Gokhale, for his supervision, advice, guidance, and unwavering support over the years. I am grateful that he gave me admit at the Vanderbilt School of Engineering and provided me with the opportunity to work in the Distributed Object Computing (DOC) group. He not only imparted intellectual guidance but also supported me personally during difficult times. I am deeply indebted to him for always being there for research discussions, advice, reviews and feedback, while also teaching how to be an independent researcher.

I would like to thank Dr. Douglas C. Schmidt, Dr. Gautam Biswas, Dr. Janos Sztipanovits, and Dr. Xenofon Koutsoukos for serving on my dissertation committee and their feedback. I am grateful to Dr. Xenofon Koutsoukos for insightful discussions, suggestions and comments at the DDDAS project meetings. I appreciate the feedback and research contributions from senior researchers and Professors at Vanderbilt, particularly from Dr. Gautam Biswas, Dr. Abhishek Dubey, Dr. Hongyang Sun, Dr. Guillaume Aupy, and Dr. John S. Kinnebrew.

My research has been supported by various agencies and this work would not have been possible without their financial assistance. I would like to thank National Science Foundation (NSF) for their grants EAGER 1257955 and US Ignite CNS 1531079, and Air Force Office of Scientific Research (AFOSR) for DDDAS FA9550-13-1-0227 and DDDAS FA9550-18-1-0126.

I appreciate the feedback and research contributions on multiple projects from my colleagues, Ajay Chhokra and Hamzah Abdel-Aziz. I would also like to thank the members of DOC group: Anirban Bhattacharjee, Yogesh Barve, Shweta Khare, Shunxing Bao, Kyoungho An, Faruk Caglar, Subhav Pradhan and Prithviraj Patil for their collaboration, feedback and encouragement. I would especially like to thank Dr. Faruk Caglar and Dr. Kyoungho An for being my mentors, and showing me research direction during my initial years as a doctoral student.

Finally, I would like to acknowledge my parents, Pushpa and Nageshwar Prasad Choudhary, and rest of my family for their patience, support and encouragement. Most importantly, I would like to express my deepest gratitude to my beloved wife, Kasturi for her help, understanding, patience, and continuous support during the ups and downs of my Ph.D. journey.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

I.1 Emerging Trends

The elastic properties and cost benefits of the cloud has made it an attractive hosting platform for a variety of soft real-time Cyber Physical Systems (CPS)/Internet of Things (IoT) applications, such as cloud gaming, cognitive assistance, patient health monitoring and industrial automation. The stringent quality of service (QoS) considerations of these applications mandate both predictable performance from the cloud and lower end-to-end network latencies between the end user and the cloud. To date, security and performance assurance continues to be a hard problem to resolve in cloud platforms due to their virtualized and multi-tenant nature [52]. Although recent advances in fog and edge computing have enabled cloud resources to move closer to the CPS/IoT devices thereby mitigating the network latency concerns to some extent [30], there is still a general lack of scientific approaches that can dynamically manage resources across the cloud-edge spectrum. This is a hard problem to resolve due to the highly dynamic behaviors of the edge and cloud. Consequently, any pre-defined and fixed set of resource management policies will be rendered useless for hosting CPS/IoT applications across the cloud to edge spectrum.

A promising approach for resolving these challenges is to apply the dynamic data driven application systems (DDDAS) paradigm [47]. DDDAS prescribes an approach where the applications being controlled are instrumented adaptively so that their models can be learned and enhanced continuously, and in turn these models can be analyzed and used in a feedback loop to steer the target applications along their intended trajectories. Previous works that have leveraged the DDDAS paradigm have focused on either a domain-specific application instead of the infrastructure, or applied DDDAS for resilience and security [16]. In our case, we treat the distributed infrastructure ranging from the edge to the cloud as the target application that is to be managed and controlled. To that end, we propose to apply the DDDAS principles to the pool of resources spanning the cloud-edge spectrum for enabling and enforcing dynamic resource management decisions that deliver the required QoS properties to cloud-hosted domain-specific applications. Specifically, we propose *Dynamic Data Driven Cloud and Edge Systems (D^{3}CES),* which uses performance data collected from instrumented cloud and edge resources to learn and enhance models of the distributed resource pool. It then uses these models in a feedback loop to make effective and dynamic resource management decisions to host CPS/IoT applications and deliver their QoS properties. We now give an overview of the key research challenges and our solutions to address them.

I.2 Key Research Challenges and Solution Needs

Dynamic resource management across the cloud-edge resource spectrum is a hard problem for a variety of reasons stemming from having to address (a) the applications' functional and QoS requirements, (b) the cloud providers' ability to satisfy the service level objectives (SLOs) of all its customers while maintaining healthy revenues and keeping energy costs low, and (c) dynamically instrumenting the resources to collect measurements needed to learn the models of the distributed resource pool. These challenges span the centralized data center (CDC) hosted as the public or private cloud [118], micro data centers (MDCs) that reside at the edge of the network, also known as fog or cloudlet [143], and the mobile edge devices themselves [78]. Our research calls for an effective use of resources across this spectrum. In this context, we have identified a number of challenges that we address in this doctoral research and organized these challenges along three dimensions as described below:

I.2.1 Application-imposed Challenges

1. **Workload variations:** The workload generated by CPS/IoT applications may illustrate both transient and sustained variability, which cannot be known ahead-of-time and hence needs to be predicted and addressed. With the stringent SLO requirements for these applications, the resources need to be scaled rapidly.

2. **Stochastic execution semantics:** For some CPS/IoT applications, their uncertain and dynamic nature may require several instances of the same tasks to be executed to reach specified confidence levels. Each execution may take different execution times but impose certain QoS needs.

3. **Application structure:** Increasingly, cloud-based applications are realized as a collection of communicating micro-services, which can be deployed independently across the spectrum of resources. This gives rise to challenges where part or entire service must be migrated closer to the edge.

4. **High degree of user mobility:** CPS/IoT systems, such as autonomous transport vehicles, unmanned aerial vehicles, and mobile devices, operate in highly uncertain environments with dynamic movement profiles. Thus, a designated edge resource cannot serve such users for long durations of time.

5. **Distributed user base:** Collaborative applications such as online games may often involve a distributed set of users. Consequently, determining the appropriate MDC to migrate the application to and whether to migrate it to multiple MDCs remains an open question.

I.2.2 Cloud Provider-imposed Challenges

1. **Virtualization and multi-tenancy:** Although exploiting edge resources is an intuitive solution to addressing the network latency issues, the MDCs will also face the same challenges as a CDC, which stem from virtualization and multi-tenancy.

In a cloud setup, it is common to overbook resources [35, 162] resulting in performance deterioration of the application. Even if the resources are not overbooked, performance suffers due to interference from co-located workload and sharing of non-partitionable resources [116].

2. **Workload consolidation and migration:** Applications running in virtualized containers need to be migrated from one server to the other in order to minimize the number of physical servers used while facilitating better performance across edge and cloud resources.

3. **Hardware Heterogeneity:** Cloud data center servers and edge resources will often have different architectures, speed and count of processors. They also exhibit variability in memory speed and size, storage type and size and network connectivity. In addition, recent advances in newer hardware features such as non-uniform memory access (NUMA), cache monitoring technology (CMT) and cache allocation technology (CAT) raises the management complexity. This leads to challenges in application migration, performance metric collection and performance estimation.

I.2.3 Measurement-related Challenges

1. **Data Collection:** The plethora of deployed hardware configurations with different architectures and versions makes it hard to collect various performance metrics. Modern architectures are making it easier to collect more finer-grained performance metrics, however, much more research is needed in identifying effective approaches to control the hardware and derive the best performance out of them.

2. **Lack of benchmarks:** There is a general lack of open source and effective benchmarking suites that researchers can use to conduct studies and build models of the cloud-edge spectrum of resources that subsequently can be used in resource management.

I.3 Doctoral Research Contributions: Dynamic Resource Management across the Cloud to Edge Spectrum

To resolve the range of challenges described in Section I.2 by applying the DDDAS paradigm in a novel way, this doctoral research proposes the D^3CES framework. At the onset of this research, we realized that designing and validating novel ideas for any systems research such as ours will require a solid framework to conduct empirical studies (Challenges I.2.3-1 and I.2.3-2). Consequently, we have developed a data collection and benchmarking framework that is available in open source at `https://github.com/docvu/indices`. The benchmark gathers both system and micro-architectural performance metrics while varying application workload and collocation patterns. We have utilized this framework in the following dissertation contributions, and continue to improve its capabilities:

Contribution 1: Algorithms for elastic and scalable scheduling of CPS/IoT tasks in the cloud: To address Challenges I.2.1-2 and I.2.2-1, where the CPS/IoT applications may need to execute a large number of task instances (e.g., stochastic simulations) in the cloud environment with QoS requirements, we need feedback-based algorithms that provide the desired QoS guarantees while scaling across multiple servers. With that goal, we designed the *Simulation as a Service (SIMaaS)* cloud middleware-based approach that leverages the Linux container-based infrastructure. The key research contributions include an admission control and a resource management algorithm that reduces the cost to the service provider while enhancing the parallelization of the simulation jobs by fanning out increasing number of instances as needed until the deadline of the tasks is met while simultaneously auto-tuning itself based on the feedback. Chapter II provides the research details.

Contribution 2: Exploiting fog resources to move applications closer to the users and the data sources: To address Challenge I.2.2-2, the cloud-based service must move (partially or entirely) closer to the users and data sources. This approach needs distributed coordination, synchronization and resource allocation challenges to be investigated in the context of CDCs and MDCs, and even utilizing spare capacity of other edge resources. This task also entails discovery and orchestration of those resources not provided out-of-the-box by the traditional cloud. Thus, we also need to address the Challenges I.2.2-1 and I.2.2-3.

Our contributions have addressed the needs of applications such as augmented reality that offload computer vision algorithms including SIFT [111] for processing at the cloud. For this work, we have assumed near constant workload and low mobility of the users, which is the case for image processing performed by a stationary camera. The user continuously sends data in the form of images to the cloud for processing and in turn receives responses within a specified time bound. However, due to geographical distance and network variabilities, a central cloud may be unable to meet the QoS needs, requiring MDCs closer to the user to be leveraged opportunistically. As highlighted in Challenges I.2.2-1, performance interference caused by co-located applications extends itself from the central cloud to edge and leads to delayed response times at the edge resources too. Our initial work [36] in this regard was targeted at the centralized cloud alone and did not address hardware heterogeneity related challenges I.2.2-3.

To address these challenges, we have formulated an optimization problem to minimize the cost to the cloud provider while meeting the QoS constraints imposed by the application. As part of our framework called *INDICES*, we perform the optimization at two different layers, local, i.e. at a MDC, and global, i.e. at a CDC. Chapter III details this research.

Contribution 3: Optimizing resources across edge, fog and cloud while accounting for recent hardware enhancements: In the research described in Chapter III, we considered applications whose users remain in the same network latency and bandwidth zone throughout the execution duration. However, this is often not the case with mobile users as they move through zones of varying network connectivity and latency. Chapter IV caters to the mobility Challenge I.2.1-4. Furthermore, advances in hardware architecture such as NUMA and Cache Monitoring Technology (CMT) requires advancing existing micro-benchmarking and performance prediction techniques, which we address in this chapter as part of Challenge I.2.2-3 and propose our solution known as *URMILA*.

Contribution 4: Rapid scaling of latency-sensitive applications for workload variations: All the above contributions address resource needs by either migrating the application (Chapters III and IV) or by instantiating newer instances of the application (Chapter II). However, these mechanisms can be too costly for scenarios where the resource requirement is transient and immediate. This could occur due to workload variability as stated in Challenge I.2.1-1, or due to performance interference from co-located workload described in Challenge I.2.2-1. Hence, in Chapter V we propose a mechanism to forecast the workload and rapidly adjust the resources in order to meet the latency requirements while optimally trading-off resources.

I.4 Organization

Th rest of the dissertation is organized as follows. Chapter II describes the Simulation-as-a-Service (SIMaaS) research contributions; and Chapter III describes the INDICES research contributions; Chapter IV builds upon the INDICES approach by relaxing a number of assumptions we made including the ones regarding mobility and incorporating hardware advances such as NUMA and CMT to propose our solution URMILA; and Chapter V caters to vertical elasticity of latency-sensitive applications and thus completes the D^3CES framework. Finally, Chapter VI summarizes the research.

CHAPTER II

A SIMULATION AS A SERVICE CLOUD MIDDLEWARE

II.1 Motivation

With the advent of the Internet of Things (IoT) paradigm [15], which involves the ubiquitous presence of sensors, there is no dearth of collected data. When coupled with technology advances in mobile computing and edge devices, users are expecting newer and different kinds of services that will help them in their daily lives. For example, users may want to determine appropriate temperature settings for their homes such that their energy consumption and energy bills are kept low yet they have comfortable conditions in their homes. Other examples include estimating traffic congestion in a specific part of a city on a special events day. Any service meant to find answers to these questions will very likely require substantial number of computing resources. Moreover, users will expect a sufficiently low response time from the services.

Deploying these services in-house is unrealistic for the users since the models of these systems are quite complex to develop. Some models may be stochastic in nature, which require a large number of compute-intensive executions of the models to obtain outcomes that are within a desired statistical confidence interval. Other kinds of simulation models require running a large number of simulation instances with different parameters. Irrespective of the simulation model, individual users and even small businesses cannot be expected to acquire the needed resources in-house. Cloud computing then becomes an attractive option to host such services particularly because hosting high performance and real-time applications in the cloud is gaining traction [12, 117]. Examples include soft real-time applications such as online video streaming (e.g., Netflix hosted in Amazon EC2), gaming (Microsoft's Xbox One and Sony's Playstation Now) and telecommunication management [65].

Given these trends, it is important to understand the challenges in hosting such simulations in the cloud. To that end we surveyed prior efforts [62, 100, 105, 107] that focused on deploying parallel discrete event simulations (PDES) [61] in the cloud, which reveal that the performance of the simulation deteriorates as the size of the cluster distributed across the cloud increases. This occurs due primarily to the limited bandwidth and overhead of the time synchronization protocols needed in the cloud [165]. Thus, cloud deployment for this category of simulations is still limited.

Despite these insights, we surmise that there is another category of simulations that can still benefit from cloud computing. For example, complex system simulations that require statistical validation or those that compare simulation results under different constraints and parameter values often need to run repeatedly are suited to cloud hosting. Running these simulations sequentially is not a viable option as user expectations in terms of response times have to be met. Hence there is a need for a simulation platform where a large number of independent simulation instances can be executed in parallel and the number of such simulations can vary elastically to satisfy specified confidence intervals for the results. Cloud computing becomes an attractive platform to host such capabilities [161]. To that end we have architected a cloud-based solution comprising resource management algorithms and middleware called Simulation-as-a-Service (SIMaaS).

It is possible to realize SIMaaS on top of traditional cloud infrastructure, which utilize a virtual machine (VM)-based data center to provide resource sharing. However, in a scenario where real-time decisions have to be made based on running a large number of multiple, short-duration simulations in parallel, the considerable setup and tear down overhead imposed by VMs, as demonstrated in Section II.5.2, is unacceptable. Likewise, a solution based on maintaining a VM pool that is used by many cloud resource management frameworks such as [39, 64, 89, 190] is not suitable either since it can lead to resource wastage and may not be able to cater to sudden increases in service demand. Thus, a lightweight solution is desired.

To address these challenges, we make the following key contributions in this chapter:

- We propose a cloud middleware for SIMaaS that leverages Linux container [112]-based infrastructure, which has low runtime overhead, higher level of resource sharing, and very low setup and tear down costs.

- We present a resource management algorithm, that reduces the cost to the service provider and enhances the parallelization of the simulation jobs by fanning out more instances until the deadline is met while simultaneously auto-tuning itself based on the feedback.

- We show how the middleware intelligently generates different configurations for experimentation, and intelligently schedules the simulations on the Linux container-based cloud to minimize cost while enforcing the deadlines.

- Using two case studies, we show the viability of a Linux container-based SIMaaS solution, and illustrate the performance gains of a Linux container-based approach over hypervisor-based traditional virtualization techniques used in the cloud.

The rest of this chapter is organized as follows: Section II.2 deals with relevant related work comparing them with our contributions; Section II.3 provides two use cases that drive the key requirements that are met by our solution; Section II.4 presents the system architecture in detail; Section II.5 validates the effectiveness of our middleware; and finally Section II.6 presents concluding remarks alluding to lessons learned and opportunities for future work.

II.2 Related Work

This section presents relevant related work and compares them with our contributions. We provide related work along three dimensions: simulations hosted in the cloud, cloud frameworks that provide resource management with deadlines, and container-based approaches. These dimensions of related work are important because realizing SIMaaS requires effective resource management at the cloud infrastructure-level to manage the life-cycle of containers that host and execute the simulation logic such that user-specified deadlines are met.

II.2.1 Related Work on Cloud-based Simulations

The mJADES [134] effort is closest to our approach in terms of its objective of supporting simulations in the cloud. It is founded on a Java-based architecture and is designed to run multiple concurrent simulations while automatically acquiring resources from an ad hoc federation of cloud providers. DEXSim [42] is a distributed execution framework for replicated simulations that provides two-level parallelism, i.e., at CPU core-level and at system-level. This organization delivers better performance to their system. In contrast, SIMaaS does not provide any such scheme; rather it relies on the OS to make effective use of the multiple cores on the physical server by pinning container processes to cores. The RESTful interoperability simulation environment (RISE) [11] is a cloud middleware that applies RESTful APIs to interface with the simulators and allows remote management through Android-based handheld devices. Like RISE, SIMaaS also uses RESTful APIs for clients to interact with our service and for the internal interaction between the containers and the management solution. PADS [23, 24] provides an environment focused on teaching distributed systems algorithms, and supports running cloud hosted simulations in a Docker based execution environment. Another similar work presents running distributed co-simulations in the cloud [22].

In contrast to these works, SIMaaS applies an adaptive resource scheduling policy to meet the deadlines based on the current system performance. Also, our solution uses Linux containers that are more efficient and more suitable to the kinds of simulations hosted by SIMaaS than the VM-based approaches used by these solutions.

CloudSim [38] is a toolkit for modeling and simulating VMs, data centers and resource allocation policies without incurring any cost, which in turn helps to measure the feasibility and tune the performance bottlenecks. EMUSIM [37] enhances CloudSim by integrating an emulator to achieve the same purpose. SimGrid [40] is another distributed systems simulator used to improve the algorithms for data management infrastructure. We believe that the contributions of SIMaaS are orthogonal to these work. These related projects provide the platforms to evaluate resource allocation algorithms in the cloud while SIMaaS is a concrete realization of infrastructure middleware that supports different resource allocations. SIMaaS can benefit from these related work where resource management algorithms can first be evaluated in these platforms, and then deployed in the SIMaaS middleware. Additionally, we believe these related work do not yet support support Linux container based simulation of the cloud.

II.2.2 Related Work on Cloud Resource Management

There has been some work in cloud resource management to meet deadlines. Aneka [39] is a cloud platform that supports quality of service (QoS)-aware provisioning and execution of applications in the cloud. It supports different programming models, such as bag of tasks, distributed threads, MapReduce, actors and workflows. Our work on SIMaaS applies an advanced version of a resource management algorithm that is used by Aneka in the context of our Linux container-based lightweight virtualization solution. Aneka also provides algorithms to provision hybrid clouds to minimize the cost and meet deadlines. Although SIMaaS does not use hybrid clouds, our future work will consider some of the functionalities from Aneka.

Another work close to our resource allocation policy is [164] that employs a cost-efficient scheduling heuristics to meet the deadline. However, this work is sensitive to execution time estimation error, whereas our work self-tunes based on feedback.

CometCloud [89] is a cloud framework that provides autonomic workflow management by addressing changing computational and QoS requirements. It adapts both application and infrastructure to fulfill its purpose. CLOUDRB [155] is a cloud resource broker that integrates deadline-based job scheduling policy with particle swarm optimization-based resource scheduling mechanism to minimize both cost and execution time to meet a user-specified deadline. Zhu et al. [190] employed a rolling-horizon optimization policy to develop an energy-aware cloud data center for real-time task scheduling. All these efforts provide scheduling algorithms to meet deadlines on virtual machine-based cloud platforms where they maintain a VM pool and scale up or down based on constraints. In contrast to these efforts, our work uses a lightweight virtualization technology based on Linux containers which provides significant performance improvement and mitigates the need to keep a pool of VMs or containers. We also apply a heuristic based feedback mechanism to ensure deadlines are met with minimum resources.

In prior work [99, 145], we have designed and deployed multi-layered resource management algorithms integrated with higher-level task (re-)planning mechanisms to provide performance assurances to distributed real-time and embedded applications. These algorithms were integrated within middleware solutions that were deployed on a distributed cluster of machines, which can be viewed as small-scale data centers. These prior works focused primarily on affecting the application, such as migrating application components, load balancing, fault tolerance, deployment planning and to some extent scheduling. We view these prior works of ours as complementary to the current work. In the present work, we are more concerned with allocating resources on-demand. A more significant point of distinction is that the prior works focused on distributed applications that are long running while in current work we are focusing on applications that have a short running time but where we need to execute a large number of copies of the same application.

II.2.3 Related Work using Linux Containers

The Docker [120] open source project that we utilize in our framework automates the deployment of applications via software containers utilizing operating system (OS)-level virtualization. Docker is not an OS-level virtualization solution; rather it uses interchangable execution enviroments such as Linux Containers (LXC) and its own *libcontainer* library to provide Container access and control.

Previous work exists on the creation [119] and benchmark testing [154] of generic Linux-based containers. Similarly, there exists work that use containers as a means to provide isolation and a lightweight replacement to hypervisors in use cases such as high performance computing (HPC) [175], reproducible network experiments [72], and peer-to-peer testing environments [19]. The demands and goals of each of these three efforts focus on a different aspect of the benefit stemming from the use of containers. For HPC, the effort focused more on the lightweight nature of containers versus hypervisors. The peer-to-peer testing work focused on the isolation capabilities of containers whereas the reproducible network experiments paper focused more on the isolation features and the ability to distribute containers as deliverables for others to use in their own testing. Our work leverages or can leverage all these benefits.

II.3 Motivating Use Cases and Key Requirements for SIMaaS

We now present two use cases belonging to systems modeling that we have used in this chapter to bring out the challenges that SIMaaS should address, and to evaluate its capabilities.

II.3.1 System Modeling Use Cases

System modeling for simulations is a rich area that has been used in a wide range of different engineering disciplines. The type of system modeling depends on the nature of the system to be modeled and the level of abstraction needed to be achieved through the simulation. We use two use cases to highlight the different types of simulations that SIMaaS is geared to support.

II.3.1.1 Use Case 1: The Multi-room Heating System

In use case 1, we target complex engineering systems which exhibit continuous, discrete, and probabilistic behaviors, known as stochastic hybrid systems (SHS). The computer model we use to construct a formal representation of a SHS system and to mathematically analyze and verify it in a computer system is the *discrete time stochastic hybrid system (DTSHS)* model [6].

We discuss here a DTSHS model of a multi-room heating system [13] with its discretized model developed by [5]. The multi-room heating system consists of h rooms and a limited number of heaters n where $n < h$. Each room has at most one heater at a time. Moreover, each room has its own user setting (i.e., constraints) for temperature. However, the rooms have an exchangeable effect with their adjacent rooms and with the ambient temperature.

Each room heater switches independently of the heater status of other rooms and their temperatures. The system has a hybrid state where the discrete component is the state of the individual heater, which can be in ON or OFF state, and the continuous state is the room temperature. A discrete transition function switches the heater's status in each room based on using a typical controller which switches the heater on if the room temperature gets below a certain threshold xl and switches the heater off if the room temperature exceeds a certain threshold xu.

The main challenge for our use case is the limited number of heaters and the need for a control strategy to move a heater between the rooms. Typical system requirements that can be evaluated using simulations are:

- The temperature in each room must always remain above a certain threshold (i.e., user comfort level).

- All rooms share heaters with other rooms (i.e., acquire and relinquish a heater).

In our model of the system, we have used one of many possible strategies where room i can acquire a heater with a probability p_i if:

- $p_i \propto get_i - x_i$ when $x_i < get_i$.
- $p_i = 0$ when $x_i \geq get_i$.

where get_i is control threshold used to determine when room i needs to acquire a heater. The simulation model for this use case uses statistical model checking by Bayesian Interval Estimates [191].

II.3.1.2 Use Case 2: Traffic Simulation for Varying Traffic Density

Use case 2 targets transportation researchers and traffic application providers, such as Transit Now (`http://transitnownashville.org/`), who want to model and simulate different traffic scenarios within a relevant time window but do not have sufficient resources to do it in-house. We motivate this use case with a microscopic traffic simulator called SUMO [26] that can simulate city level traffic. The simulator can import a city map in OpenStreetMap [71] format to its own custom format. The user can supply various input parameters such as number of vehicles, traffic signal logic, turning probability, maximum lane speed and study their impact on traffic congestion.

One such "what if" scenario involves the user changing the number of vehicles moving in a particular area of the city and studying its impact. In contrast to use case 1 where all the stochastic simulation instances had nearly the same execution time in ideal conditions, in this use case, the simulation execution time varies with the input number of vehicles. Figure 1 illustrates how the execution time varies with the number of vehicles for a duration of 1000 seconds.

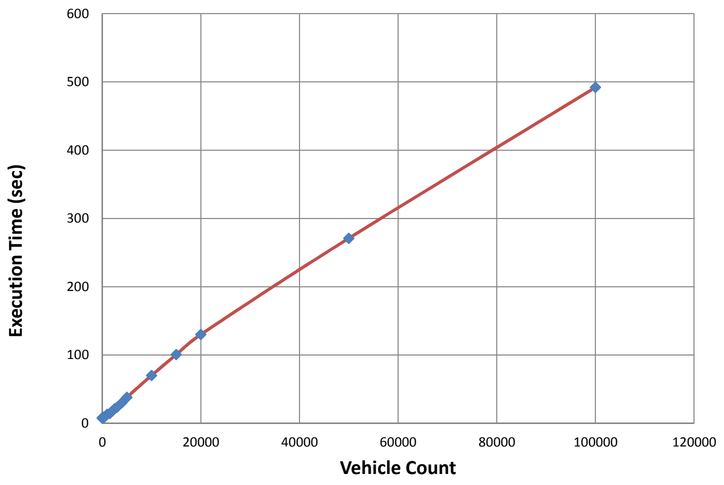


Figure 1: Simulation Execution Time

II.3.2 Problem Statement and Key Requirements for SIMaaS

Based on the two use cases described above, we now bring out the key requirements that must be satisfied by SIMaaS. Addressing these requirements forms the problem statement for our research presented in this chapter.

• **Requirement 1: Ability to Elastically Execute Multiple Simulations –** Recall that the simulation model for use case 1 is stochastic, which means that every simulation execution instance may yield a different simulation trajectory and results. To overcome this problem, we have to use the *statistical model checking (SMC)* approach based on Bayesian statistics [190, 191]. SMC is a verification method that provides statistical evidence to check whether a stochastic system satisfies a wide range of temporal properties with a certain probability and confidence level or not. The probability that the model satisfies a property can be estimated by running several different simulation trajectories of the model and dividing the number of satisfied trajectories (i.e., true properties) over the total number of simulations. Thus, SMC requires execution of a large number of simulation tasks.

On the other hand, although the simulation models for use case 2 are not stochastic, the result of the simulation will often be quite different depending on the parameters supplied to the model. For example, varying the number of vehicles on the road, number of traffic lights, number of lanes, and speed limits will all generate different results. A user may be interested in knowing the results for various scenarios, which in turn requires a number of simulations to be executed seeded with different parameter values. In addition, the service will be used by multiple users who need to execute different number of simulations, which is not known to the system a priori. This requirement suggests the need to elastically scale the number of simulation instances to be executed.

In summary, the two use cases require that SIMaaS be able to elastically scale the number of simulations that must be executed.

• **Requirement 2: Bounded Response Time –** In both our use cases, the user expects that the system respond to their requests within a reasonable amount of time. Thus, the execution of a large number of simulations that are elastically scheduled on the cloud platform, and result aggregation must be accomplished within a bounded amount of time so that it is of any utility to the user. Moreover, use case 2 illustrates an additional challenge that requires estimating the expected execution time for previously unknown parameters and ensuring that the system can still respond to user request in a timely manner.

In summary, SIMaaS must ensure bounded response times to user requests.

• **Requirement 3: Result Aggregation –** Both our use cases highlight the need for result aggregation. In use case 1, there is a need to aggregate the results from the large number of model executions to illustrate the confidence intervals for the results. In use case 2, the user will need a way to aggregate results of each run corresponding to the parameter values. Since SIMaaS is meant to be a broadly applicable service, it will require the user to supply the appropriate aggregation logic corresponding to their needs.

In summary, SIMaaS needs an ability to accept user-supplied result aggregation logic, apply it to the results of the simulations, and present the results to the user.

• **Requirement 4: Web-based Interface –** Since SIMaaS is envisioned as a broadly applicable cloud-based, simulation-as-a-service, it will not know the details of the user's simulation model. Instead, it will require the user to supply a simulation model of their system and various parameters to indicate how SIMaaS should run their models. For example, since use case 1 requires stochastic model checking, it will require a large number of simulation trajectories to be executed. Thus, SIMaaS will require the user to supply the simulation image and specify how many such simulations should be executed, the building layout, the number of heaters, the strategy used and so on. Similarly, for use case 2, SIMaaS will need to know how the model should be seeded with different parameter values and how they should be varied, which in turn will dictate the number of simulations to execute and their execution time. Finally, the aggregated results must somehow be displayed to the user.

In summary, SIMaaS should provide a web-based user interface to the users so they can supply both the simulation model and the parameters as well as receive the results using the interface.

II.4 SIMaaS Cloud Middleware Architecture

A cloud platform is an attractive choice to address the requirements highlighted in Section II.3.2 because it can elastically and on-demand execute the multiple different simulation trajectories of the simulation models in parallel, and perform aggregation such as SMC to obtain results within a desired confidence interval. The challenge stems from provisioning these simulation trajectories in the cloud in real-time so that the response times perceived by the user are acceptable. To that end we have architected the SIMaaS cloud-based simulation-as-a-service and its associated middleware as shown in Figure 2. The remainder of this section describes the architecture and shows how it addresses all the requirements outlined earlier.

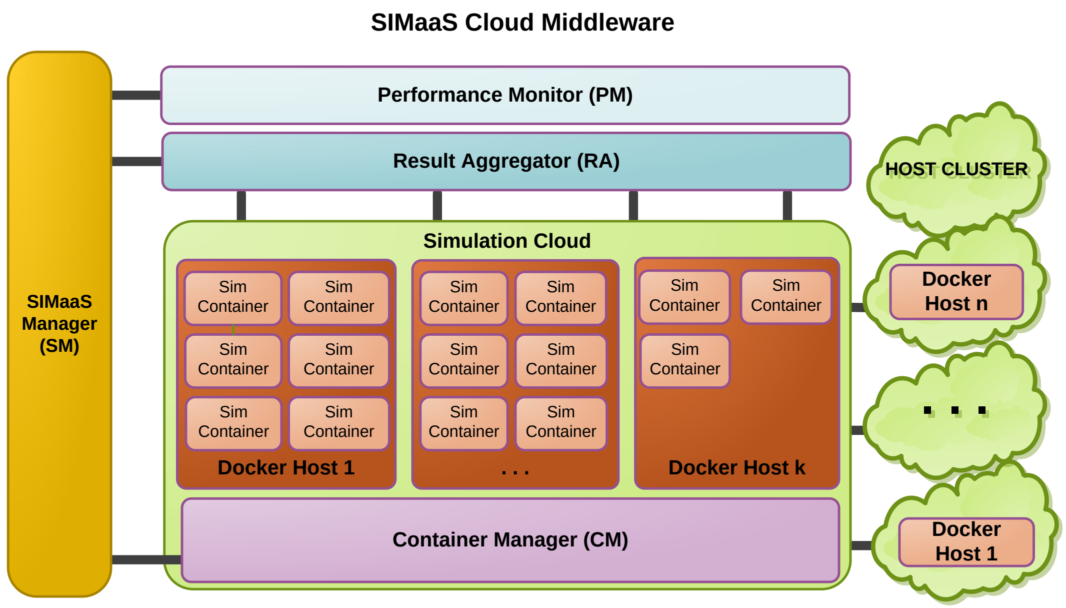


Figure 2: System Architecture

II.4.1 Dynamic Resource Provisioning Algorithm: Addressing Requirements 1 and 2

Requirement 1 calls for elastic deployment of a large number of simulation executions depending on the use case category, which needs dynamic resource management. Requirement 2 calls for timely response to user requests. Thus, the dynamic resource management algorithm should be geared towards meeting the user needs.

For this chapter, we define a QoS-based resource allocation policy that allocates containers for each requested simulation model such that its deadline is met and its cost, i.e.

the number of assigned containers is minimized. We assume that the user provides the following inputs to our allocation algorithm: simulation model, number of simulations and their corresponding simulation parameters, and the estimated execution time using some of the simulation parameters.

Formally, we define the requested execution of the simulation model as a *job*. Each job is made up of several different tasks representing an execution instance of that simulation job. At any instant in time k, $J(k)$ is the set of jobs which our allocation algorithm handles. Furthermore, for the j^{th} job, $J_j(k) \in J(k)$, we define its deadline as DL_j, the number of containers it uses as $B_j(k)$, its i^{th} simulation task as $T_{ij}(k)$, the simulation parameter of its i^{th} task as θ_{ij} and finally, the expected execution time of its i^{th} task with its corresponding parameter θ_{ij} as $E_{\theta_{ij}}(T_{ij}(k))$.

The primary objective of our allocation algorithm is to minimize the resource usage cost considered in terms of the number of containers used to serve the user simulation request, while maintaining the user constraint stated as meeting the deadline. To formalize this objective, we define it as the following optimization problem:

$$\begin{aligned} \min_{B_j} \quad & c(K) = \sum_j \sum_k B_j(k) = c(K-1) + \sum_j B_j(K) \\ \text{subject to} \quad & \forall j \in J(k), \frac{R_j(k)}{B_j(k)} = \frac{\sum_i E_{\theta_{ij}}(T_{ij}(k))}{B_j(k)} \leq DL_j \end{aligned}$$

where, $c(k)$ is the cost function at time instant k, and $R_j(k)$ is the j^{th} job's total execution time which is equal to the summation of the execution time $E_{\theta_{ij}}(T_{ij}(k))$ for all the unserved tasks $T_{ij}(k)$. This constraint equation calculates the total time a job would take to finish if its simulations' executions have been parallelized using $B_j(k)$ containers. Therefore, it bounds the selection of $B_j(k)$ such that each job finishes before or by the deadline. To tackle this problem, we developed a simple heuristic shown in Algorithm 1 for efficiently selecting the minimum $B_j(k)$ such that each job finishes its required simulation tasks by their deadline.

Algorithm 1 QoS-based Resource Allocation Policy

```
Require: $J$, $\alpha$;
1: while TRUE do
2:     Wait for(max(feedback_event, minimum_period));
3:     for all $J_j \in J$ do
4:         ▷ Update only jobs with new feedback data;
5:         if (HasFeedback($J_j$)) then
6:             ▷ Update the estimated error factor;
7:             $F_j \longleftarrow$ UpdateErrorFactor($E^{*}_{\theta_{ij}}(T_{ij})$);
8:             ▷ Update the estimated execution time function;
9:             UpdateExecutionTimeFunction($F_j$);
10:            ▷ Update the number of containers with their scheduling;
11:            extraContainersNeeded $\longleftarrow$BestFitDecreasing($DL_j$) - $B_j$;
12:            if extraContainersNeeded $>$ 0 then
13:                Reserve(extraContainerNeeded, $J_j$);
14:                ▷ Avoid frequent resource allocation and de-allocation;
15:            else if extraContainersNeeded $<$ threshold then
16:                Release(extraContainerNeeded, $J_j$);
```

Formally, we calculate B_j using the following formula. To simplify the notation, we will omit the time index k throughout the remainder of this section:

$$B_j = \frac{R_j}{DL_j}$$

Two major challenges arise when calculating B_j based on the above formula. First, it is difficult to calculate analytically R_j in a mathematically closed form because a task's execution time varies based on many dynamic factors such as performance interference, overbooking ratio, etc. Second, the execution times of a job's tasks are not necessarily identical when they have different simulation parameter θ_{ij}, as demonstrated in Figure 1. Therefore, the above formula is not accurate and we may need to increase the value of B_j calculated above in order to meet the deadline. Moreover, scheduling the job's tasks T_{ij} in the reserved containers B_j is a non-trivial task.

To overcome the first challenge, we make our heuristic calculation of B_j based on an estimated execution time $E'_{\theta_{ij}}(T_{ij})$ of each task T_{ij} and periodically update this estimation and consequently the B_j calculation based on the feedback of the actual executed time $E^*_{\theta_{ij}}(T_{ij})$. This simple feedback mechanism allows us to maintain our algorithm objective mentioned above while tolerating the effect of estimation errors, and handling the dynamic change of our system environment (e.g. performance interference). Furthermore, our algorithm has to wait for at least a minimum period of time even if there is a new feedback, in order to enhance the algorithm's performance by avoiding high frequent recalculation. In addition to this, we recalculate F_j and $E'_{\theta_{ij}}(T_{ij})$ in every iteration and the algorithm is executed only for jobs that have new feedback data.

In order to estimate the execution time R_j of the j^{th} job, we use an initial execution time function $E_{\theta_{ij}}(T_{ij})$ as an initial function to our estimator. Since the user provides the estimated execution time for only a few parameters, we use a regression algorithm based on sequential exponential kernel [138] to build the initial function of $E_{\theta_{ij}}(T_{ij})$ using the data point provided by the user. Then, we update this function by an error factor F_j calculated using the feedback data. The calculation of the error factor F_j and the estimated execution time function $E'_{\theta_{ij}}(T_{ij})$ are shown in the following equation:

$$E'_{\theta_{ij}}(T_{ij}) = E_{\theta_{ij}}(T_{ij}) \times (1 + F_j)$$

such that:

$$F_j = \mathbb{E}[\Delta E_{\theta_{ij}}(T_{ij})/E_{\theta_{ij}}(T_{ij})] + \alpha \times \sqrt{Var\left[\Delta E_{\theta_{ij}}(T_{ij})/E_{\theta_{ij}}(T_{ij})\right]}$$

$$\Delta E_{\theta_{ij}}(T_{ij}) = E_{\theta_{ij}}(T_{ij}) - E^*_{\theta_{ij}}(T_{ij})$$

where $\alpha \geq 0$ is an estimator parameter that determines how pessimistic is our estimator because F_j covers more errors as α increases. For example, when $\alpha = 1, 2, 3, F_j$ will approximately estimate the worst-case scenario of $68\%, 95\%$, and 99.7% of the feedback error values, respectively. For a real-time implementation of error factor calculation, we use an online algorithm developed by Knuth [91] to calculate $\mathbb{E}[.]$ and $Var[.]$ incrementally, in order to avoid saving and inspecting the entire feedback data every time a new feedback entry has arrived.

To overcome the second challenge, we used best fit decreasing bin packing algorithm [88], where, we pass DL_j to this bin packing algorithm as its bin's size input, and the estimated task execution times $E'_{\theta_{ij}}(T_{ij})$ as its items' size input. Therefore, the number of slots produced by the bin packing algorithm represents the required containers B_j and the distribution of the tasks T_{ij} in each slot represents the tasks' schedule over the containers.

Note that the system resources constraints limit the number of jobs in J that can be serviced at the same time. Therefore, we use an admission control algorithm shown in Algorithm 2 to maintain a reliable service. The admission control algorithm is used to accept any new incoming user request which can be handled using the remaining available resources without missing its and other running jobs deadline. It basically estimates the number of containers needed to serve the new request such that it finishes by its deadline. Then, it checks whether there are idle containers available to serve it or not. The algorithm has two administrator configurable parameters $(\beta \geq 1)$ and $(\gamma \geq 0)$ which add a margin of resources to overcome the estimation error and to maintain another margin of resources for other running jobs to be used by the above allocation algorithm.

Algorithm 2 Admission Control Algorithm

```
Require: availableCapacity, newJob
Ensure: Accepted/Rejected
1: containersNeeded ⟵BestFitDecreasing($DL_{newJob}$);
2: if containersNeeded $\times \beta$ < availableCapacity $-\gamma$ then
3:     Accept newJob;
4: else
5:     Reject newJob;
```

II.4.2 Dynamic Resource Provisioning Middleware: Addressing Requirements 1 and 2

The second aspect of dynamic resource management is the middleware infrastructure that encodes the algorithm and provides the service capabilities. The middleware aspect is described here.

II.4.2.1 Architectural Elements of the SIMaaS Middleware

The central component of the SIMaaS middleware shown in Figure 2 that is responsible for resource provisioning and handling user requests is the SIMaaS Manager (SM). All the coordination and decision making responsibilities are controlled by this component. It employs the strategy design pattern; thus it has a pluggable design that is used to strategize the virtualization approach to be used by the hosted system. The strategy pattern also allows the SM to swap the scheduling policy if needed, however, we use a single scheduling policy during the life-cycle of SIMaaS to avoid conflicts.

A cloud platform typically uses virtualized resources to host user applications. Different types of virtualization include full virtualization (e.g., KVM), paravirtualization (e.g., Xen) and lightweight containers (e.g., LXC Linux containers). Since full and para virtualization require the entire OS to be booted from scratch whenever a new virtual machine (VM) is scheduled, this boot up time incurs a delay in availability of new VMs, not to mention the cost of the application's initialization time. All of these impact the user response time. Since Requirement 2 calls for bounded response time, SIMaaS uses the lightweight containers, which suffice for our purpose.

The life cycle of these containers is managed by the Container Manager (CM) shown in the Figure 2. The pluggable architecture of SM allows CM to switch between various container providers, which can be Linux container or hypervisor-based VM cloud. The Linux container is the default container provider of CM. Specifically, we use the Docker [120] container virtualization technology since it provides portable deployment of Linux containers and provides a registry for images to be shared across the hosts with significant performance gains over hypervisor-based approaches. Thus, the CM is responsible for keeping track of the hosts in the cluster and provision the running and tearing down of the Docker containers. It downloads and deploys different images from the Docker registry for instantiating different simulations on the cluster hosts.

Our earlier design of the CM leveraged Shipyard [152] for communicating with the Docker hosts, however, due to sluggish performance we observed, we had to implement a custom solution with a reduced role for Shipyard. Overcoming the reasons for the sluggish performance and reusing existing artifacts maximally is part of our future investigations when we also evaluate other container managers such as Apache Mesos, Google Kubernetes and Docker Swarm.

II.4.2.2 Resource Instrumentation and Monitoring

Recall that meeting user-specified deadlines is an important goal for SIMaaS (Requirement 2). These deadlines must be met in the context of either the stochastic model checking that requires multiple simultaneous runs of the stochastic simulation models or simulations executed under a range of parameter values. Thus, SIMaaS must be cognizant of overall system performance so that our resource allocation algorithm can make effective dynamic resource management decisions. To support these system requirements, effective system instrumentation is necessary.

Since SIMaaS uses Linux containers, we leveraged the Performance Monitor (PerfMon) package from the JMeter Plugins group of packages on Linux. PerfMon is an open-source Java application which runs as a service on the hosts to be monitored. Since the monitored statistics are required by the Performance Manager (PM) component instead of a visual rendition, we implemented a custom software to tap into PerfMon via its TCP/UDP connection capabilities. PerfMon is by no means the only option available but it sufficed our needs.

PerfMon depends on the SIGAR API and uses it for gathering system metrics. The metrics available are classified into eight broad catagories. These catagories include: CPU, Memory, Disk I/O, Network I/O, JMX (Java Management Extensions), TCP, Swap, and Custom executions. We are currently not using the JMX, TCP, or Swap metrics, but they are available for use if needed. Each of these catagories have parameters to allow customization of the desired returned metrics, e.g., Custom allows for the returning of any custom command line execution. We use this to execute a custom script that returns the process id and container id pairs of each running Docker container. This allows us to monitor each individual container's performance precisely.

II.4.3 Result Aggregation: Addressing Requirement 3

Stochastic model checking as in use case 1 requires that results of the multiple simulation runs be aggregated to ascertain if the specified probabilistic property is met or not. Similarly, as in use case 2, multiple simulation runs for different simulation parameters result in different outcomes, which must be aggregated and presented to the user. To accomplish this and thereby satisfy Requirement 3, a key component of our middleware is the Result Aggregator (RA). RA receives the simulation results from the Docker containers. It uses ZeroMQ messaging queue service for reliable result delivery. It has two roles: first, it sends feedback to the SM about the completion of task for decision making. Second, it performs the actual result aggregation.

Since the aggregation logic is application-dependent, it is supplied by the user when the service is hosted, and is activated when the simulation job completes. For use case 1, the aggregation logic is a Bayesian statistical model checking which produces a single string result. On the other hand, use case 2 aggregation logic parses and collates the XML files produced as the result of simulation runs.

II.4.4 Web Interface to SIMaaS - SIMaaS Workflow and User Interaction: Addressing Requirement 4

Finally, we discuss how the user interacts with SIMaaS, which is a web-based interface, and the workflow triggered by a typical user action. The interface to the Simulation Manager of SIMaaS is hosted on a lightweight web server, CherryPy [74] to interact with the user and also to receive feedback from other SIMaaS components. The interaction involves two phases. In the design phase a user interacts with the SIMaaS interface and provides the initial configuration which includes the simulation executables and the aggregation logic. A container image is generated after including hooks to send the temporary results. This image is then uploaded to a private cloud registry accessible to the container hosts. The aggregation logic is deployed in the Result Aggregator component that can collect the temporary simulation results and generate the final response.

The execution phase is depicted in Figure 3, wherein, time bounded, on-demand simulation jobs are performed. The user can use a RESTful API (or a web-form if deadline is not immediate) to supply name-value pairs of parameters. The following parameters are supplied by all the users:

- Simulation Model Name: A simulation model name is required to identify the container image and aggregation logic.
- Number of Simulations: The number of simulation instances to run.
- Deadline: The deadline for the job
- Required Resources: Number of CPUs to be allocated for each container. Other types of resources will be added in future.
- Simulation Command: The command to initiate the simulation.
- List of (Execution Time Parameter, Estimated Execution Time): This is a small set of the execution time parameter values and the corresponding execution times that is

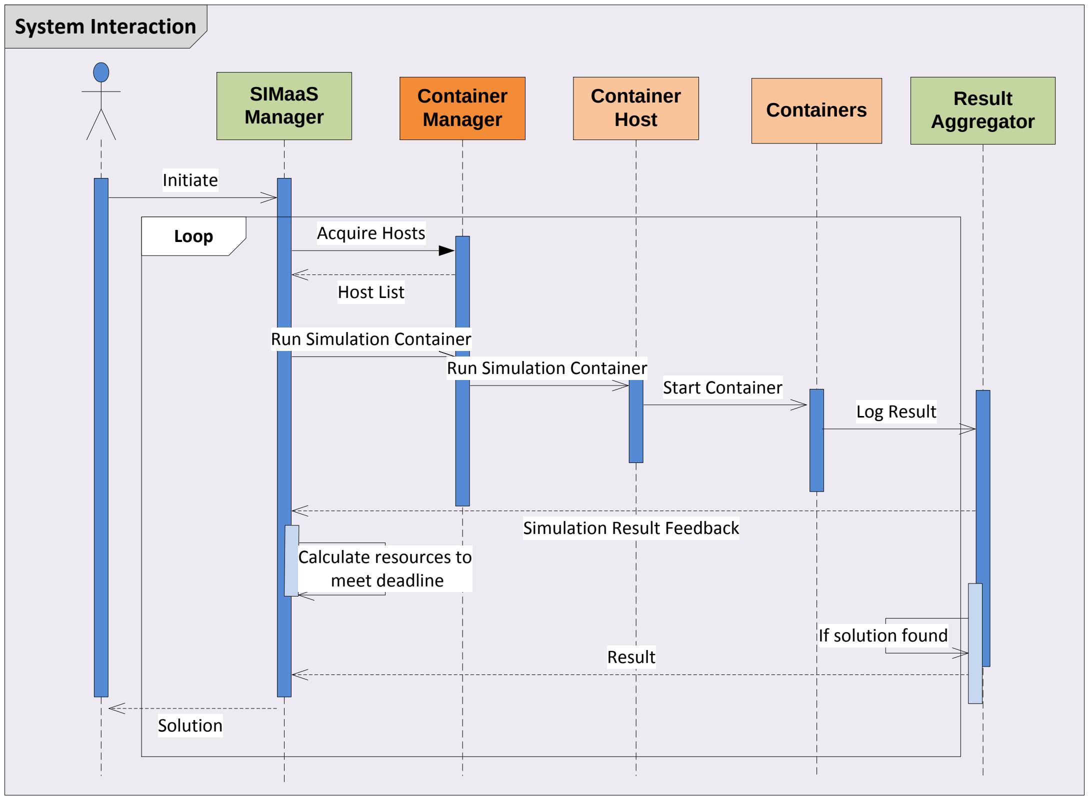


Figure 3: SIMaaS Interaction Diagram

used to generate the regression curve for estimating the unknown execution time for remaining parameters.

The following parameters are specific to the use case:

- Use Case 1 - Number of Heaters: The number of heaters active in the building (explained in section II.3.1.1)
- Use Case 1 - Sampling Rate: The rate at which data is sampled for temperature simulation.
- Use Case 1 - Strategy: The model strategy to be used.

- Use Case 1 - Confidence Level: A confidence value for the Bayesian Aggregator (explained in section II.3.1.1).
- Use Case 2 - SUMO Configuration File: A configuration file used by SUMO simulation for selecting the inputs and deciding the output details.
- Use Case 2 - List of Vehicle Counts: Different vehicle counts to be simulated.

We note that the simulation execution time is also a user input, however, this value can be determined in a sandbox environment, wherein executing a single simulation instance gives the value for a constant time simulation (such as use case 1) or by executing a subset of simulation instances from a different range of parameters (such as use case 2) and using a regression curve to estimate the execution time for others.

The request is then forwarded and processed by the SM. It validates the input and applies admission control as explained in Algorithm 2 using a resource allocation and scheduling policy, and checks if sufficient resources are available. If not, then it immediately responds to the user with a failure message. In future, based on the criticality of the request, some jobs may be swapped for a higher priority job. If the job can be scheduled, then it allocates the resources and contacts the CM to run the simulation containers. The containers log the result to RA that keeps sending feedback to the SM and performs the aggregation when the desired number of simulation results are received. The SM also runs a service, applying Algorithm 1 at a configurable interval, to determine if the deadline will be met based on the current performance data, and accordingly contacts the CM to acquire additional resources and run the containers. Once the simulation completes, the RA responds to the user with the result. Currently it uses a shared folder. However, going forward we plan to implement either an interface that can send the response as an asynchronous callback or send a notification to the user about the availability of the result.

II.5 Experimental Validation

This section evaluates the performance properties of the SIMaaS middleware and validates our claims in the context of hosting the use cases described in Section II.3.1.

II.5.1 Experimental Setup

Our setup consists of ten physical hosts each with the configuration defined in Table 12. The same set of machines were used for experimenting with both Linux containers and virtual machines. Docker version 1.6.0 was used for Linux container virtualization and QEMU-KVM was used for hypervisor virtualization with QEMU version 2.0.0 and Linux kernel 3.13.0-24.

Table 1: Hardware & Software Specification of Physical Servers

Processor	2.1 GHz Opteron
Number of CPU cores	12
Memory	32 GB
Disk Space	500 GB
Operating System	Ubuntu 14.04 64-bit

Even though our solution is designed to leverage the Linux containers instead of virtual machines, since we did not have access to large number of physical machines yet had to measure the scalability of our approach, we tested our solution over a homogeneous cluster of 60 virtual machines deployed as docker hosts for running the simulation tasks, i.e., the docker containers were spawned inside the VMs. The same set of physical machines were used to host the VMs with configuration as defined in Table 2.

Note that the SM, CM, RA and PM components of the SIMaaS middleware reside in individual virtual machines deployed on a separate set of hosts, each with 4 virtual CPUs, 8 GB memory and running Ubuntu 14.04 64-bit operating system in our private cloud managed by OpenNebula 4.6.2. The Simulation Manager was deployed on CherryPy 3.6.0

Table 2: Configuration of VM Cluster Nodes

Kernel	Linux 3.13.0-24
Hypervisor	Qemu-KVM
Number of Virtual Machines	6
Overbooking Ratio	2.0
Guest CPUs	4
Guest Memory	4 GB
Guest OS	Ubuntu 14.04 64-bit

web server. The container manager used Shipyard version v2 for managing the docker hosts. The performance monitor relied on a customized Perfmon Server Agent 2.2.3.RC1 residing in each docker host to collect performance data. The Result Aggregator utilized ZeroMQ version 4.0.4 for receiving simulation results from the docker containers.

II.5.2 Validating the Choice of Linux Container-based SIMaaS Solution

We first show why we used the container-based approach in the SIMaaS solution instead of traditional virtual machines. This set of experiments affirm the large difference in startup times for containers in Linux container-based cloud and virtual machines in hypervisor-based traditional cloud. In [115], the authors showed that there is a high start up time required on different popular public clouds. We tested similar configurations in our private cloud, managed by OpenNebula and running QEMU-KVM hypervisor. We used overbooking ratios of 1, 2 and 4 with a minimal image from use case 1. While the startup time were in the order of sub-seconds for our Linux container host, they were 176, 300 and 599 seconds, respectively, for the hypervisor host. The large start up time can be ascribed to the time taken in cloning the image as the VM file system and booting up of the operating system.

Another set of experiments were performed to compare the performance of a host running simulations using Linux container versus virtual machines. Table 3 shows that the

Linux container host performs better in most of the cases as it does not incur the overhead of running another operating system as a VM does.

Table 3: Comparison of Simulation Execution Time

	Overbooking Ratio 1	Overbooking Ratio 2	Overbooking Ratio 4
Linux Container (Physical Server - 4s)	4.74s	7.19s	13.32s
Virtual Machine (Physical Server - 4s)	5.17s	9.71s	19.05s
Linux Container (Physical Server - 50s)	50.5s	98.29s	180.45s
Virtual Machine (Physical Server - 50s)	52.4s	97.56s	202.5s

II.5.3 Workload for Container-based Experimentation

The workload we used in our experiments consists of several jobs corresponding to user requests, each having a number of simulation instances as a bag of independent tasks. The simulations are containerized as docker images on Ubuntu 14.04 64-bit operating system. The jobs are based on both the use cases described in section II.3.1, however, we created several variations of these use cases by changing the execution parameters. The building heating stochastic simulation jobs have near constant execution time, but we used three variations of it by using different sampling rates of 10, 5, and 2 milliseconds. The smaller the sampling rate, better is the accuracy of the simulation results at the expense of longer execution times. The traffic simulations jobs also had several variations based on the range of the vehicle count.

The simulations for each job may have different resource requirements that will be provided by the user. For these experiments, we have considered CPU-intensive workloads and modeled the user input as three resource types with 1, 2 or 4 CPUs per container, which is an indication of how much CPU share that container gets. Thus, we convert these values to CPU share per host as the docker container input.

We generated a synthetic workload to measure the system performance. We conducted two sets of experiments. The first set of experiments were conducted to measure the efficacy of our algorithm using a single type of job on a ten physical host cluster, whereas the second set of experiments were performed to demonstrate the scalability of our algorithm using a 60 virtual-host cluster with different types of jobs arriving at different points in time. We applied the Poisson distribution with λ of 1 for a duration of two hours to find the job arrival distribution. The number of tasks per job was uniformly distributed from 100 to 500. The deadline per job was also varied as a uniform distribution from 5 minutes to 20 minutes.

II.5.4 Evaluating SIMaaS for Meeting Deadlines and Resource Consumption

We evaluate the ability of the SIMaaS middleware to meet the user-specified deadline and its effectiveness in minimizing the resources consumed. In use case 1 described in section II.3.1.1, the user provides the approximate number of simulations needed for stochastic model checking as an input to attain the desired confidence level for the output [191]. For the second use case describe in II.3.1.2, the number of simulations is a user input. These studies were conducted for different resource overbooking ratios, simulation count, deadlines, simulation duration, and execution times. Overbooking refers to the number of times the capacity of a physical resource is exceeded. For example, suppose each container is assigned a single CPU; thus for a 12-core system, an overbooking ratio of 2 translates to 24 containers running on the host. This strategy is cost effective when the guests do not consume all the assigned resources all at the same time. We run the scheduling policy defined in Algorithm 1 at an interval of 2 secs that dynamically allocates extra hosts if the deadline cannot be met with the assigned hosts.

Test 1 – Determining the Error Estimation Parameter (α): These experiments were performed on the ten physical host cluster with use case 1 as the simulation model with the following parameters: a deadline of 2 minutes, 500 simulation tasks, one CPU per container and overbooking ratio of 2 per host. The purpose of these tests was to determine the error estimation parameter – α for our system. This value is used to calculate the error factor, explained in Section II.4.1, that plays a crucial role in meeting the deadline and allocating container slots. Recall also that our algorithm attempts to minimize the number of containers while meeting deadlines on a per simulation job basis.

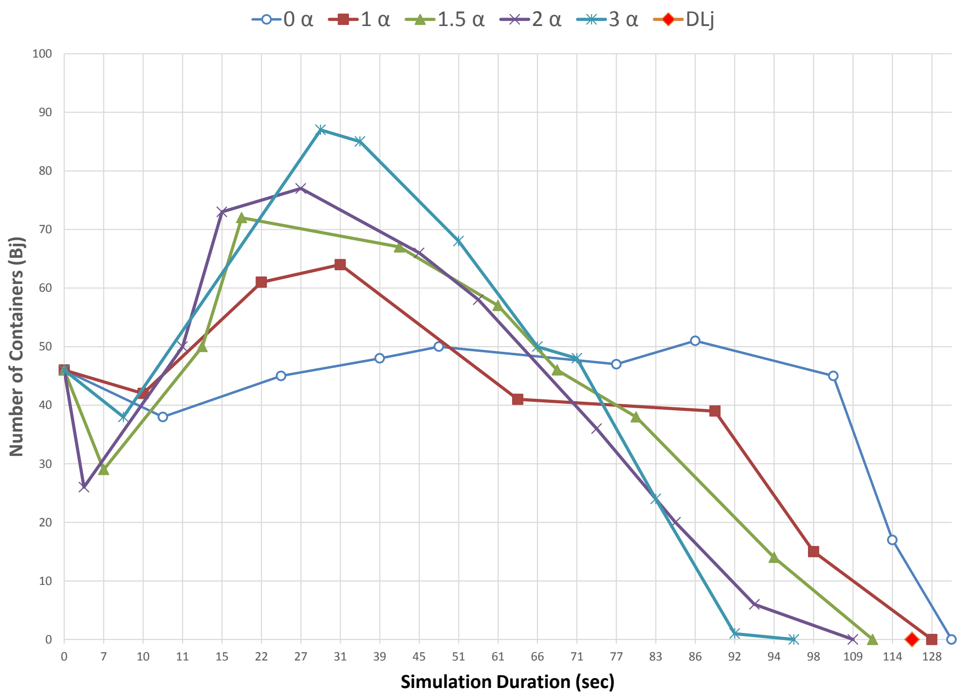


Figure 4: Container Count and Deadline Variation with α

Figure 4 depicts the simulation results for α values 0, 1, 1.5, 2 and 3. We observe that values of 0 and 1 were too low and the system missed the deadline. A value of 1.5 was found to meet the deadline as well as causing less peak resource usage. This value can be made dynamic based on the user urgency and strictness of the deadline.

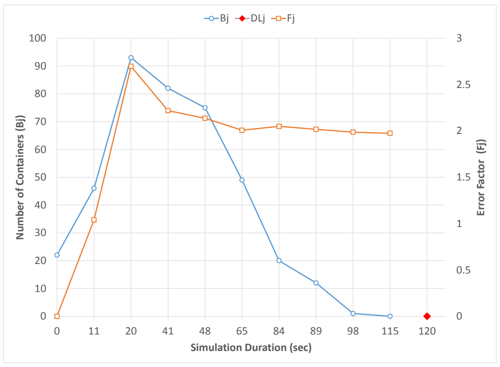
Test 2 – Studying Variations in Container Count (B_j) with Error Factor (F_j): These tests were conducted to study the impact of changing the feedback based error factor (F_j) on the container count (B_j) as well as the input simulation execution time. From Figure 5, we observe that initially, the resource consumption varies according to the error factor. Later, the resource consumption stays constant after the error estimate reaches a steady state. We conclude that the error estimate becomes accurate and close to the real value we get from the feedback. However, as the simulation moves towards completion, resources get released as lesser number of simulations remain to be executed.

Another observation we make from the results is that a pessimistic execution time estimate (here 20s) results in more initial resource allocation. Resource allocation has its own cost. For example, we measured the cost of deployment of simulation from the private registry to the docker hosts for both of our use cases. For the image deployment of heater simulation of use case 1, it took 135.1 sec and for the SUMO simulation deployment of use case 2, it took 34.6 sec. These values are network-dependent but are incurred one-time per host which can be done as a setup process.

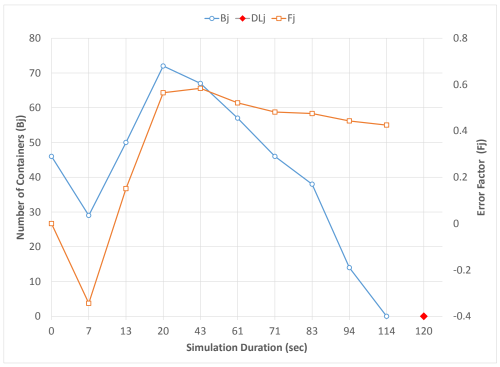
Since resource allocation incurs cost, an optimistic estimate is better because the system can adjust itself. However, if the estimate is too optimistic, the system may not be able to finish the job within the user-defined deadline.

Test 3 – Validating the Applicability of the Feedback-based Approach: The goal of applying Algorithm 1 is to spread the load per job over the deadline period so that multiple jobs can run in parallel while meeting their deadlines. In other words, the system does not schedule a job (and its tasks) immediately upon a request but delays it such that the load is balanced yet ensuring that the deadline will be met.

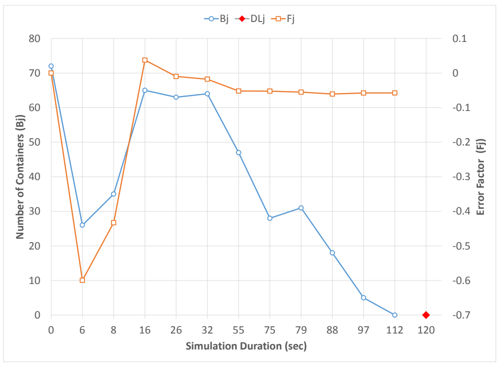
Figure 6 compares a scenario where we do not apply the feedback-based approach and instead allocate resources based on a fixed expected execution time. Estimating an accurate



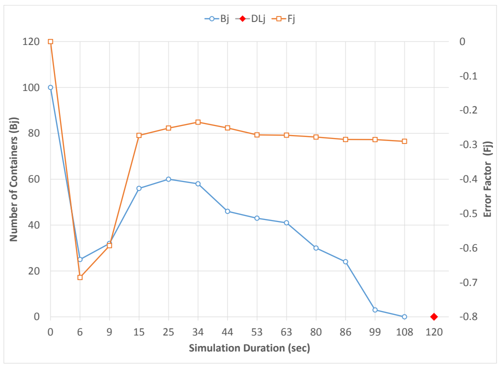
(a) 5 seconds simulation execution time



(b) 10 seconds simulation execution time



(c) 15 second simulation execution time

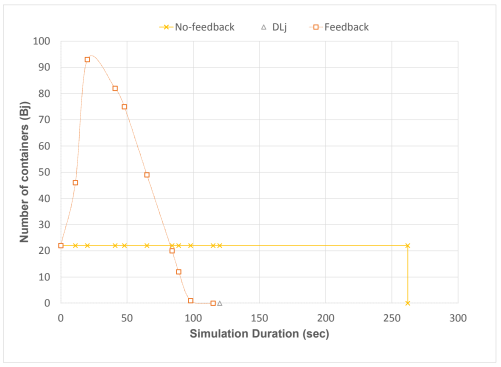


(d) 20 second simulation execution time

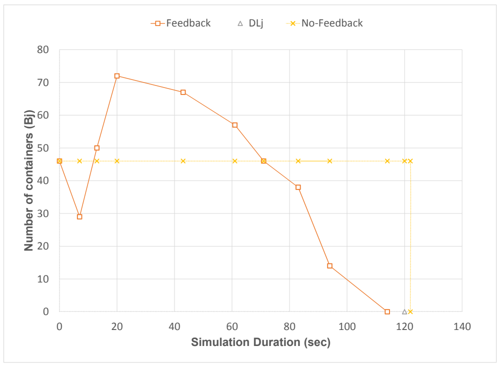
Figure 5: Variation in Error Factor (F_j) and Container Count (B_j)

execution time is not a trivial task and may not even be realistically feasible. The execution time does not just depend on the input parameter and hardware; it is also dependent on the performance interference due to other processes running on the shared resource. Too little a value, and we miss the deadline while too high value will result in wastage of resources. We observe from the results that the feedback based approach meets the deadline in all the cases while minimizing the resource consumption by releasing the containers if not needed.

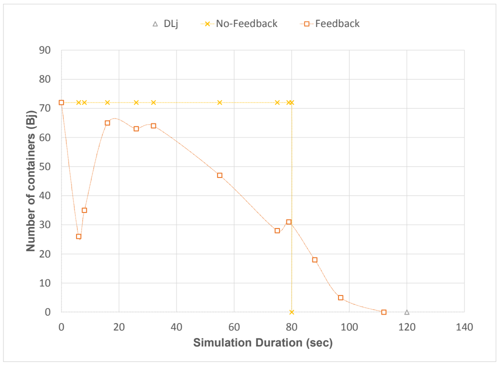
Test 4 – Varying Host Overbooking Ratios: This set of experiments were performed to measure the capabilities of the system to handle multiple parallel requests made to the hosts with varying overbooking ratios, and study their performance while executing the



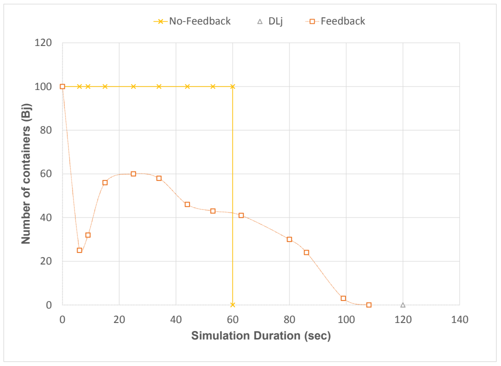
(a) 5 seconds simulation execution time



(b) 10 seconds simulation execution time



(c) 15 second simulation execution time



(d) 20 second simulation execution time

Figure 6: Comparison of Feedback vs No Feedback Approaches

containers. Table 4 shows the results of the experiments where we varied the overbooking ratio from 0.5 to 6. The specified deadline was 4 minutes and expected simulation time was 10 seconds.

We measure the container count and the actual number of hosts acquired by the system to meet the deadline. The system's goal is to minimize this number to keep the economic cost within the bounds. We also measure the simulation duration observed by the system user after the system finds the desired solution, the average turnaround time per simulation from the instant it gets requested till the results get logged, the actual simulation execution time per simulation and the corresponding system overhead. This overhead includes the

Table 4: System Performance with Varying Host Overbooking Ratios

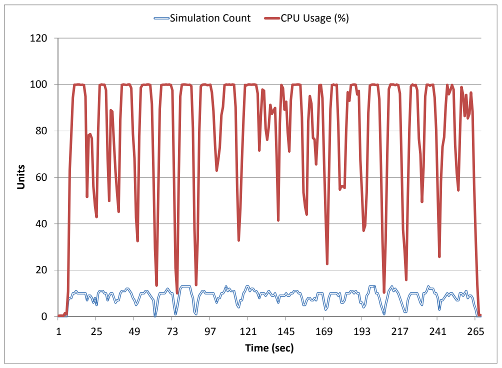
Overbooking Ratio	Max Container Count	Max Hosts Acquired	Simulation Duration (in sec)	Measured Execution Time per Sim(in sec)	Turnaround Time per Sim (in sec)	Measured Overhead(%)
0.5	29	5	239.4	9.31	10.48	12.57
1	41	4	233.6	9.53	11.52	20.88
2	59	3	231.3	15.62	18.59	19.01
4	114	3	213.3	28.72	32.76	14.07
6	160	3	Deadline Missed	41.62	46.9	12.67

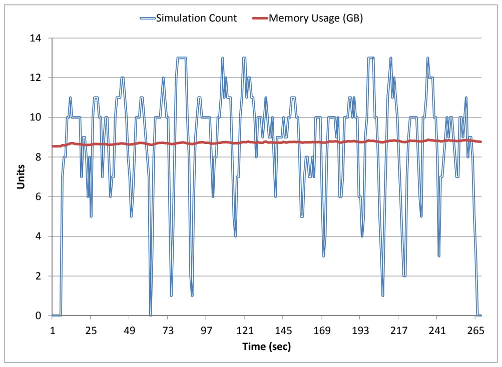
performance interference overhead, resource contention and the time consumed in data transfer at different components of the SIMaaS workflow as shown in Figure 3.

From the results we can conclude that for CPU-intensive applications – simulations tend to fall in this category – the non-overbooked system provides the best results, however, the number of hosts needed is also high, which in turn increases the economic cost. A highly overbooked system too has high cost and will be unable to meet the deadlines due to performance overhead and should be avoided. Based on empirical results, a lower overbooked scenario provides ideal trade-off as it needs less number of hosts and is able to meet the deadlines. We also note that the system overhead remains at a reasonable level of less than 21% during the experiments.

Based on the experiments, we illustrate in Figures 7, the CPU utilization and memory utilization for use case 1. The simulations have a low memory footprint but the CPU utilization is quite high. This conforms to our earlier result that having no or low overbooking for the host will provide better performance. The results were similar for use case 2.

Test 5 – Varying Number of Simulations: The purpose of these tests is to demonstrate the scalability of SIMaaS middleware with increasing number of simulations that are needed as the fidelity of statistical model checking increases. The tests were executed with a deadline





(a) CPU Utilization Variations with Simulation Count

(b) Memory Utilization Variations with Simulation Count

Figure 7: Use Case 1: CPU Utilization Variations with Simulation Count

of 600 seconds while other parameters were kept the same as in previous experiments. Table 5 shows the results, which illustrates that the system is able to scale to 5,000 simulations for a job without significant overhead.

Table 5: System Performance with Varying Number of Simulations

Number of Simulations	Max Container Count	Max Hosts Acquired	Simulation Duration (in secs)	Measured Execution Time per Sim (in sec)	Turnaround Time per Sim (in ms)	Measured Overhead(%)
500	18	1	513.8	4.81	7.79	61.95
1000	33	2	547.1	5.26	11.45	117.68
2500	71	3	588.5	5.52	11.02	99.64
5000	137	6	591.4	5.49	10.72	95.26

Test 6 - Varying Simulation Execution Time: For these experiments, we vary the sampling rate parameter of use case 1 and use a deadline of 10 minutes to increase the simulation execution time of our simulation model. Table 6 measures and presents the simulation performance for varying execution time. We observe that the overhead reduces significantly as the duration of the container execution increases, which is attributed mainly to less percentage of time spent in scheduling and start up of containers.

Table 6: System Performance with Varying Simulation Duration

Sampling Rate (in ms)	Max Container Count	Max Hosts Acquired	Simulation Duration (in secs)	Measured Execution Time per Sim (in sec)	Turnaround Time per Sim (in ms)	Measured Overhead(%)
10	6	1	526.3	4.69	5.87	25.91
5	11	1	533.0	9.35	10.48	12.09
1	108	9	526.0	63.91	66.81	4.53

Test 7 – Scalability Test and Admission Control with Incoming Workload: These are scalability experiments with the workload described in Section II.5.3 with 60 docker hosts and an incoming request flow generated using Poisson distribution. The system applied admission control and informed the users about the decision to accept the request. Table 7 summarizes the results for the tests.

Table 7: Scalability Test Summary

Number of Jobs	103
Test Duration	1h:47min:57s
Number of Simulations Performed	15873
Hosts Utilized	54 / 60
Jobs Rejected	1
Number of Jobs that Missed Deadline	1

We observed that one job with deadline of 618 seconds missed it by 1.39 seconds. This failure can be eliminated with stricter error estimation parameter (α), explained in Section II.4.1. SIMaaS scaled to 54 virtualized hosts during the experiments. In future, we would like to experiment with a larger cluster to test the system's scalability limit.

II.6 Concluding Remarks

This chapter described the design and empirical validation of a cloud middleware solution to support the notion of simulation-as-a-service. Our solution is applicable to those systems whose models are stochastic and require a potentially large number of simulation runs to arrive at outcomes that are within statistically relevant confidence intervals, or systems whose models result in different outcomes for different parameters.

Many insights were gained during this research as follows and resolving these form the dimensions of our future investigations:

- Several competing alternatives are available to realize different aspects of cloud hosting. Effective application of software engineering design patterns is necessary to realize the architecture for cloud-based middleware solutions so that individual technologies can be swapped with alternate choices.

- Our empirical results suggest that an overbooking ratio of 2 and α value of 1.5 provided the best configuration to execute the simulations. However, these conclusions were based on the existing use cases and the small size of our private data center. Moreover, no background traffic was considered. Our future work will explore this dimension of the work as well as determine a mathematical bound for the optimal configuration.

- In our approach the number of simulations to execute for stochastic model checking were based on published results for the use case. In future there will be a need to determine these quantities through modeling and empirical means.

- We have handled basic failures in our system where a container is scheduled again if it does not start, however we need advanced fault tolerance mechanism to handle failure of hosts and various SIMaaS components. Our prior work [14] has explored the use of VM-based fault tolerance, however, for the current work we will need container-based fault tolerance mechanisms.

- We did not consider a billing model for the end users in our work but such a consideration should be given to generate revenues for such a service and also so that the user does not abuse the system. In addition, security and vulnerabilities [103, 178] remain open challenges.

- As most of the cloud providers are rolling out Linux container-based application deployment, we need to design a hybrid cloud to leverage it.

- Currently our middleware architecture is realized as a centralized deployment in our small-scale private data center. In large data centers, we will require a distributed realization of the various entities of our middleware. This will give rise to a number of distributed systems issues, and addressing these form the dimensions of our future work.

All the scripts and source code, and experimental results of SIMaaS are available for download from `http://www.dre.vanderbilt.edu/~sshekhar/download/SIMaaS`.

CHAPTER III

INDICES: EXPLOITING EDGE RESOURCES FOR PERFORMANCE-AWARE CLOUD-HOSTED SERVICES

III.1 Motivation

The cloud has become an attractive hosting platform for a variety of interactive and soft real-time applications, such as cloud gaming, cognitive assistance, health monitoring systems and collaborative learning due to its elastic properties and cost benefits. Despite these substantial advantages, the response time considerations of users mandate lower latencies for the applications. Prior works [82, 93] have shown that in highly interactive applications, latencies exceeding 100 milliseconds (ms) may be too high for acceptable user experience. However, real-world experiments have shown that the latencies experienced by geographically distributed users of an interactive service may tend to be on the order of several hundreds of milliseconds [169]. Consequently, there is a need to bound the resulting response times within acceptable limits.

To better understand the key contributing factors for end-to-end latencies experience by cloud-hosted interactive applications, consider Equation III.1:

$$t_{total} = t_{client} + t_{access} + t_{transit} + t_{datacenter} + t_{server} \tag{III.1}$$

where,

- t_{total} is the end to end latency experienced by the user.
- t_{client} is the processing delay at the client endpoint.
- t_{access} is the sum of inbound and outbound message transmission delays between the client and its nearest network access point.

- $t_{transit}$ is the sum of inbound and outbound communication delays between the network access point and the cloud data center.

- $t_{datacenter}$ is the communication delay from the data center front end (e.g., a web server and load balancer) to the target server in the data center that actually handle the request in both directions.

- t_{server} is the processing delay at the target server.

Both t_{client} and t_{access} cannot be controlled and managed by the cloud service provider. Moreover, since $t_{datacenter}$ is usually less than 1 ms [43], it can be ignored. On the other hand, a cloud provider can control and manage $t_{transit}$ and t_{server}, both of which are key factors in meeting the response time requirements of the interactive applications. Note that $t_{transit}$ is governed by the number of hops incurred by application messages to traverse the wide area network to reach the cloud data center and for responses to traverse back to the user.

In recent years, edge computing, cloudlets [142] or Micro Data Centers (MDCs) [17] have emerged as one of the key mechanisms to manage and bound the transit latency $t_{transit}$ by supporting cloud-based services closer to the clients. MDCs can be viewed as "a data center in a box," which act as the middle tier in the emerging "mobile device–MDC–cloud" hierarchy [142]. MDCs possess key attributes of soft states, sufficient compute power and connectivity, and proximity to clients, and conform to standard cloud technologies.

Recent efforts [44, 46, 94, 189] have leveraged the cloud, MDCs and mobile ad-hoc networks by focusing primarily on cyber foraging, where tasks are offloaded from mobile devices to the cloud/MDCs for faster execution and conserve resources on the mobile client endpoints. Nonetheless, less efforts have focused on moving tasks from the central clouds to the MDCs. Those that do, however, have seldom considered the resulting application performance because these efforts tend to overlook the fact that servers within the MDC may themselves get overloaded, thereby worsening the user experience as compared to that of a traditional cloud-hosted interactive service. Efforts that consider performance of MDCs, however, make very simplistic assumptions regarding their performance models.

In this chapter, our focus is on performance of MDCs, specifically the key contributing factors in performance degradation of MDCs and data centers in general. A fundamental system property that is often overlooked is performance interference, which is caused by co-located applications in virtualized data centers [35, 49, 84, 92]. Performance interference being an inherent property of any virtualized system, it manifests itself in MDCs also and therefore must be factored in any approach that is performance-aware. Thus, we focus on a "just-in-time and performance-aware" service migration approach for moving cloud-based interactive services hosted in the centralized cloud data center to a MDC.

A number of challenges are incurred in supporting such a vision as follows:

- *Hardware heterogeneity:* Differences in the hardware configurations of the servers in a traditional data center versus a MDC will necessarily provide different performance profiles, and hence should be accounted for in any analyses.

- *Performance Interference:* Noisy neighbors cause performance issues, which is an issue that must be considered in both the traditional data centers as well as MDCs. However, since a MDC is orders of magnitude smaller than a traditional data center, the performance interference may be more pronounced and manifest more rapidly in MDCs than in traditional data centers.

- *Network performance measurements:* Accurate latency and bandwidth measurement techniques are required that can reliably work over Wide Area Networks (WANs). This dimension of the challenge is important since estimating an accurate value for $t_{transit}$ is important in our problem formulation and its solution.

- *System performance measurements:* Accurate application and server performance measurement and logging techniques are required to accurately measure the service execution (i.e., t_{server}) on the hardware of the data centers or MDCs.

We address these challenges in the context of providing a ubiquitous deployment approach that spans the cloud-edge spectrum and make the following contributions:

- We present a technique to estimate the performance of a cloud application on different hardware platforms subjected to performance interference stemming from various co-located applications.
- We formulate server selection as an optimization problem that finds an apt server among micro data centers to migrate an application to, so it can meet its performance needs while minimizing the deployment cost to the service provider.
- We describe the INDICES (**IN**telligent **D**eployment for ub**I**quitous **C**loud and **E**dge **S**ervices) framework that codifies our algorithms for online performance monitoring, performance prediction, network performance measurements, server selection and application migration.
- We show experimental results to validate our claims and evaluate the efficacy of the INDICES framework.

The rest of the chapter is organized as follows: Section III.2 presents the system model and assumptions; Section III.3 describes the problem formulation we address in this research; Section III.4 delves into details of our solution including the design and implementation; Section III.5 presents empirical proof that validates our claims; Section III.6 compares related work with our work; and finally Section III.7 presents concluding remarks alluding to lessons learned and future work.

III.2 System Model and Assumptions

This research is geared towards platform-as-a-service (PaaS) cloud providers, who seek to meet service level objectives (SLOs) of soft real-time applications such as online gaming, augmented reality, virtual desktop etc. by improving application response times. To that end they exploit micro data centers. In doing so, however, cost considerations and energy savings for the PaaS provider in operating and managing the resources beyond the traditional data centers are critical issues while ensuring that such an approach provides an additional source of revenue to the PaaS provider. In this chapter, however, we do not discuss revenue generation issues.

III.2.1 Architectural Model

Figure 8 depicts our architectural model that consists of a centralized data center CDC, owned by a PaaS cloud provider. The CDC is connected to a group of micro data centers (MDCs), $M = \{m_1, m_2, .., m_n\}$. These MDCs are deployed at the edge, and are either owned by the CDC provider or leased from an edge-based third party MDC provider. A leased MDC is assumed to be exclusively under the control of the that CDC provider. Once a MDC is leased, all its resources are considered to be the part of CDC provider and hence customers of the CDC can be transparently diverted to MDCs using their CDC-based security credentials.

The CDC contains a global manager gm, which is responsible for detecting and mitigating global SLO violations. We assume that for all $m \in M$, there exist links to the CDC with a backhaul bandwidth of b_m. Each MDC m comprises a set of compute servers, H_m, that can be allocated to the CDC for its operations at a specified cost. One of the hosts from H_m or a specially designated MDC host acts as the local manager (lm_m) for that MDC and is responsible for data collection, performance estimation, latency measurements and MDC-level decision making. This decision-making logic is deployed at the MDC by the CDC provider.

III.2.2 Application Model

For this work, we consider a set $Apps$ of latency-sensitive applications that can be collaborative or single user and interactive or streaming in nature. Each application $a \in Apps$

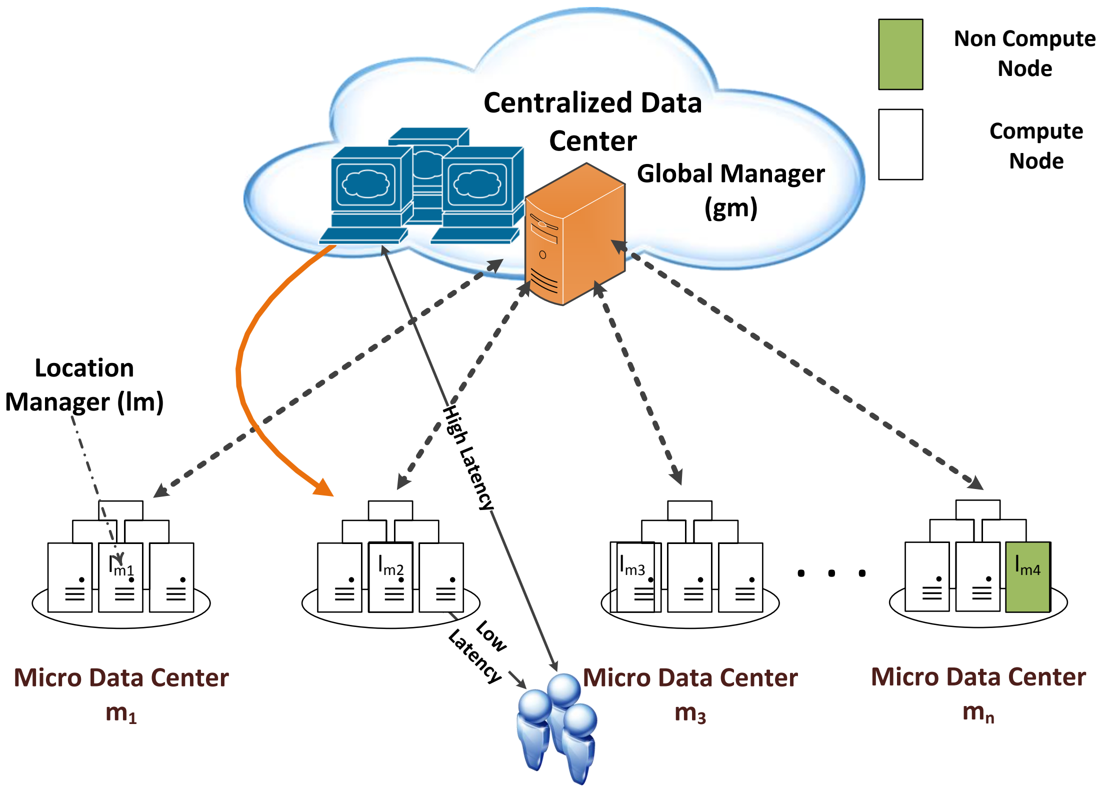


Figure 8: Architectural Model

is initially deployed in a CDC, with U_a number of users and is assumed to be containerized inside a virtual machine (VM). We assume that for a collaborative application a, its users are located in proximity of each other where they incur similar round trip latencies. These scenarios are common when we consider collaborative educational applications such as [34] where the users are a group of students working from a school library or a coffee shop, or could be a single user system, such as augmented reality [69], where image processing operations are performed in the cloud.

Each application a can be hosted on any active host in CDC, $\eta \in H$, where H is the set of all active hosts that provide virtualization using a hypervisor or virtual machine monitor (VMM), such as KVM [90] and Xen [7]. We let eed_a represent the expected execution duration for which the application will be used by the end-user clients. An interactive or streaming application comprises multiple individual interactions between the user and the application. Each interactive or streaming step of a is assumed to take an estimated execution time $eet_{a,\eta}$ on host η; for collaborative applications, it indicates the time needed for all users to have completed that step. Section III.4.2 discusses in detail a systematic way of estimating these per-step execution times. Finally, for all users $u \in U_a$, let $el_{a,\eta,u}$ represent the estimated round-trip network latency and ϕ_a be the application-defined bounds on acceptable response time for each interactive step of the application. Formally, the SLO for each application a hosted on host η can be characterized by:

$$eet_{a,\eta} + \max_{u \in U_a}(el_{a,\eta,u}) \leq \phi_a \tag{III.2}$$

Over time, a subset PA from the set of applications $Apps$ are identified by the system as suffering from performance degradation such that each application $p \in PA$ has a subset of one or more users, $U'_p \subseteq U_p$ experiencing SLO violations. These impacted applications can be identified reactively either by the end-user client, which notices missed deadlines using special instrumentation features supplied in the client-side "app" that is installed by the end-user as part of the PaaS platform and notified to the CDC service. Alternately, such applications can be be identified proactively via a predictive decision based on the existing user profiles, where the system predicts that the users are likely to experience SLO violations if they had connected from their profiled location during a certain time period.

Our objective is thus to minimize the SLO violations, which is achieved by identifying and migrating application p to a MDC host $h \in H_m$ that will provide significantly improved performance. Since any application migration will involve state transfer, we assume that application p has the snapshot of current state which has to be transferred as part of the migration over the backhaul network from CDC to MDC m. Moreover, $ci_{p,h}$ is the initialization cost of the migrated application p on host h_m before the application can start processing requests on the MDC host. However, once the user-specific state has been transferred, there is minimal interaction between the CDC-based server and the MDC-based server for the remainder of the functioning of application p. For this chapter we do not consider further consolidation of resources where applications migrate back to the CDC. The transfer cost $transfer_{p,h}$ incurred while transferring application p from CDC to host h of a MDC, and associated constraint are defined in the following equations:

$$transfer_{p,h} = \frac{s_p}{b_m} + ci_{p,h} \tag{III.3}$$

$$transfer_{p,h} \ll eed_p \tag{III.4}$$

where, b_m is the backhaul bandwidth, s_p is the size of the snapshot of the application's state, and eed_p is the remaining expected execution duration of application p's usage by the client. Equation III.4 is a necessary condition for the motivation to use the edge and our solution to be relevant. To ensure that Equation III.4 holds, we do not require transferring entire images of the VM and its containers. Instead, we use a layered file system architecture at the MDC that is pre-populated with base images used at the CDC as described in Section III.4.4. This assumption is realistic because we surmise that a MDC is either owned entirely or leased exclusively by a CDC provider. We also ensure Equation III.4 holds by considering δ_p as a tolerance percentage value for the application user before (s)he starts to observe the improved response time:

$$transfer_{p,h}/eed_p \leq \delta_p \tag{III.5}$$

Finally, another critical issue we must account for is that any migration of a new application from CDC to a MDC should not violate the SLOs of existing applications in that MDC. To capture this aspect, let J_h represent the set of all applications currently running on a MDC host h, $eet_{j,h}$ be the estimated execution time for each application $j \in J_h$, which must be updated when we make a decision to migrate p to the same host, and $el_{j,h,u}$ be their corresponding measured round-trip network latency. These quantities must satisfy:

$$\forall j \in J_h, eet_{j,h} + \max_{u \in U_j}(el_{j,h,u}) \leq \phi_j \tag{III.6}$$

III.3 Problem Statement and its Formulation

We now formally present the problem statement. Recall that our objective is to improve response times for cloud-hosted interactive applications that are experiencing performance degradation by migrating the application to the edge-based MDC. To that end, we must address two key problems. First, we must have a systematic approach to understand the causes of performance degradation and determine if an application is being impacted. Second, we must find an effective approach by which an application can be migrated from a CDC to a MDC without impacting existing MDC-based applications while also minimizing the cost incurred by the cloud provider.

III.3.1 Performance Estimation Problem and Challenges

The performance of an application depends on several factors including the workload, the hardware hosting platform, and co-located applications that cause performance interference [35, 49, 84]. It is thus important for any solution to account for all these dimensions for accurately estimating performance for both CDCs and MDCs. Below we describe their role in the performance estimation problem:

III.3.1.1 Workload Estimation

For the cloud-hosted interactive applications of interest to us, we assume that the workload variation is not significant within a single user session with the service. However, different sessions may have different workloads, for example, in an image processing application, the quality and hence the size of the captured and relayed image may vary for different client mobile devices. Thus, we consider each workload as a different application setting, which is reflected in the application response time.

III.3.1.2 Heterogeneity

Our CDC and MDCs consist of heterogeneous hardware and hence each application's performance can vary significantly from one hardware platform to another [49]. Therefore, we need an accurate benchmark of performance for each hardware platform.

III.3.1.3 Performance Interference

Server virtualization platforms such as KVM [90] and Xen [7] provide high degree of security, fault and environment isolations for applications running in virtualized containers, i.e. virtual machines (VMs). However, the level of isolation is inadequate when it comes to performance isolation even though the cloud providers have well-defined resource sharing mechanisms. This happens due to to two primary reasons:

- *Presence of non-partitionable shared resources:* VMMs can provide isolation guarantees by applying strict CPU reservations and static partitioning of disk and memory spaces. There are solutions available to limit the storage and network bandwidth too. Yet, on-chip resources including cache spaces, cache and DRAM bandwidths, and interconnect networks are difficult to partition [66]. Recently, Intel has introduced Cache Allocation Technology [32] to partition the last level cache (LLC), however, it is still not widely used and cannot be applied to older generation servers. The load imposed on these shared resources by one application is detrimental to all the cache- and memory-sensitive applications [116].

- *Resource overbooking:* The average server utilization in a data center is usually low ranging from 10% to 50% [110]. Thus, to maximize the server utilization, cloud providers tend to overbook resources such as CPU cores. This precludes strict CPU reservations and leads to even the lower level caches (L1 and L2) getting shared. In addition, if the overbooked workload goes beyond the server capacity, contention takes place and the applications suffer from performance issues.

III.3.2 Cost Estimation and Objective Formulation

The objective of the framework is to assure the SLOs for all the identified applications $p \in PA$ while minimizing the overall deployment cost. Each MDC host h involves a monetary allocation cost as it is either leased or could be leased to other providers if owned by the centralized cloud. In addition, the running servers have operational costs, such as need for power and cooling. Thus, the provider wants to use as few MDC servers as possible and hence the deployment cost depends on the duration for which the MDC server is on. This cost $\widetilde{T_h}$ for deploying $p \in PA_h$ applications on host h is the extra duration for which the server has to be turned on and can be represented as:

$$\widetilde{T_h} = \begin{cases} 0, & \text{if } \max\limits_{p \in PA_h}(eed_p) < \max\limits_{j \in J_h}(eed_j), \\ \max\limits_{p \in PA_h}(eed_p) - \max\limits_{j \in J_h}(eed_j), & \text{otherwise} \end{cases} \tag{III.7}$$

We define a constant α_h denoting the cost of powering on the MDC server, and constant β_h denoting the cost for transferring the state to host h. Their values depend on the host h and its corresponding MDC. The cost for deployment on host h is thus defined as:

$$C(h) = \alpha_h * \widetilde{T_h} + \beta_h * \sum_{p \in PA_h} transfer_{p,h} \tag{III.8}$$

The optimization problem we solve for this research can then be formulated as:

$$\begin{aligned} \underset{h \in H}{\text{minimize}} \sum C(h) & \\ \text{subject to} \quad & eet_{p,h} + \max_{u \in U_p}(el_{p,h,u}) \leq \phi_p, \\ & \forall j \in J_h, eet_{j,h} + \max_{u \in U_j}(el_{j,h,u}), \\ & transfer_{p,h}/eed_p \leq \delta_p \end{aligned} \tag{III.9}$$

III.4 Design of INDICES

We now present the design of our INDICES framework, which solves the optimization problem from Equation III.9. To that end our solution depends on accurately and reliably estimating (a) the execution time of the impacted application and network latencies suffered by its clients, (b) similar parameters for the already running applications on the different hosts of the different MDCs, which is then used in selecting the appropriate host to migrate an impacted application to, and (c) the transfer time for migrating the state of the impacted application. The remainder of this section describes our framework architecture and the details of the techniques used to solve the optimization problem.

III.4.1 INDICES Architecture and Implementation

Before delving into the details of the techniques used to solve the optimization problem, we first present a high-level architecture of INDICES. Given the scale of the system, a centralized approach to performance prediction and cost estimation for every application hosted in the CDC/MDC and its clients is infeasible. Thus, we take a hierarchical approach where individual MDCs with their local managers and the global manager of the CDC participate in a two-level decision making as shown in Figure 8.

Figure 9 shows the local decision making part of INDICES. Each MDC is composed of a management node and several servers on which the applications residing on virtual machines execute. Each individual host in the system has a performance monitoring component that logs the data at the local manager lm_m. The local manager consists of a data collector, latency estimator, performance predictor and cost estimator.

The performance monitor instruments the host and collects system level metrics such as CPU, memory and network utilizations, as well as micro architectural metrics such as retired instructions per second (IPS) and cache misses. This information is periodically logged to the local manager for processing. The performance monitoring framework is based on the collectd [59] system performance statistics collection tool. To collect micro

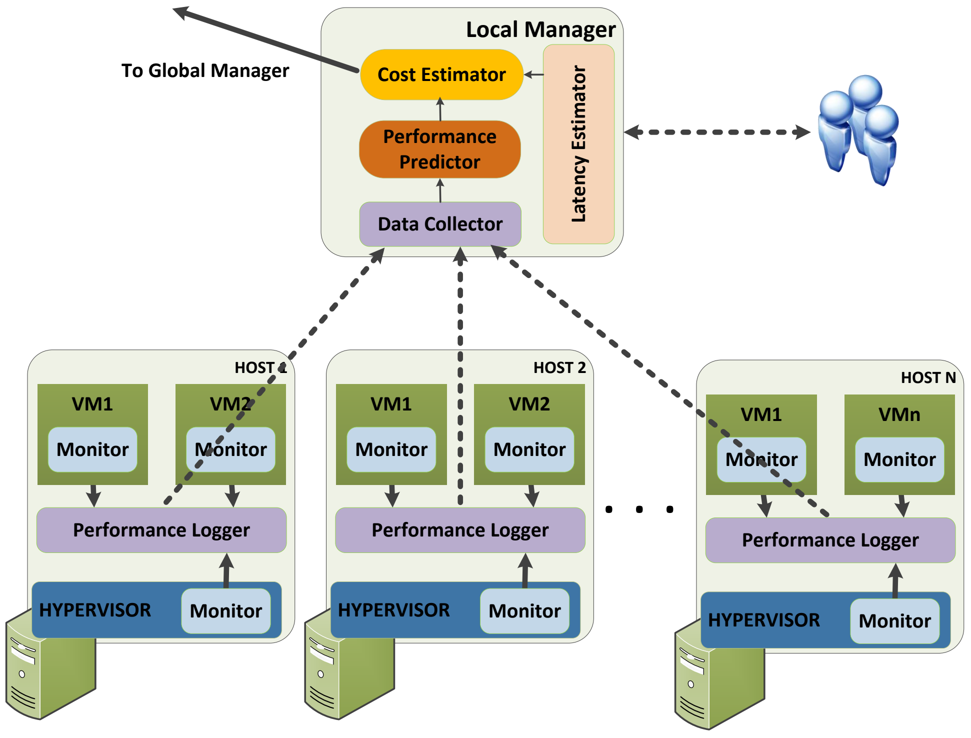


Figure 9: Local Decision

architectural performance metrics, we developed a python plugin for collectd using Linux perf. This plugin detects if the hardware platform is known, and accordingly executes code that collects hardware specific performance counter statistics. The information is then forwarded to the lm_m using AMQP [167] message queuing protocol. The lm_m runs a server developed in the Go programming language, which persists the data in the InfluxDB database, which is designed specifically for time series data.

III.4.2 Estimating Execution Time

The constraints in Equation III.9 require an accurate understanding of the predicted execution time duration of an application if it were to execute at a MDC, as well as the execution times of the existing applications executing on the hosts of the MDCs. Hence, we build an application's expected performance profile and in turn its interference profile [81] when co-located with other applications on different hardware platforms given the hardware heterogeneity across the CDC and MDCs. Although prior efforts [116, 180, 187] have used retired instructions per cycle (IPC) or last-level cache (LLC) miss rate as the performance indicators, Lo et. al [110] have shown the limitations of these metrics for latency-sensitive applications. Thus, we consider *execution time* as the primary indicator of performance.

The *Interference profile* of an application [95, 116, 176, 188] is a property that identifies the degree to which that application will (a) degrade the performance of other running applications on the host – known as *pressure* – and (b) how much its own performance suffers due to interference from other applications – known as *sensitivity*. The performance degradation of an application depends, to varying degrees, on different system components and architectures, and other collocated applications. Several prior efforts have used pairwise application execution to estimate their sensitivity and pressure [95, 116, 176, 188], however, these solutions are not viable for a data center given the significantly large number of hosted applications. Some other efforts [180] pause non-critical applications to measure pressure and sensitivity of live applications, which may not be a realistic solution.

Thus, for a given application p, its performance on a host with hardware configuration w is modeled by Equation III.10, where Y is the execution time, X is a vector of system-level metrics that quantify the state of the host, and the function $f_p^1()$ models the relation between the state of the host machine and performance of the application p. Moreover, the information needed by the second constraint of Equation III.9 is obtained through Equation III.11, which depicts the change in the state of the host with hardware configuration w if application p were to be hosted on it. Equation III.11 is an indirect measure of performance interference since its output can be used to calculate the change in execution time of an already running application by plugging the new state vector X^{new} into Equation III.10 and solving it for each running application.

$$Y = f_p^1(X_w) \tag{III.10}$$

$$X_w^{new} = f_p^2(X_w^{old}) \tag{III.11}$$

Table 8: Server Architectures

Config	Hardware Model	sockets/cores/ threads/GHz	L1/L2/L3 Cache (KB)	Mem Type/ MHz/GB	Memory Bandwidth	Count
A	i7 870	1/4/2/2.93	32/256/8192	DDR3/ 1333/16	(UNC_IMC_NORMAL_READS.ANY + UNC_IMC_WRITES.FULL.ANY) * 64 / time in sec	2
B	Xeon W3530	1/4/2/2.8	32/256/8192	DDR3/ 1333/6	(UNC_IMC_NORMAL_READS.ANY + UNC_IMC_WRITES.FULL.ANY) * 64 / time in sec	1
C	Core2Duo Q9550	1/4/1/2.83	32/6144/-	DDR2/ 800/8	BUS_TRANS_MEM.ALL_AGENTS * 64 *1e9 * CPUFrequency / CPU_CLK_UNHALTED.CORE	1
D	Opteron 4170HE	2/6/1/2.1	64/512/5118	DDR3/ 1333/32	SamplingPeriod * DRAM_ACCESSES_PAGE.ALL * 64 / time in sec	9

Another required step is to identify the right system level metrics to use. Previous works [48, 49, 81] have identified several sources of interference including caches, prefetchers, memory, network, disk, translation lookaside buffers (TLBs), and integer and floating point processing units. Both Intel and AMD architectures provide hardware counters to monitor the performance of micro-architectural components. However, not all the sub-components can always be monitored. Moreover, the list of available counters is significantly smaller for older generation servers. Due to these constraints and driven by the need to support a broadly applicable solution, we selected the following host metrics for performance monitoring:

- **System Metrics:** CPU utilization, memory utilization, network I/O, disk I/O, context switches, page faults.
- **Hardware Counters:** Retired instructions per second (IPS), cache utilization, cache

misses, last-level cache (LLC) bandwidth and memory bandwidth. The bandwidth metrics are not directly available and the counters are vary from one hardware to other. In our analysis described later, we found that the LLC bandwidth and memory bandwidth were highly correlated and hence we selected the memory bandwidth and not LLC bandwidth due to its easier availability on different architectures and versions. Table 10 lists the hardware counter-based equations for memory bandwidth, which are derived from [1, 55].

- **Hypervisor metrics:** Scheduler wait time, Scheduler I/O wait time, scheduler VM exits. These metrics are the summation for all the executing virtual machines for the KVM hypervisor.

By applying standard supervised machine learning techniques on the collected metrics, we estimate the functions in Equations III.10 and III.11 using the following sequence of steps:

1. **Feature Selection:** Feature selection is the process of finding relevant features in order to shorten the training times and reduce errors due to over-fitting. We have adopted the Recursive Feature Elimination (RFE) approach using Gradient Boosted Regression Trees [56]. We performed RFE in a cross-validation loop to find the optimal number of features that minimizes a loss function (mean squared in our case).

2. **Correlation Analysis:** To remove the linearly dependent features, correlation analysis is required. This step further reduces the training time by decreasing the dimensions of the feature vector. We used the Pearson Coefficient to eliminate highly dependent metrics with a threshold of ± 0.8.

3. **Regression Analysis:** In this step curve fitting is performed using ensemble methods. We have used standard off-the-shelf Gradient Tree Boosting method, which is widely used in the areas of web page ranking and ecology. The primary advantage

of this method lies in its ability to handle heterogeneous features and its robustness to outliers.

The performance estimation of applications consists of two phases: (1) Offline Phase and (2) Online Phase. The offline phase occurs at CDC and concerns with finding estimators whereas the online phase is performed by the local manager (lm_m) of MDCs to estimate the performance of the target application and also to estimate the performance degradation of the running application. The two phases are described next.

III.4.2.1 Offline Phase

Whenever the data center receives a request for migrating an as yet un-profiled application, it is benchmarked on a single host with a given hardware configuration and then co-located with other applications to develop its interference profile. However, since the number of profiling configurations can be huge, we select a uniformly distributed subset of possible co-location combinations for profiling. The estimators can be found either by following the above listed three steps or choosing an existing estimator of some application based on similarity between the projected performance and the actual performance. We use a hybrid approach, which first predicts the performance of the new application and its interference profile using estimators of an existing application for the same hardware specifications. If the measured performance and the estimated performance are within a predefined threshold, then we consider the new application to be similar in performance to the existing application. Among all such similar applications, the estimator of the application with least error is selected for all MDC hardware configurations. This saves profiling time and cost. However, if there is no match, the application profile is developed by performing feature pruning followed by model fitting on each unique hardware platform maintained by the data center.

III.4.2.2 Online Phase

The learned models are then exported and forwarded to the MDC local manager lm_m for the available hardware platforms in the MDC for estimating the performance of any application to be deployed in the MDC. Since each MDC is small in size and typically illustrates limited heterogeneity in the supported hardware, the number of estimation models will be small. On receiving a request from the global manager gm, the local manager lm_m estimates the performance of an application by feeding the estimator with presently logged data set using estimation function III.10. The pressure on existing applications J_h on the host h is calculated by first applying Equation III.11 on the target application and then Equation III.10 for existing applications.

III.4.3 Network Latency Estimation

The constraints of the optimization problem of Equation III.9 require an accurate understanding of the network latencies incurred by the clients, specifically the worst among all the clients of each application. This information is needed in identifying the appropriate host of the appropriate MDC to which an impacted application can be migrated to such that it satisfies the SLOs for the worst suffering client while not unduly affecting existing applications of the MDC hosts.

Thus, estimating the latency to different MDC servers is another key component for achieving the targeted SLOs. To that end we must determine the clients who suffer SLO violations from Equation III.2. In each client, the instrumented "app" that is installed by the user as part of the client application periodically reports to gm the application response time it is observing. To not overwhelm the gm, such data logging need not occur directly on the gm; instead it can be logged on an ensemble of servers that then report to the gm or the application server can itself gather data and forward the information when SLO violations occur.

Since there could be multiple MDC choices to migrate an impacted application to, the first step in our algorithm for server selection requires reducing the target set of MDCs for latency estimation to decrease the load and amount of time for server selection. To that end, we use the logged performance data from the clients to extract its IP address in order to determine the closest MDCs to that client. The extracted client IP address may not be accurate since often internet users have private addresses and the reported external address is that of the network router or one from the pool of network provider's addresses in case the connection is via a cellular network. However, this information is still sufficient for us as we use the client location to reduce the set of MDCs that we need to query. The client's geolocation and consequently its region is derived from the IP address.

The next step is measuring latencies to the nearby MDCs. To obtain a reliable latency estimate, we use HTTP-based and TCP socket-based latency measurement techniques for HTTP-based and plain TCP-based cloud applications, respectively. We can easily add additional protocols to this list based on the protocol used. Subject to the collected information, the *gm* forwards to the client app a list of "nearby" MDC gateway servers that are also the local managers lm_m, each hosting a server for the purpose of latency measurement. The client then posts n requests to each lm_m with a file that it typically posts to the cloud for processing (e.g. an image for image processing application) and also the average size of the response it receives from the application. The server responds with a response for the same number of bytes. For each of the n interactions, the client records the elapsed time and thus measures $t_{client} + t_{access} + t_{transit}$. The client app selects the SLO latency (e.g., usually 95^{th} percentile) from the n latencies for each lm_m and reports it to *gm*. This approach also accounts for the delay due to bandwidth size as we transfer the actual request data instead of a ping. This step can be considered as similar to the *speed test* done to measure download/upload speeds to an internet provider.

III.4.4 State Transfer

The final constraint of Equation III.9 requires estimating the cost of state transfer. The local managers calculate the state transfer cost using Equation IV.5 and use it in local decision making. Once the *gm* selects the h_m for migrating an application p, the application state has to be transferred before the clients can be switched to the new server location. In this regard, there exist several solutions available for WAN-scale virtual machine migration [31, 142, 163, 174]. We leverage the cloud virtual disk format such as qcow2 features for WAN migration. The VM disk is composed of a base image and can contain several overlays on top of it for change sets. The VM overlay when combined with the base image constructs the VM that needs to run for serving the clients.

This base image can contain just an operating system such as Ubuntu or an entire software stack such OpenCV for image processing. The base image is assumed to be present on MDC hosts to save on migration costs and can be shared by multiple VMs. For the target application, overlays are created using external snapshots. The VM overlay is the state that gets transferred to h_m and is synthesized with the base image for execution. Equation IV.5 displays the cost.

Once the application starts running, it informs the *gm* and all the application clients are redirected to the new application URL. This happens for a custom client by forwarding the new location to the clients which can then use the new URL for processing. However, for browser-based clients, the communication with the gm occurs via application server due to cross-domain restriction and the existing application issues HTTP-redirect to the new location. In future, we will enhance our solution to support live migration of VMs using solutions, such as Elijah cloudlet [68] or the recently introduced Docker Linux container's [121] live migration feature.

III.4.5 Solving the Optimization Problem at Runtime

The final piece of the puzzle is solving the optimization problem in Equation III.9. The optimization problem described in III.3.2 cannot be solved offline due to the changing dynamics of the system, and being an NP-Hard problem when we consider multiple applications violating SLOs and need to be migrated. We employ a heuristics-based algorithm described in Algorithm 3 that selects aptly suited servers in a MDC while minimizing the overall deployment cost for the entire system.

Algorithm 3 Deployment Server Selection Algorithm

```
Require: Apps
 1: for all $a \in Apps$ do
 2:     $t_{a,CDC} \leftarrow \max(U_a)$                                  ▷ $t$ is response time
 3:     if $t_{a,CDC} > \phi_a$ then $PA.insert(a)$
 4: if $PA = \emptyset$ then return                                      ▷ Do nothing
 5: for all $p \in PA$ do
 6:     $eed_p \leftarrow GetExpectedExecutionDuration(p)$
 7:     $clientLoc \leftarrow GetLocation(max(U'_p))$
 8:     $nearbyMDCs \leftarrow FindNearbyMDCs(clientLoc)$
 9:     for all $m \in nearbyMDCs$ do
10:         $lm_m \leftarrow LocalManager(m)$
11:         $el_{p,lmTransit} \leftarrow GetLatency(lm_m, clientLoc)$
12:         $H_m \leftarrow GetServerList(m)$
13:         for all $h_m \in H_m$ do
14:             if $transfer_{p,h_m} > \delta_p$ then
15:                 skip $h_m$                                           ▷ Constraint Violated
16:             $perf_{p,h_m} \leftarrow PredictPerfInterf(h_m, p)$
17:             for all $j \in J_{h_m}$ do
18:                 $eet_{j,h_m} \leftarrow EstExecTime(perf_{p,h_m}, j)$
19:                 if $el_{j,h_m} + eet_{j,h_m} > \phi_j$ then
20:                     skip $h_m$                                       ▷ Constraint Violated
21:             $eet_{p,h_m} \leftarrow EstExecTime(perf_{h_m}, p)$
22:             if $el_{p,lmTransit} + eet_{p,h_m} > \phi_p$ then
23:                 skip $h_m$                                           ▷ Constraint Violated
24:             $transfer_{p,h_m} \leftarrow EstTransDur(h_m, p)$
25:             $C_{p,H_m}.insert(EstCost(transfer_{p,h_m}, eed_p))$
26:         $C_{p,m}.insert(min(C_{p,H_m}))$
27:     $minC_{p,h} \leftarrow min(C_{p,m})$
```

The algorithm consists of two phases. First, we identify the applications suffering SLO violations (Line 3). In the second phase we select the suitable server. For the identified applications in PA, we find the location and address of the client that suffers the worst latency (Line 7). There are standard APIs such as the one in Android, getLastKnownLocation that allows to get the last location [104]. That location is used to perform a lookup for nearby MDCs (Line 8). We then identify the server within the identified MDCs that provide the best performance. This step is carried out in parallel across all the identified MDCs (Loop starting at Line 9). The client measures the latency to the local manager lm_m of each nearby MDC and if it is within the acceptable application response time threshold ϕ_p, then we select that MDC and fetch the corresponding list of servers (Line 12).

For each such server, we predict the performance interference and estimate the execution time of the application p were it to execute on that host (Line 16), and update the estimated execution time of existing applications J on that host (Loop starting at Line 17). We then calculate the cost according to Equation III.8 if the constraints defined in Equation III.9 can be met (Line 25). The minimum cost server is identified for each MDC (Line 26). Finally the minimum cost server is selected across all identified MDCs (Line 27) and the application is migrated and clients are redirected to the migrated application. Due to the distributed nature of our framework, the algorithm can be solved in $\mathscr{O}(H_m * J_h)$ for each application p.

III.5 Experimental Validation

We now present results of evaluating INDICES in the context of a latency senstive application use case.

III.5.1 Experimental Setup

Table 10 illustrates the hardware platforms and their counts used in our experiments. The CDC uses Openstack cloud OS version 12.0.2 where the guests receive their own public IP addresses. The MDC servers are managed directly by libvirt virtualization APIs and the guests communicate via port forwarding on the host. Each machine has Ubuntu 14.04.03 64-bit OS, QEMU-KVM hypervisor version 2.3.0 and libvirt version 1.2.16. Guests are configured with 2 GB memory, 10 GB disk, Ubuntu 14.04.03 64-bit OS and either 1 or 2 VCPUs. Since we are not concerned with VM migration within a CDC, we do not depict the CDC heterogeneity.

We use PARSEC and Splash-2 benchmarks [28] to generate the training data. As described in section III.4.2, to preclude profiling every new application on all the hardware, we need some training data. PARSEC targets Chip-Multiprocessors composing virtualized data centers, and provides a rich set of applications with different instruction mix, cache and memory utilization, needed for stressing different system subcomponents. We selected 20 tests from the benchmarks for data generation and validation. Due to lack of access to servers in different geographical regions, we used the network emulation tool, *netem*, and hierarchy token bucket based traffic control, *tc-htb*, for emulating the desired network latencies and bandwidth among the client, CDC and different MDCs.

III.5.2 Application Use Case

We use an image processing application to validate the efficacy of our framework. The application performs feature detection, which is a critical and expensive part of any of the computer vision problem such as object detection, facial recognition [60] etc. We use the well-known Scale Invariant Feature Transform (SIFT) [111] to find the scale and rotation independent features. The client-side interface of the application continuously streams frames from a video or a web camera at a fixed rate of a frame per 200 milliseconds. The video resolution is 640x360 pixels and average frame size is 56 KB. The server comprises a

Python-based application that receives frames over a TCP socket, processes it, and responds with the identified features along with the processing time. The client expects to receive a response within this duration, implying that 200 ms is the deadline for the application. Although our use case considers the performance for a single client connected to the cloud-hosted application, it can easily be extended to multiple clients residing in a similar latency region.

When the image processing application is submitted for hosting in our cloud, we execute it on different hardware platforms in isolation to find its base execution times. For hardware platforms A, B, C, D defined in Table 10, the base execution times, eet_a, were measured to be 86, 91, 146, 157 ms, respectively. Table 9 displays the emulated ping latency el_a from this client to CDC or different MDCs in the same region as the client. The table also lists their server composition, and the measured 95^{th} percentile network latency while sending TCP/IP and HTTP post requests of 56 KB size and receiving a response of size less than 1 KB. The expected duration for which the client needs to perform the image processing, eed_a, was set as 1 hour and the SLO was set to 95%.

Table 9: CDC and MDC set up for use case (Section III.5.2)

Conf	Distance	Ping Latency (±20%) ms	TCP Latency (ms)	HTTP Latency (ms)	Servers
1	1 hop	<1	2	6	1C + 1D
2	2 hops	5	14	28	1A + 2D
3	Multi hops	20	54	96	1B + 2D
4	Multi hops	30	76	142	1A + 3D
5	Central	50	127	220	1D

III.5.3 Evaluating the Performance Estimation Model

We first benchmarked our use case application on hardware platform D in order to develop its performance estimators. The threshold to discern applications with similar interference performance profile, as described in Section III.4.2.1, was set to 10% error. However, as illustrated in Figure 10, none of the existing applications met the criteria. Thus, we decided not to use any of the existing estimators for the use case application and benchmarked the application on all hardware configurations to develop its estimators. Figure 10 confirms that the estimation errors were high for all the hardware types requiring us to develop its estimators. We also found that the mean estimation error for our use case application to be less than 4% on all the platforms with low standard deviations as depicted in Figure 11. We can also account for this estimation error in our response time constraint (Equation III.2) for stricter SLO adherence.

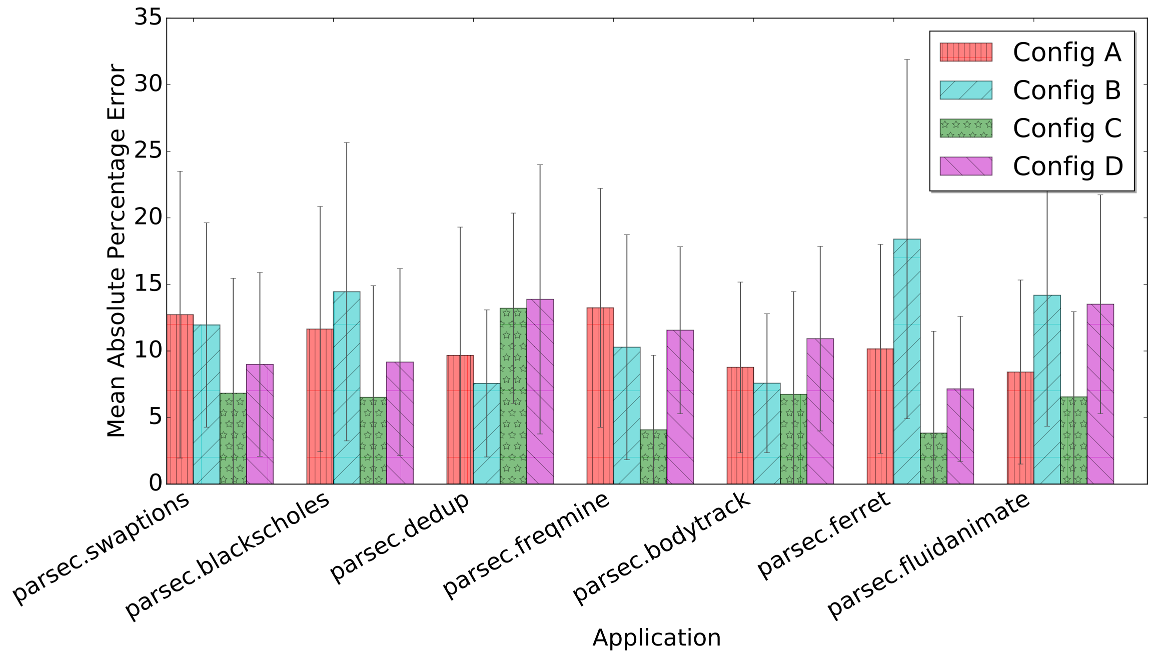


Figure 10: Estimation of SIFT Profile Similarity with Parsec Benchmark

III.5.4 Evaluating the Server Selection Algorithm

We compare our server selection algorithm results against two approaches: server selection algorithms based on minimum number of hops and least loaded server (among

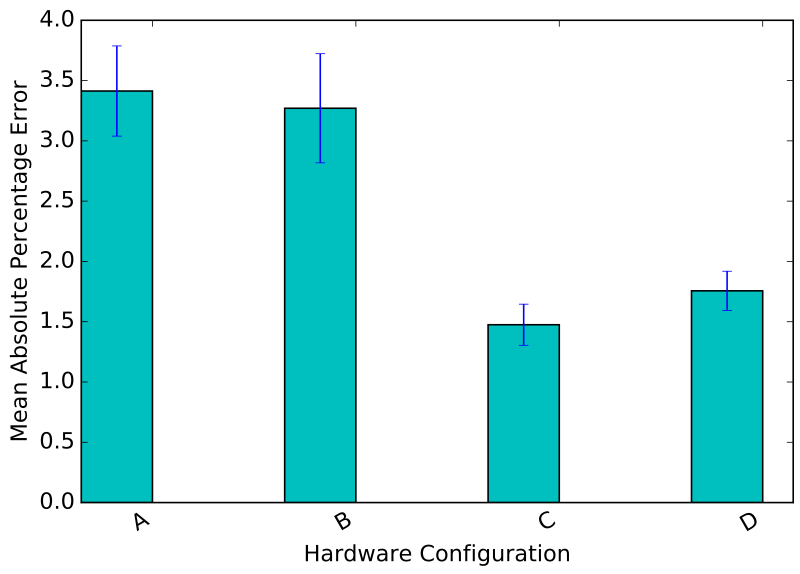


Figure 11: SIFT Application Performance Estimation Error

reachable MDCs). From Table 9 we observe that the minimum hop is 1. There are 2 servers in the minimum hop MDC 1 with hardware configuration types C and D. We create interference load on both the servers but ensured that the total load on the server does not exceed its capacity in terms of memory and vCPUs to eliminate unrealistic performance deteriorations. For the least-loaded server algorithm, we considered the server with least existing allocated resources, i.e. containing only a single VM. We did not consider a server with no existing load as it results in acquiring a new server and thus causes additional cost to the service provider. We found the server of hardware type D with MDC configuration 4 to be least loaded.

Applying SLO from Equation III.2, INDICES found 2 servers of type A and D from MDC 2 and one server of type B from MDC 3 to be suitable for which we plot their response times for eed_a of one hour. Figure 12 displays the comparison of each of the suitable servers found by INDICES against the least loaded server. We observe that in this scenario, the least loaded server had 100% SLO violation because of network latency. However, the servers found by INDICES met their deadline 100%, 99.38% and 98.94%, respectively, which was well over the target SLO of 95%. Also, the minimum hop servers met the deadline only 66.64% and 60.64% of times due to performance interference shown in Figure 13.

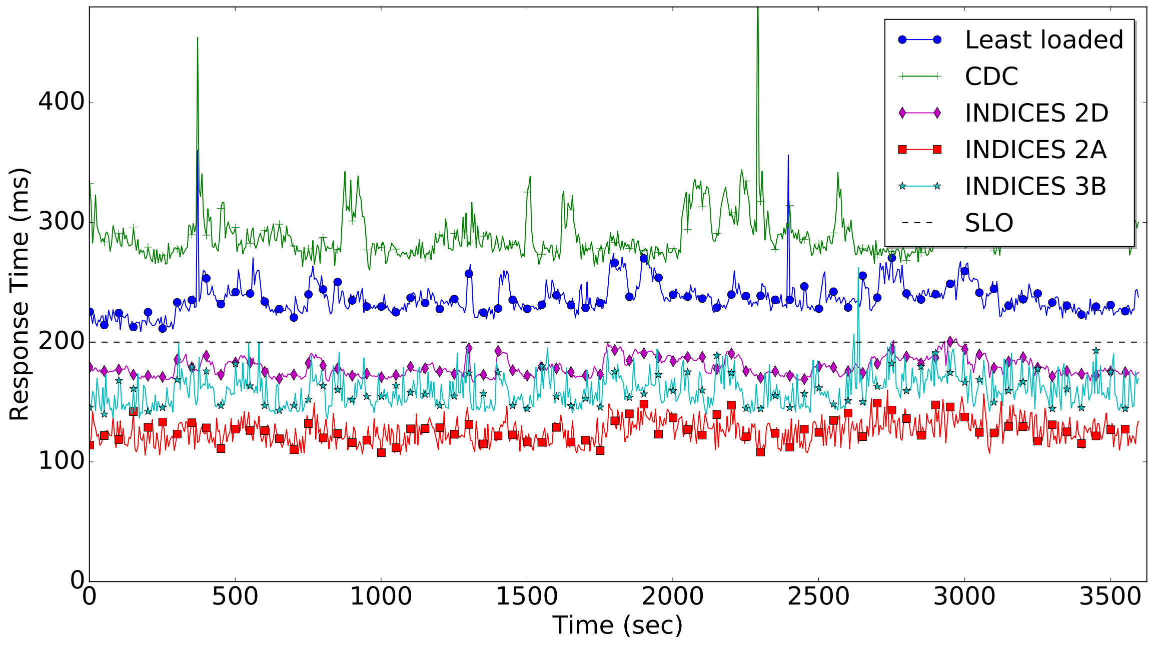


Figure 12: INDICES vs Least Loaded

Applying Algorithm 3 further, INDICES found the server of type B from MDC 3 to be most suitable since our objective is to select the minimum cost server to the service provider if it can meet the SLO. Thus, it preferred a server which already had an application that was going to run longer and had better bandwidth from the CDC server for migration. Figure 14 compares 3 migration scenarios (a) an overlay with the software stack already present on the target server and the bandwidth is 10 Mbps, (b) same as previous but with bandwidth 1 Mbps, (c) overlay is not present on the target server and the compressed file of size 938 MB has to transferred over 10 Mbps bandwidth. In all the scenarios, the application overlay and configuration files have to be transferred and the application has to be initialized. We observe that the server selection takes ≈ 1 sec, however, the migration and initialization takes 32s, 56s and 190s respectively for a, b and c scenarios. Thus, the overlay based image transfer should be the preferred methodology wherever applicable.

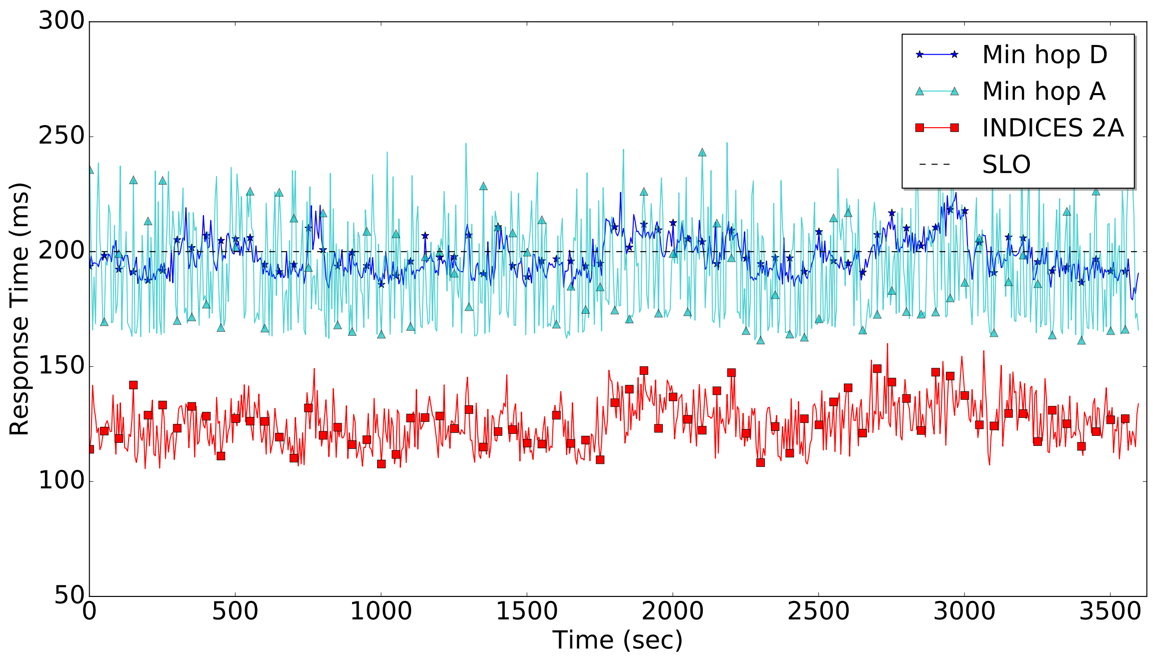


Figure 13: INDICES vs Minimum Hop

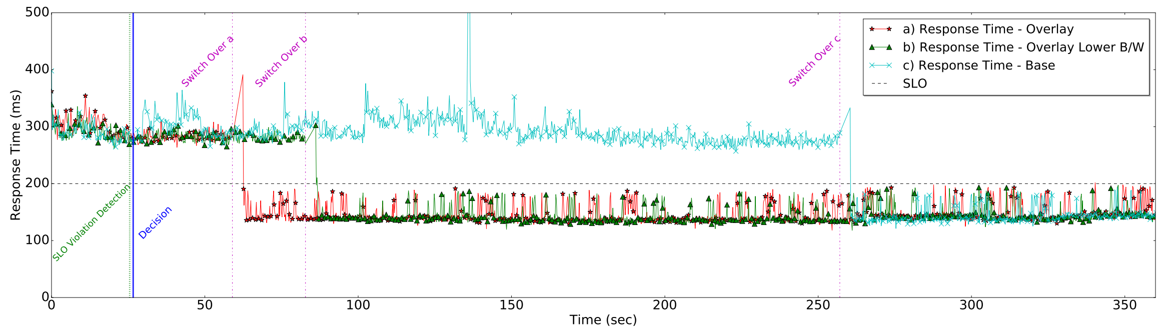


Figure 14: Application Switch-Over Performed by INDICES under 3 Different Scenarios

III.6 Related Work

In this section we compare and contrast our work with related work along three dimensions: network latency-based server selection, performance interference-based server selection and performance-aware edge computing. Unlike our work, our survey has found that existing works seldom consider all dimensions holistically.

III.6.1 Network Latency-based Server Selection

DONAR [172] addresses the global replica selection problem using a decentralized, selection algorithm where the underlying protocol solves an optimization problem that takes into account client performance and server load. CloudGPS [53] is a server selection scheme that considers network performance, inter-domain transit traffic and server workload for decision making. This work also reduces the network distance measurement costs. Dealer [70] targets geo-distributed, multi-tier and interactive applications to meet their stringent deadline constraints by monitoring individual component replicas and their communication latencies, and selects the combination that provides the best performance. Kwon et al. [96] applied network latency profiling and redundancy for cloud server selection while suggesting using cloudlets. We contend that these efforts consider simplistic models of server workload and their impact on performance, and do not cater to edge resource management.

III.6.2 Performance Interference-aware Server Selection

Paragon [49] identified the sources of interference that impact application performance and developed micro benchmarks for heterogeneous hardware. The system benchmarks applications and classifies them to find collocation patterns for scheduling. SMiTe [188] designed rulers for estimating sensitivity and degree of contention between applications when they are collocated. Bubble-Flux [180] assures QoS for latency-sensitive applications by dynamic interference profiling of shared hardware resources and collocating latency-sensitive applications with batch applications. These works, however, do not apply to virtualized data centers where the hypervisor places its own overhead on the resources and impacts performance. Moreover, our framework requires virtualized environments to support migration of applications on heterogeneous platforms.

DeepDive [129] first identifies an abnormal behavior using a warning system and employs an interference analyzer by cloning the target VM and running synthetic benchmarks.

Such an approach can be a costly runtime operation. Our prior work [35] designed a performance interference-aware resource management framework that benchmarks applications residing in virtual machines and applies a neural network-based regression mechanism that estimates a server's performance interference level. However, hardware heterogeneity and per application performance were not considered.

Heracles [110] mitigates performance interference issues for latency-sensitive applications by partitioning different shared resources. However, partitioning for resources, such as memory bandwidth is still not available, and moreover, cache partitioning is only available on newer hardware which cannot be applied to existing hardware.

III.6.3 Performance-aware Edge Computing

Zhou et al. [189] described a multi attribute decision analysis algorithm to offload tasks amongst mobile ad-hoc network, cloudlet and public cloud. Their work performs cost estimation considering execution time, power consumption, bandwidth and channel conjunction level which is utilized by the decision making algorithm. The approach utilizes ThinkAir [94] for offloading the tasks. However, they target only Java-based tasks and the solution is not catered to latency-sensitive applications such as those targeted by us.

Fesehaye et al. [58] described a design to select between cloudlets and central cloud server for interactive mobile cloud applications based on the number of hops, mobility and latency. SEGUE [186] is an edge cloud migration decision system that applies state-based Markov Decision Process (MDP) model incorporating network and server states. Both the approaches have not been evaluated on real systems and the results are only simulation-based.

SmartRank. [153] is a tool for offloading facial recognition from mobiles to cloudlets, and the scheduling is performed based on the round trip time and CPU utilization. In our approach, we optimize the cost to the cloud provider while maintaining the end user SLOs.

MobiQoR [102] is an optimization framework that trade-offs quality of result with response time and energy efficiency on mobile platforms. The approach provides significant improvement over the existing strategies, however, the solution is entirely edge centric and does not focus on offloading tasks to cloud/fog which we do in INDICES.

III.7 Concluding Remarks

This chapter presents an approach for dynamic cloud resource management that exploits the available edge/fog resources in the form of micro data centers, which are used to migrate cloud-hosted applications closer to the clients so that their response times are improved. In doing so, our algorithm ensures that existing edge-deployed services are not unduly impacted in terms of their performance nor are the operational and management costs for the cloud provider overly affected. These objectives are met using an online optimization problem, which is solved using a two-level cooperative and online process between system-level artifacts we have developed and deployed at both the micro data centers and centralized cloud data center. Our experimental results evaluating our framework called INDICES support our claims.

This work has opened up many new challenges and directions, which forms our future work. These insights are presented below:

- **Lack of benchmarks:** There is a general lack of open source and effective benchmarking suites that researchers can use to conduct edge/fog computing studies.
- **Collecting metrics under hardware heterogeneity:** The plethora of deloyed hardware configurations with different architectures and versions makes it hard to collect various performance metrics. Modern architectures are making it easier to collect more finer grained performance metrics, however, much more research is needed in identifying effective approaches to control the hardware and derive the best performance out of them.

- **Workload consolidation and migration across MDCs:** In the current work once an application is migrated to a MDC, it will complete its operation until termination. Our future work will consider dynamic server consolidation across MDCs and CDCs.

- **Reconciling application state:** In current work we have assumed that once the application state is transferred to the MDC, there is no additional state that accumulates at the CDC. However, for a broader set of applications, not all application state may be transferrable to the MDC and may have to be reconciled periodically with the CDC, which gives rise to interesting consistency versus availability tradeoffs.

- **Distributed user base:** In current work we have assumed that all distributed users of an interactive applications are located in close proximity to each other. However, for applications such as online games, this assumption may not hold for which additional research will be necessary.

- **Energy savings and revenue generation:** In current work we did not discuss revenue issues stemming from the use of edge resources. Moreover, energy savings is only indirectly referred to through our experimental results. Addressing these limitations forms dimensions of our future work.

- **Shared micro data centers:** In current work we have assumed that a MDC is exclusively controlled by a CDC provider. In future it is likely that MDC providers may lease their resources to multiple different CDCs. Additional research is needed to address situations where MDCs are shared including those that address security and isolation guarantees. Also, we need to ensure that the deployed code is free of vulnerabilities [182].

All scripts, source code, and experimental results for INDICES are available for download from `https://github.com/shekharshank/indices`.

CHAPTER IV

URMILA: UBIQUITOUS RESOURCE MANAGEMENT FOR INTERFERENCE AND LATENCY-AWARE SERVICES

IV.1 Motivation

The fog/edge computing paradigm [144] has evolved in recent years as a means to significantly alleviate the unpredictable and long round-trip latencies experienced by interactions between latency-sensitive cloud-based services and their clients over wide area networks. Yet, several issues that arise in cloud computing remain unresolved in fog/edge computing while new problems arise. For instance, hardware heterogeneity and performance interference [35, 49, 116, 129], which stems from resource sharing on multi-tenant hosts thereby degrading service performance, are issues that are even harder to resolve for fog/edge computing.

Mobility of the clients of latency-sensitive applications brings its own challenges. Consider the use cases of human, bike or vehicle to infrastructure communication commonly found in mobility assistance/optimization projects. In these scenarios, a user can move through regions with varying received wireless signal strength and intermittent connectivity. Frequent migration of service-related tasks and their data between the cloud and fog/edge can have an adverse impact on the overall user experience and lead to violations of service level objectives (SLOs). Finally, from the perspective of the service provider, there is a larger cost involved in running and maintaining the servers across the cloud-edge spectrum, which needs to be minimized. Thus, we need an approach that takes a holistic view of the edge, fog and cloud resources and provides a solution that not only performs an initial selection of appropriate fog/edge resources that can satisfy latency guarantees while minimizing the cost to the provider but also ensures that SLOs are not violated throughout the lifetime of the service which includes mobility of its users. This assurance must be provided while minimizing energy consumption of the users' mobile devices as well as minimizing cost to the cloud provider. There is increasing interest in addressing these problems as evidenced by recent research efforts [86, 148, 149, 156, 168]. However, these solutions do not holistically solve all the dimensions of the problems that we have outlined, namely, (a) initial server selection for application deployment from the cloud to the fog, (b) minimizing co-hosted application interference, and (c) minimizing battery consumption on edge devices.

To that end, we present *URMILA (Ubiquitous Resource Management for Interference and Latency-Aware services)*, which is a middleware solution to manage the cloud, fog and edge resource spectrum and to ensure that SLO violations are minimized for latency-sensitive applications, particularly those that are utilized in mobile environments. Specifically, we make the following contributions:

- We provide a wireless signal strength estimation mechanism to *a priori* estimate the energy consumption and network latency in mobile environment, which aids in resource selection for mobile users.
- We formulate an optimization problem that minimizes the cost to the cloud provider and energy consumption on mobile devices while adhering to SLO requirements.
- We propose a server selection algorithm that accounts for performance interference due to co-location of services on multi-tenant servers and performs NUMA-aware performance prediction in cloud/fog environment. We also deliver a runtime control algorithm for task execution that ensures SLOs are met in real time.
- We evaluate our solution in realistic environments using two types of client applications.

The rest of this chapter is organized as the following. Section IV.2 discusses the application and the system models; Section IV.3 formulates the optimization problem and describes the challenges we address; Section IV.4 explains the system architecture and the solutions for the problem; Section IV.5 provides empirical validations of our work; Section IV.6 provides a discussion of the work alluding to future work; Section IV.7 describes related work comparing it to URMILA; and finally Section IV.8 provides the concluding remarks.

IV.2 System Model and Assumptions

We formally present the different aspects of our system model and assumptions we made in this chapter including a motivational use case that fits this model.

IV.2.1 Application Model and Motivational Use Case

We target latency-sensitive services that are interactive or streaming in nature, such as augmented reality, online gaming, navigational applications and cognitive assistance. Individual tasks of these services can run on both mobile devices and on cloud-hosted servers that cater to the Software-as-a-Service (SaaS) or Platform-as-a-Service (PaaS) model. Users of these services are assumed to be mobile within a region that has a high density of wireless access points (WAPs), such as a university campus or wireless hotspots owned by internet service providers that can host fog resources. As a motivational use case we consider a real-time object detection cognitive assistance application targeted towards the visually impaired. Advances in wearable devices and computer vision algorithms have enabled cognitive assistance and augmented reality applications to be realized. Examples of such work include Microsoft and PivotHead's SeeingAI [4] and Gabriel [144] that leverages Google Glass and cloudlets. However, either because these solutions are still not available to the users or use discontinued technologies such as Google Glass, we have developed our own applications for the experimental purposes.

To that end, we developed two applications. First is an Android application interoperating with a Sony SmartEyeGlass that captures video frames as the user moves in a region and provides audio feedback after processing the frame. The second is a Python application running on Linux-based board devices such as MinnowBoard with a web camera. We use MobileNet [76] and Inception V3 [160] real-time object detection algorithms from Tensorflow for image processing. The application's execution is considered to be streaming in nature having multiple independent time steps of approximately equal length. Each step provides a complete cycle of service to the user, e.g., a cycle in the cognitive assistance application involves capturing a video frame on the mobile device, sending it to the image processing unit for processing, and responding to the user.

Since the user needs feedback in real-time, we have tight deadlines with predefined SLOs that need to be guaranteed. Moreover, since the image processing is a compute and memory intensive application, it consumes the already scarce battery resources on a mobile device. To address this second issue, although cyberforaging enables a mobile application to be offloaded from the edge device to a fog/cloud node where it gets deployed and processed [18], this process itself is energy consuming and is platform dependent because of application executable gets transferred. Hence, in this work we take an approach where one version of the application is deployed in containerized form at the cloud/fog node and another instance runs on the client device. This also helps in fault tolerance and guaranteeing SLOs.

At each time step, the application can execute either locally on the mobile device or remotely in the fog or cloud. If it executes locally, there is no network delay, but the execution time and power consumed by the device is high. On the other hand, if the application executes remotely, the mobile device's power consumption is low, but the network latency becomes non-negligible, which will depend on the location of the user and his/her proximity to the fog node where the application is deployed. The timing requirements of the application, resource scarcity and trade-offs in local/remote execution impose certain SLOs and energy costs for the application and service hosting costs for the provider, which we capture in our model that is described next.

For each user (or application[1]), u, let ϕ_u denote the user-specific bound on the acceptable response time in each service cycle, which also defines the length of a time step. For our consideration, the total response time experienced by the user at any time step ℓ can be expressed as:

$$t_{total}(u,\ell) = t_{process}(u) + t_{execute}(u,\ell) + t_{network}(u,\ell) \tag{IV.1}$$

where $t_{process}(u)$ denotes the required local pre/post-processing time of the application (which is fixed and independent of the execution mode and time step), $t_{execute}(u,\ell)$ denotes the actual execution time for the service at step ℓ (which depends on whether the execution is on-device or remote), and $t_{network}(u,\ell)$ denotes the network latency for step ℓ (only if remote execution is involved).

The goal is to meet the SLO for the user, i.e., $t_{total}(u,\ell) \leq \phi_u$, while minimizing the total cost (which includes both the server deployment cost and the user energy cost as formulated in Section IV.3). Note that the SLO needs to be guaranteed for each time step ℓ in the user's anticipated duration of application usage. Since we consider user mobility, this duration is typically from the start to the end of the user's trip. Nonetheless, there is nothing to prevent us from applying the model even in the stationary state or after the user has reached his/her destination.

Let $t_{local}(u)$ denote the execution time when application u is run locally, which is fixed regardless of the time step and no network latency will be incurred in this case. Additionally, we assume that the SLO can always be satisfied with local executions, i.e., $t_{process}(u) + t_{local}(u) \leq \phi_u$ for all u and ℓ. This could be achieved by a lightweight mobile version of the application, such as MobileNet for real time object detection on the mobile device, which is less compute-intensive and time-consuming, thereby ensuring the SLO albeit with a low detection accuracy.

[1]In this work, we assume that the considered applications are all single-user applications and that each user runs only one such application. Hence, we will use the terms "user" and "application" interchangeably.

IV.2.2 Architectural Model

We now describe the architectural model in order to assure the SLOs for the applications modeled in Section IV.2.1 while minimizing the energy consumption of the mobile device and the cost to the service provider.

The architectural model shown in Figure 15 that we consider in this work consists of wireless access points (WAPs) that leverage fog resources, which comprise compute servers. The mobile devices have standard 2.4 GHz WiFi adapters to connect to the WAPs and they implement well-established mechanisms to hand-off from one WAP to another. We assume that mobile clients are not using cellular network for the application's data transmission needs due to higher monetory cost for cellular services as well as higher energy consumption for cellular over wireless networks [51, 67, 79].

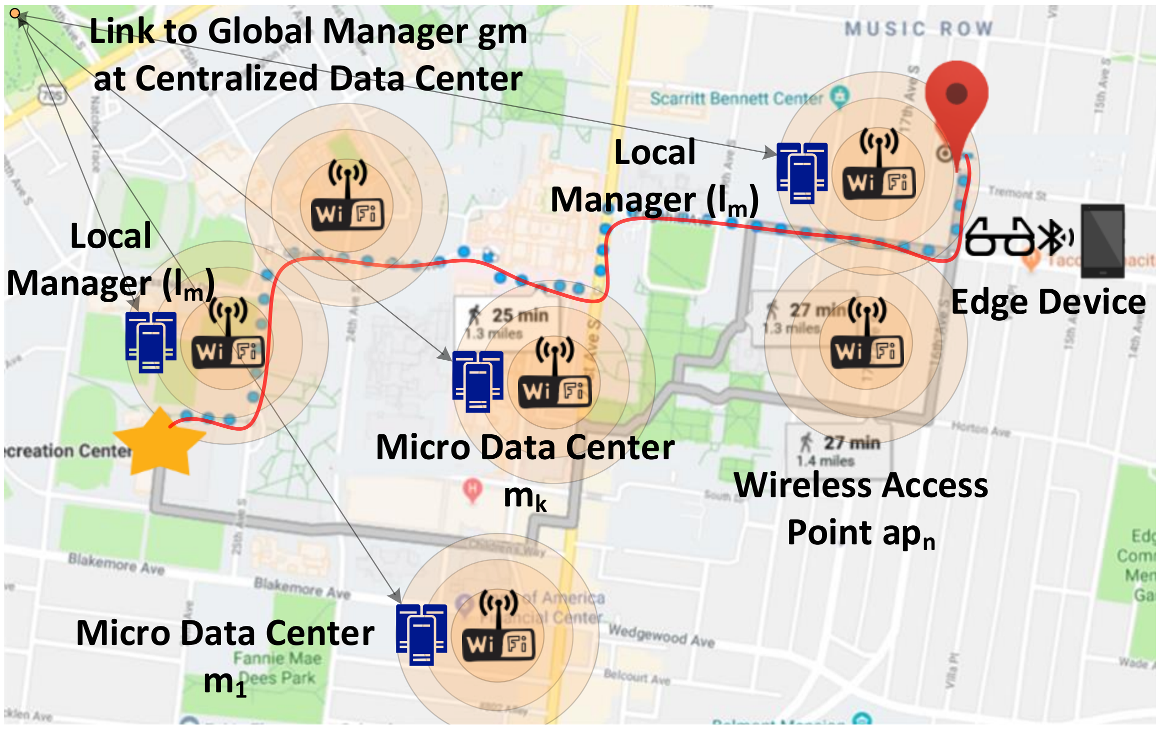


Figure 15: Architectural Model

In our model, the applications and the fog resources are managed by a centralized authority known as the global manager (*gm*) hosted at a centralized data center (CDC). We denote the set $AP = \{ap_1, ap_2, \ldots, ap_n\}$ of n WAPs with a subset of them also hosting fog resources in the form of micro data centers (MDCs) or cloudlets. Such capabilities could be offered by college campuses. We assume that the *gm* owns or has exclusive lease to a set $M = \{m_1, m_2, \ldots, m_k\}$ of k MDCs. Note that M is a subset of AP since only some WAPs have an associated MDC. Each MDC $m_i \in M$ contains a set $S_i = \{s_{i,1}, s_{i,2}, \ldots, s_{i,h_i}\}$ of h_i compute servers (possibly heterogeneous) that are connected to their MDC's associated WAP ap_i. From a traditional cloud computing perspective, since an application can be deployed and executed on the CDC, we model the CDC as a special MDC that is also contained in the set M, and correspondingly, the set AP contains the access point that hosts the CDC as well.

The network latency between ap_i and any server $s_{i,h} \in S_i$ in m_i is assumed to be negligible, i.e., $t_{ap_i,s_{i,h}} \approx 0$, as they are connected via fast local area network (LAN). Different WAPs are connected to each other over a wide area network (WAN) and may incur significant latency. Let t_{ap_i,ap_j} denote the round trip latency between ap_i and ap_j, and this latency can vary depending on the distance, connection type and number of hops between the two WAPs.

In this architecture, if a mobile user is connected to a nearby WAP, say ap_i, which also has an MDC m_i that hosts the user's application on one of its servers, then there is no additional access point involved, hence the latency between access points, i.e., $t_{ap_i,ap_i} = 0$. However, if the application is deployed on another MDC, say m_j, then the round trip latency $t_{ap_i,ap_j} > 0$ can be significant since the request/response will be forwarded from the connected access point ap_i to the access point ap_j associated MDC server hosting the application. Moreover, due to mobility of the user, the user could at times have no connection to any access point (e.g., out of range). In this case, we assume the presence of a virtual access point ap_0 to which the user is connected and define $t_{ap_0,ap_i} = \infty$ for any $ap_i \in AP$. Obviously, the application will have to run locally to avoid SLO violations.

In addition to the round trip latency, the selection of MDC and server to deploy the application can also significantly impact the application execution time, since the MDCs can have heterogeneous configurations and each server can host multiple virtualized services, which do not have perfect isolation and hence could interfere with each other's performance [35, 49, 116, 129]. Each MDC m_i, also contains a local manager lm_i which is responsible for maintaining a database of applications it can host, their network latencies for the typical load, and server type and load-specific application execution time models. Note that there could be a varying number of co-located applications and hence a varying load on each server over time but we assume that individual application's workload does not experience significant variation throughout its lifetime, which is a reasonable assumption for many interactive applications, such as processing image frames with constant size.

IV.2.3 User Mobility Model

Since our focus is on assuring SLOs for mobile users by intelligently executing service tasks either locally or on fog resources, it is important to estimate user mobility patterns with reasonable accuracy. The observed latencies depend heavily on the route taken by the user for a given configuration of WAPs in a region.

Estimating mobility patterns can broadly be classified into two categories: 1) *Probabilistic*; and 2) *Deterministic*. The probabilistic approach uses data driven AI techniques [54, 128, 171] to calculate the most probable route that a user would take at a given time of the day. However, this approach can be substantially data intensive and also lack generality. The second approach relies on user's input and a navigation service such as Open Street Maps (`http://www.openstreetmap.org`), Here APIs (`https://developer.here.com/`) and the one used in our current implementation, Google Maps APIs (`https://cloud.google.com/maps-platform/`) to determine a fixed route for a given pair of start and end locations. URMILA uses the deterministic approach while we assume a constant velocity model, i.e., the user moves at a constant speed throughout the route.

Despite knowing the route, estimating latencies along the route is challenging due to dynamic nature of the WiFi channels and changing traffic patterns throughout the day. URMILA employs a data driven model that maps every route point on the path to expected latency to be observed at that point. One of the salient features of this estimation model is its adaptability, i.e., the model is refined continuously in accordance to the actual observed latencies. Latency estimation is described in detail in Section IV.4.

For our mobility model, we divide the travel duration for each user u into a sequence $I(u) = \{1(u), 2(u), \ldots, L(u)\}$ of time steps that cover the user's path. The length of each time step $\ell \in I(u)$ is the same and sufficiently small so that the user is considered to be constantly and stably connected to a particular WAP $ap(u,\ell) \in AP \bigcup \{ap_0\}$ (including the virtual access point). Moreover, the round trip latency $t_{u,ap(u,\ell)}$ between the user and this access point can be estimated based on the user's position, channel utilization, and number of active users connected to that access point [158].

IV.2.4 Request Flow Summary

We now summarize the runtime interactions using our cognitive assistance use case. The client-side application is assumed to be aware of the *gm* and communicates with it to provide the start time, source and destination for the trip, as well as the route the user is going to take. Based on this information, and the *a priori* knowledge of the application behavior and its load information communicated by the lm_i, the *gm* decides a suitable server in an MDC for the application, informs the client application about the server location and deploys the application on the selected MDC server. Using this deployment information and the network condition at that instant, at each time step the client application decides whether to process the request locally or remoted. The process continues till the user reaches the destination and stops using the service.

IV.3 Problem Formulation

The problem we solve in this chapter is to determine for each user as to which cloud or fog server should the user's application be deployed on and which execution mode (local or remote) should be invoked at each step of its execution in a way that will assure the SLO for the user and also minimize the overall cost that includes both the deployment cost for the service provider and the energy cost on the user's mobile device. To formalize this optimization problem, we define two binary variables that indicate the decisions for application deployment and execution mode selection, respectively:

$$x_{u,s_{i,h}} = \begin{cases} 1 & \text{if user } u \text{ is deployed on server } s_{i,h} \\ 0 & \text{otherwise} \end{cases}$$

$$y_{u,s_{i,h},\ell} = \begin{cases} 1 & \text{if user } u \text{ executes on server } s_{i,h} \text{ at step } \ell \\ 0 & \text{otherwise} \end{cases}$$

Using these two variables and our system model, we now express the total response time of an application and the total cost, and then present the complete formulation of the optimization problem.

IV.3.1 Total Response Time

Recall from Equation (IV.1) that the total response time for a user u at a time step ℓ consists of three parts, and among them the pre/post-processing time $t_{process}(u)$ is fixed. To express the execution time, let $t_{remote}(u, s_{i,h}, \ell)$ denote user u's execution time if it is run remotely on server $s_{i,h}$ at time step ℓ. Note that, due to the hardware heterogeneity and co-location of multiple applications on the server which can result in performance interference [116, 180, 187], this execution time will depend on the set of existing applications that are running on the server at the same time. This property is known as *sensitivity* [95, 116, 176, 188]. Similarly, the execution times for these users may in turn be affected by the application execution of user u were it to execute on this server – a property known as *pressure* [95, 116, 176, 188]. Techniques to estimate $t_{remote}(u, s_{i,h}, \ell)$ appear in Section IV.4.1.3.

For the network latency, let $t_{network}(u, s_{i,h}, \ell)$ denote the total latency incurred by running the application remotely on server $s_{i,h}$ at time step ℓ. We can express it as:

$$t_{network}(u, s_{i,h}, \ell) = t_{u,ap(u,\ell)} + t_{ap(u,\ell),ap_i} + t_{ap_i,s_{i,h}} \tag{IV.2}$$

In particular, the total includes the latency from the user to the connected access point $t_{u,ap(u,\ell)}$, which we refer to as the *last-hop latency*; the latency from the connected access point to the serving access point $t_{ap(u,\ell),ap_i}$, which we refer to as the *WAN* latency; and the latency from the serving access point to the server that deploys the user's application $t_{ap_i,s_{i,h}}$, which we refer to as the *server latency*. Among these, the server latency is negligible, and the first two depend on the user's location at time step ℓ. Latency estimation is discussed in Section IV.4.1.2.

Now, the total response time of user u at time step ℓ can be expressed as follows:

$$\begin{aligned} t_{total}(u, \ell) = t_{process}(u) + \Big(1 - \sum_{i,h} y_{u,s_{i,h},\ell}\Big) t_{local}(u) \\ + \sum_{i,h} y_{u,s_{i,h},\ell} \Big(t_{remote}(u, s_{i,h}, \ell) + t_{network}(u, s_{i,h}, \ell)\Big) \end{aligned} \tag{IV.3}$$

In the above expression, the first line includes the constant pre/post-processing time as well as the execution time when the application runs locally, and the second line includes the execution time when it is run remotely as well as the incurred total network latency.

IV.3.2 Total Cost

The total cost consists of two parts: the server deployment cost and the user energy cost. On the server side, a monetary allocation cost is involved. In addition, running a server incurs operational costs, such as need for power and cooling. Thus, the provider must use as few server-seconds as possible and hence the deployment cost depends on the duration for which a server remains on. For a server $s_{i,h}$, let $U(s_{i,h})$ denote the set of existing users whose applications are deployed on the server, and define $L_{\max}(s_{i,h})$ to be the maximum duration for which these existing applications will run, i.e., $L_{\max}(s_{i,h}) = \max_{v \in U(s_{i,h})} L(v)$. The cost for deploying a new application u on server $s_{i,h}$ is proportional to the extra duration for which the server has to remain on and can be expressed as:

$$T_{deploy}(u, s_{i,h}) = \max\left(0, L(u) - L_{\max}(s_{i,h})\right) \tag{IV.4}$$

In addition to the operational cost, deploying a service on an MDC server requires transferring its state over the backhaul network from the application repository in the CDC to the MDC. The time to transfer the state of an application u to a server $s_{i,h}$ can be expressed as:

$$T_{transfer}(u, s_{i,h}) = \frac{state_u}{b_i} + ci_{u,s_{i,h}} \tag{IV.5}$$

where $state_u$ is the size of application u's state, b_i is the backhaul bandwidth from CDC to MDC m_i, and $ci_{u,s_{i,h}}$ is the initialization time of the application before it can start processing requests on the MDC server. Hence, the earliest time step at which the application can be executed remotely on the server is:

$$\ell'(u, s_{i,h}) = 1(u) + \left\lceil \frac{T_{transfer}(u, s_{i,h})}{\phi_u} \right\rceil \tag{IV.6}$$

On the user side, recall that executing the application locally incurs higher power consumption than executing it remotely. Hence, the cost for user u can be measured in terms of the total number of time steps when the application is being run locally, which is directly proportional to the additional energy expended by the mobile device had the application been run remotely throughout the user's travel. The number of local time steps by deploying application u on server $s_{i,h}$ can be expressed as:

$$\begin{aligned} T_{user}(u, s_{i,h}) = \min\Big(\ell'(u, s_{i,h}), L(u)\Big) - 1(u) \\ + \sum_{\ell=\ell'(u,s_{i,h})}^{L(u)} \Big(1 - y_{u,s_{i,h},\ell}\Big) \end{aligned} \tag{IV.7}$$

To combine the costs from different sources above, we define $\alpha_{i,h}$ and $\beta_{i,h}$ to be the unit-time costs of powering on the server $s_{i,h}$ and transferring the state to server $s_{i,h}$, respectively. Both these values depend on the server and its corresponding MDC. In addition, we define κ_u to be the per-step energy cost of local execution for user u (relative to remote executions), and its value depends on the user's application and mobile device. Thus, for a given solution that specifies the application deployment (i.e., $x_{u,s_{i,h}}$) and its execution mode at each time step (i.e., $y_{u,s_{i,h},\ell}$), the total cost can be expressed as:

$$\begin{aligned} C(u) = \sum_{i,h} x_{u,s_{i,h}} \Big(\alpha_{i,h} \cdot T_{deploy}(u, s_{i,h}) \\ + \beta_{i,h} \cdot T_{transfer}(u, s_{i,h}) \\ + \kappa_u \cdot T_{user}(u, s_{i,h})\Big) \end{aligned} \tag{IV.8}$$

IV.3.3 Optimization Problem

Given the expressions for total response time (Equation (IV.3)) and total cost (Equation (IV.8)), the optimization problem needs to decide, for each new user or application u, where to deploy the application and which execution mode to run the application in order to minimize the total cost subject to the response time constraints. Let V denote the set of all existing applications at the time of deploying application u, i.e., $V = \sum_{i,h} U(s_{i,h})$. The problem can then be formulated as:

$$
\begin{aligned}
\text{minimize} \quad & C(u) \\
\text{subject to} \quad & t_{total}(u,\ell) \leq \phi_u, \ \ \forall \ell && \text{(IV.9)} \\
& t_{total}(v,\ell) \leq \phi_v, \ \ \forall \ell, v \in V && \text{(IV.10)} \\
& x_{u,s_{i,h}}, y_{u,s_{i,h},\ell} \in \{0,1\}, \ \ \forall s_{i,h}, \ell && \text{(IV.11)} \\
& \textstyle\sum_{i,h} x_{u,s_{i,h}} \leq 1 && \text{(IV.12)} \\
& y_{u,s_{i,h},\ell} \leq x_{u,s_{i,h}}, \ \ \forall s_{i,h}, \ell && \text{(IV.13)} \\
& y_{u,s_{i,h},\ell} = 0, \ \ \forall s_{i,h}, \ell < \ell'(u, s_{i,h}) && \text{(IV.14)}
\end{aligned}
$$

In particular, Constraints (IV.9) and (IV.10) require meeting the SLOs for user u as well as for all existing users at all times. Constraint (IV.11) requires the decision variables to be binary. Constraint (IV.12) requires the application to be deployed on at most one server. We enforce this constraint because there is a high cost in transferring the application state from the CDC to an MDC server, initializing and running it. Note that an application need not be deployed on any server, in which case it will be executed locally throughout the user's travel duration. Constraint (IV.13) allows the application to run remotely only on the server it is deployed at each step and Constraint (IV.14) restricts the remote executions to start only after the application state has been transferred.

IV.4 Design and Implementation

We now describe our URMILA solution to solve the optimization problem formulated in Section IV.3.3 focusing on its deployment and runtime phase responsibilities.

For the considered problem, the client initially connects to the global manager (*gm*) and informs it about the application being accessed as well as the route information. The service then gets initialized on the client device and the *gm*. This section describes the initial deployment phase (Section IV.4.1), which is triggered by the *gm*, and the runtime phase (Section IV.4.2), which occurs predominantly on the client device.

IV.4.1 Deployment Phase

The initial deployment phase consists of three components, *Route Calculation*, *Latency Estimation* and *Fog Node Selection*, connected in sequence as shown in Figure 16. These stages are repeated when new users/jobs are added to the system. The first component calculates the route taken by the user, the second provides quantitative estimates about the observed latency along the route and the third determines the fog node for application deployment by solving the optimization problem defined in Section IV.3.3. These components are functional in nature and are discussed next.

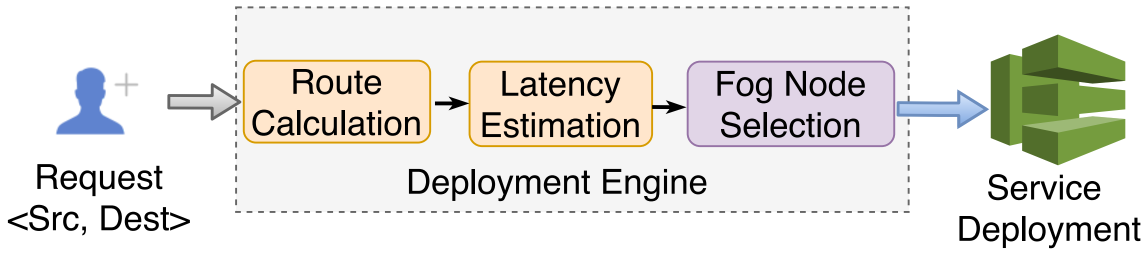


Figure 16: Components in Initial Deployment

IV.4.1.1 Route Calculation

This component is responsible for determining the user's mobility pattern based on the deterministic methodology described in Section IV.2.3. We leverage Google Maps APIs for finding the shortest route between the user's specified start and destination locations. This component takes a tuple comprising the start and destination GPS coordinates as input, and produces a list of GPS coordinates for the various steps along the route. The raw list of route points returned by the navigation service are re-sampled as per a constant velocity model (1.4 meters/sec) with an interval equal to the response time deadline enforced by the SLO.

IV.4.1.2 Latency Estimation

This component gives a quantitative estimate about the latency $t_{network}(u, s_{i,h}, \ell)$ observed by user u at any step $\ell \in I(u)$ along the route for any given server $s_{i,h}$. As shown in Equation (IV.2), this latency is the sum of the last-hop latency, WAN latency, and server latency (which is negligible). The rest of this subsection describes how the first two latencies are estimated when the application is deployed in MDC m_i.

Last-hop latency $t_{u,ap(u,\ell)}$: A number of factors affect the last-hop latency, but predominantly *channel utilization*, *number of active users*, *received signal strength*, as identified in [158]. In URMILA, initially, we assume that channel utilization and number of active users are within bounds and thus does not impact latency significantly. Later, as the routes get profiled, URMILA maintains a database that stores network latency for different coordinates and time of the day. Whenever a request arrives for known route segments, the latency estimator can lookup this database.

The other key component for network connectivity and latency is received signal strength. Beyond a threshold (-67 for streaming applications) of received signal strength, network latency becomes unreliable and connection should not be used [158]. Thus, we need to find the signal strength, and it is obtained using Equation (IV.15), where $\hat{p}$ (resp. $\hat{p_0}$) is the mean received power at a distance d (resp. d_0) from the access point, and γ is the path loss exponent.

$$\hat{p}(d) = \hat{p_0}(d_0) - 10\gamma \log \frac{d}{d_0} \tag{IV.15}$$

Among these parameters, $\hat{p_0}$ and d_0 depend on the access point and are known *a priori* for typical access points. The path loss exponent γ depends on the environment, and its typical values for free space, urban area, sub urban area and indoor (line of sight) are 2, 2.7 to 3.5, 3 to 5 and 1.6 to 1.8 respectively [137]. We obtain an estimate of the exponent, $\hat{\gamma}$, using the equation IV.16 as described in [114], where P_i is the receiving power at distance d_i, P_0 is the received power at reference distance and N are the number of observations.

$$\hat{\gamma} = -1 \sum_{i=1}^{N} \frac{(P_i[dBm] - P_0[dBm]) \log_{10} d_i}{10(\log_{10} d_i)^2} \tag{IV.16}$$

The last-hop latency also depends on the access point selection and switching policy employed by the mobile device. There are many policies for access point selection [41, 45, 87, 126, 177]. In URMILA, we make access point selection based upon the received signal strength, i.e., the client device will select an access point with the highest signal strength and sticks to the same access point till the signal strength drops to some threshold. Thus, using this policy, the calculated route, and the access points data, the latency estimator is able to calculate the last-hop latency for each step of the route.

WAN latency $t_{ap(u,\ell),ap_i}$: The WAN latency between two access points depends upon the link capacity connecting the nodes and the number of hops between them. In URMILA, we use another database to keep track of the latencies between different access points. Based on this data and the estimated last-hop latency described previously, a map of the total network latency can now be generated for every step along a user's route.

IV.4.1.3 Fog Node Selection

The objective of the deployment phase is to select a server from all servers in the system to deploy the application such that the SLOs are met, the energy consumptions on the client devices are minimized, and the lowest cost to the service provider is incurred. There are two approaches to server selection. One approach could keep deploying/undeploying the application at a selected server as the user moves from the coverage of one MDC to another. This will provide lower latencies when the user is at the access point that hosts an MDC. However, there is a high cost involved in transferring the image from CDC to MDC and initializing it. Moreover, if there is a deviation from the ideal user behavior i.e., moving at variable speed, we will have a even higher cost for new server selection and deployment based on current conditions.

Thus, URMILA performs a one time initial deployment on a single cloud/fog server, and reserves the resource for the entire trip duration plus a margin to account for the deviation from ideal behavior. For the server selection, in addition to having accurate estimates of latency, we also need to have an accurate estimate for the remote execution time of an application when deployed on a particular server. Therefore, we need to model the application behavior and measure the server load. For this we need a distributed performance monitoring framework, which not only collects metrics at the system level but also at the micro-architectural level as it is required to quantify the performance interference.

To accomplish this, we leverage the INDICES [149] performance metric collection and interference modeling framework. However, the INDICES framework has a few limitations. In particular, it was designed for virtual machines (VMs). In this work, in order to have lower initialization cost compared to VMs [150], we rely on Docker containers. Hence, as a part of URMIL, we integrated INDICES while extending the framework for interference-aware Docker container deployment. In addition, modern hardwares are equipped with non uniform memory access (NUMA) architecture which forces the performance estimation and scheduling techniques to consider memory locality. Different applications have different levels of performance sensitivity on NUMA architectures [136]. Thus, we need a mechanism that is able to benchmark applications on different NUMA nodes and predict their performance and schedule them accordingly. Moreover, recent advancement in Cache Monitoring Technology (CMT) [125] provides further insights about system resource consumption, such as memory bandwidth and last-level cache utilization, which can be leveraged for better performance estimation. We account for all of these factors in URMILA.

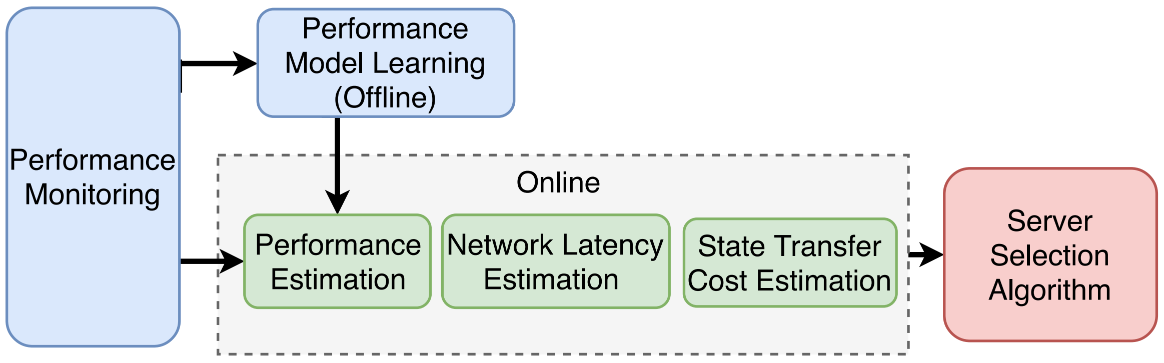

Performance Monitoring
Performance Model Learning (Offline)
Online
Performance Estimation
Network Latency Estimation
State Transfer Cost Estimation
Server Selection Algorithm

Figure 17: Fog Node Selection Process

We now describe the different components of the server selection process as illustrated in Figure 17.

Offline Stage: In the offline stage, we develop a performance model for each application. Such a model depends on two factors. The first factor is the server architecture and configuration, which is a leading cause of performance variability [49]. The second factor is the application's sensitivity in the presence of co-located applications and its pressure on those applications [95, 116, 176, 188], which exist because there is no perfect isolation in shared multi-tenant environments. Although hypervisors or virtual machine monitors allocate a separate virtual CPU, memory and network space for each virtual machine (VM) or container, there still exist a number of interference sources [116, 180, 187], e.g., shared last-level cache, interconnect network, disk and memory bandwidth.

To obtain a performance interference model, we first benchmark the execution time $t_{isolation}(u,w)$ of each application u on a particular hardware type w in isolation. Then, we execute the application with different co-located workload patterns and learn its impact g_u

on the system-level and micro-architectural metrics as follows:

$$X_w^{new} = g_u(X_w^{cur}) \tag{IV.17}$$

where X_w^{cur} and X_w^{new} denote the vectors of the selected metrics before and after running application u on hardware w, respectively. Lastly, we learn the performance deterioration $f_u(X_w)$ (compared to isolated performance) for application u under any metric vector X_w on hardware w, which allows us to predict its remote execution time under the same condition as follows:

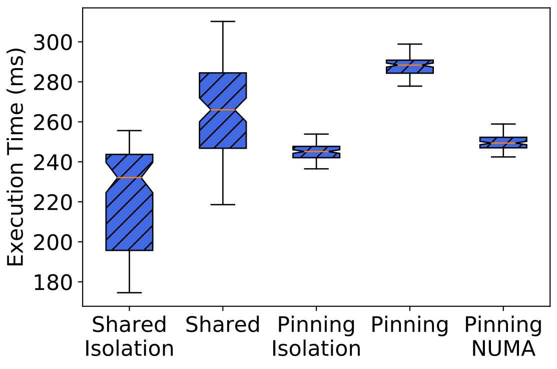
$$t_{remote}(u,w) = t_{isolation}(u,w) \cdot f_u(X_w) \tag{IV.18}$$

We apply supervised machine learning based on Gradient Tree Boosting curve fitting [56] to learn both functions g_u and f_u. This stage also involves feature selection, correlation analysis and regression technique selection among other steps, , but we do not describe them and refer to the INDICES framework [149] for more details.

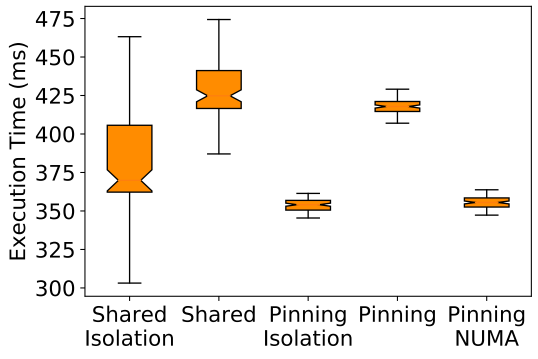
Note that Equations (IV.17) and (IV.18) can be applied together to model both sensitivity and pressure for application deployment on each server in order to accurately estimate remote execution times.

The use of Linux container-based deployment allows us to reduce the state transfer cost (i.e., $T_{transfer}(u, s_{i,h})$ in Equation (IV.5)). In case of VM-based deployment, the CPU resources are assigned as virtual CPUs (vCPUs), which share the physical cores. For Docker containers, virtual-to-physical core mapping does not exist and there are two mechanisms available for resource isolation. We can either specify the share of CPU cores available to the containers or we can pin the containers to dedicated CPU cores. In our resource allocation policy, we opt for the latter as it provides better resource isolation and results in less performance interference, although interference still exists due to the non-partitionability of shared resources as described before.

When using CPU pinning, one of the key consideration is NUMA-aware scheduling. On modern multi-chip servers, the memory is divided and configured locally for each processor. The memory access time is lower when accessed from local NUMA node compared to when accessed from remote NUMA node. Hence, it is desirable to model the performance per NUMA node and schedule the Docker containers accordingly. We achieve this by collecting the performance metrics per NUMA node and then developing sensitivity and pressure profiles at the NUMA node level instead of at the system level. The benefit of this approach is validated in Figure 18. We observe from the figure that CPU core pinning reduces the performance variability, however, if NUMA node is not accounted for, it could lead worse performance due to data locality issues.



(a) MobileNet



(b) Inception

Figure 18: Execution Time Comparison

Once we develop the performance models for the different applications, we distribute them to the different MDC locations for each of the hardware type *w* they contain. Typically, MDCs contain 10-20 servers with just a few heterogeneous types; thus we do not anticipate a large amount of performance model dissipation.

Online Stage: The online server selection operates in a hierarchical fashion. The global manager *gm* residing at the CDC initiates the server selection process as soon as it receives a request from the client application. It first calculates the route of the user as described in Section IV.4.1.1. Once it knows the route and the access points that the user will be connected to, the *gm* then queries the local managers *lm* of each MDC, which in turn queries each of their servers to find the expected execution time of the target application using the performance model developed in the offline stage such that the SLOs for the existing applications can still be met. Finally, the *gm* combines this information with the latency estimates from Section IV.4.1.2 to determine the execution mode of the application to satisfy the response time constraints at each step of the route. This allows us to estimate the cost incurred by the user (i.e., T_{user} in Equation (IV.7)).

We still need to estimate the deployment cost (i.e., T_{deploy} in Equation (IV.4)) and the transfer cost (i.e, $T_{transfer}$ in Equation (IV.5)) to solve the optimization problem. The deployment cost is based on the trip duration, which we can again obtain from the user mobility as described in Section IV.4.1.1. As mentioned before, we use Docker technology to minimize state transfer and initialization cost. The Docker container images consist of layers. Each layer other than the last one is read only and is made of a set of differences from the layer below it. Thus, to construct a container, different layers are combined. For this purpose, we ensure that a base image (such as Ubuntu 16.04) is already present on the server and only the delta layers (that dictate $state_u$ in Equation (IV.5)) need to be transferred for the application to be reconstructed at the fog location.

IV.4.1.4 Server Selection Algorithm

Algorithm 4 shows the pseudocode for the server selection. Besides deciding on the server to deploy the target application, the algorithm also suggests a tentative execution-mode plan at each step of the application execution. This execution plan will be used for cost estimation by the global manager and is subject to dynamic adjustment at runtime as explained in Section IV.4.2.

The algorithm first goes through each MDC and each server (Lines 2 and 3), and checks whether deploying the target application *u* will result in SLO violations for the existing applications on that server (Lines 4-13). For each existing application *v*, since it may have a variable network latency and execution times depending on the user's location and choice of execution mode, we should ideally check for *v*'s SLO at each step of its execution. However, doing so may incur unnecessary overhead on the global manager since the execution-mode plan for *v* is also tentative. Instead, the algorithm considers the overall SLO by using the estimated network SLO percentile latency (90^{th}, 95^{th}, 99^{th}, etc.) while assuming that the application always executes remotely. This approach provides a more robust performance guarantee for the existing applications in case of unexpected user mobility behaviors.

Subsequently, for each feasible server, the algorithm evaluates the overall cost of deploying the application on that server (Lines 14-27) and chooses the one that results in the least cost (Lines 28-30). Note that, for each feasible server, the deployment cost and state transfer cost are fixed, so the only variable cost to consider is the user's energy cost, which depends on the execution mode vector *y*. Hence, to minimize the overall cost, the algorithm offloads the execution to the remote server as much as possible subject to the satisfaction of the application's SLO.

IV.4.2 RunTime Phase

The deployment phase outputs the network address of the fog node where the application will be deployed and a list of execution modes as shown in Algorithm 4. This information is relayed to the client device, which then starts forwarding the application data to the fog node as per the execution mode list at every step. However, the execution mode list is based on the expected values of the network latencies, and hence can be different from the actual value.

The runtime phase minimizes the SLO violations caused due to inaccurate predictions by employing a robust mode selection strategy that updates the decision at any step based on feedback from previous steps. As shown in Figure 19, the *Controller* obtains sensor data

Algorithm 4 Server Selection

```
Require: User $u$ and its set of time steps $I(u) = \{1(u), 2(u), \ldots, L(u)\}$ defining the travel path; other information regarding the networks, the servers and their loads
Ensure: Server $s^*$ to deploy $u$ and a tentative execution mode vector $y^*[\ell] \in \{0,1\}$ for each time step $\ell \in I(u)$ during the travel
 1: Initialize $C_{\min} \leftarrow \infty$, $s^* \leftarrow \emptyset$, $y^*[\ell] \leftarrow 0\ \forall \ell \in I(u)$;
 2: for $i = 1$ to $k$ do
 3:     for $h = 1$ to $h_i$ do
 4:         $X^{cur} \leftarrow GetCurrentSystemMetrics(s_{i,h})$;
 5:         $X^{new} \leftarrow g_u(X^{cur})$;
 6:         $U \leftarrow GetListOfApplications(s_{i,h})$;
 7:         for each $v \in U$ do
 8:             $t_{process} \leftarrow GetProcessingTime(v)$;
 9:             $t_{isolation} \leftarrow GetIsolatedExecTime(v, s_{i,h})$;
10:             $t_{remote} \leftarrow t_{isolation} \cdot f_v(X^{new})$;
11:             $t^{SLO}_{network} \leftarrow GetPercentileLatency(v, s_{i,h})$;
12:             if $t_{process} + t_{remote} + t^{SLO}_{network} > \phi_v$ then
13:                 skip $s_{i,h}$;
14:         Initialize $y[\ell] \leftarrow 0\ \forall \ell \in I(u)$;
15:         $t_{process} \leftarrow GetProcessingTime(u)$;
16:         $t_{isolation} \leftarrow GetIsolatedExecTime(u, s_{i,h})$;
17:         $t_{remote} \leftarrow t_{isolation} \cdot f_u(X^{new})$;
18:         $L_{\max} \leftarrow \max_{v \in U} L(v)$;
19:         $T_{deploy} \leftarrow \max\left(0, L(u) - L_{\max}\right)$;
20:         $T_{transfer} \leftarrow state_u / b_i + ci_{u,s_{i,h}}$;
21:         $\ell' \leftarrow 1(u) + \lceil T_{transfer} / \phi_u \rceil$;
22:         for $\ell = \ell'$ to $L(u)$ do
23:             $t^{SLO}_{network} \leftarrow GetPercentileLatency(u, s_{i,h}, \ell)$;
24:             if $t_{process} + t_{remote} + t^{SLO}_{network} \leq \phi_u$ then
25:                 $y[\ell] \leftarrow 1$ // execute remotely ;
26:         $T_{user} \leftarrow Sum(y)$;
27:         $C \leftarrow \alpha_{i,h} \cdot T_{deploy} + \beta_{i,h} \cdot T_{transfer} + \kappa_u \cdot T_{user}$;
28:         if $C \leq C_{\min}$ then
29:             $C_{\min} \leftarrow C$;
30:             $s^* \leftarrow s_{i,h}$ and $y^* \leftarrow y$;
```

and selects appropriate mode for processing the data. The processed data is transformed and fed back to actuators. The *Controller* consists of a process, *Mode Selector* which is responsible for gathering sensor data, selecting appropriate mode and monitoring the timing deadline violations.

Mode Selector is modeled using Mealy machine, M_{sel} as shown in Figure 20. M_{sel} consists of 7 symbolic states with `Idle` being the initial state as shown in Figure 20. From `Idle` state, the state machine transitions to `SyncWithSLO` state after receiving *Start* event. The transition from `SyncWithSLO` is forced by the activation of *TimeOut*(t_2)

event that pushes the state machine into `GatheringSensorData` while emitting *Get-SensorData* event. This event activates a system level process to pull data from various sensors. If this task is not completed in t_3 secs, the $TimeOut(t_3)$ event forces the state machine back to `SyncWithSLO`. If the task of acquiring sensor data finishes before deadline, the state machine transitions to `SelectingMode` while producing *EvaluateConn* event.

EvaluateConn starts another asynchronous process, p, to acquire signal strength level and check the estimated execution mode in the list. If the execution mode is remote and signal strength is above the threshold, only then remote mode is selected at run time, which is signaled by this asynchronous task by emitting *SwitchToRemote* event, that enables M_{sel} to jump to `SendingData`. However, in the past if for the same access point, both the conditions were met and yet timing deadline had failed, then local mode will be selected as long as client device is connected to the same access point. After getting *SwitchToRemote* event, M_{sel}triggers data sending service by producing *SendData* event and moves to `SendingData`. The state machine waits for t_0 to receive the acknowledgment for the transmitted data by the server. If the acknowledgment does not arrive, it jumps to `ExecutingLocal`, whereas in the other case, the state machine transitions to `ExecutingRemote` and waits for the final response. If the response comes within t_4 secs, state machine jumps to `SyncWithSLO` and waits for the next cycle. However, if the response does not come within the deadline, an SLO violation is noted. If the asynchronous process, p, produces *SwitchToLocal* or does not emit any signal within time interval t_5 then M_{sel} jumps to `ExecutingLocal` from `SelectingMode`. While transitioning to `ExecutingLocal`, the state machine generates an event, *ProcessDataLocal* to trigger local data processing service. If the data is not processed with in t_1 secs, *TimeOut(*t_1*)* forces the state machine to move to `SyncWithSLO` and SLO violation is noted again. On the other hand if t_1 deadline is not violated, state machine also moves to back `SyncWithSLO` and waits till the next cycle starts.

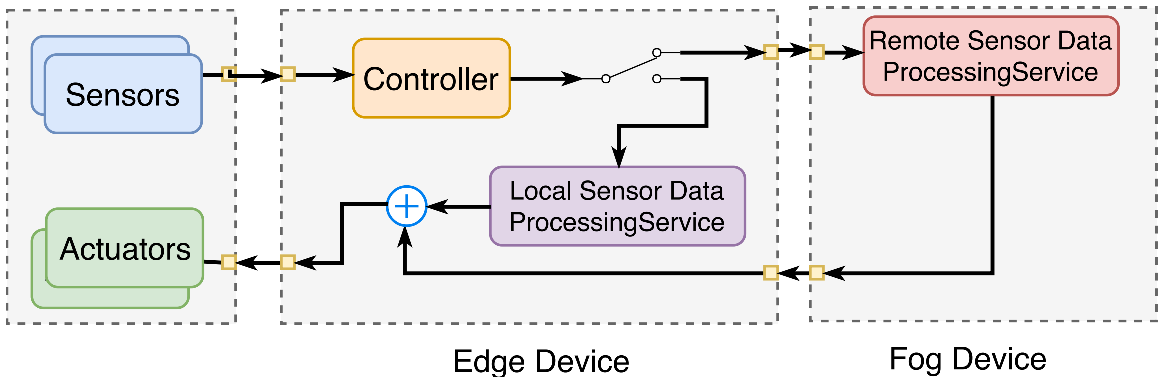

Sensors
Actuators
Controller
Local Sensor Data ProcessingService
Remote Sensor Data ProcessingService
Edge Device
Fog Device

Figure 19: Runtime Block Diagram

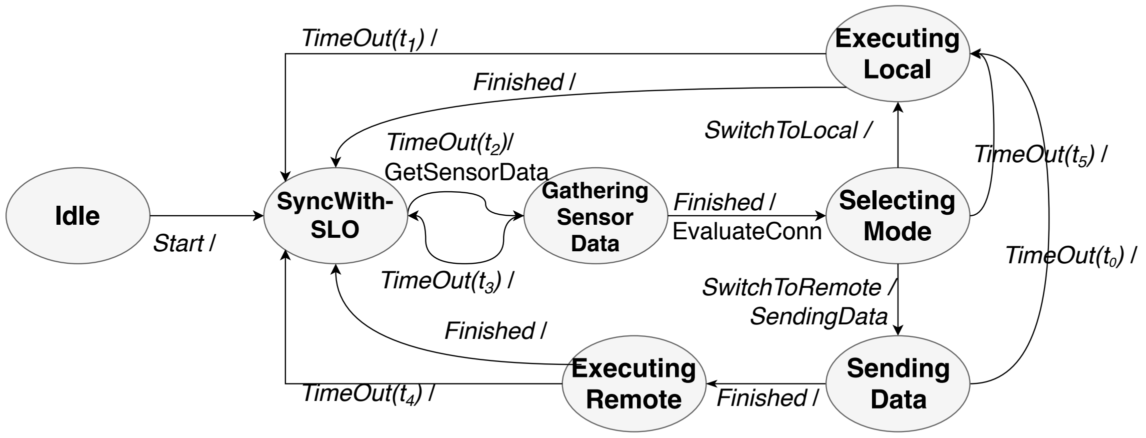


Figure 20: Mode Selector Mealy Machine

IV.5 Experimental Validation

We evaluate URMIL to answer the following questions:

- How effective is URMIL's execution time estimation on heterogeneous hardware including NUMA platforms? §IV.5.2.1
- How effective is URMIL's connectivity and network latency estimation considering user mobility? §IV.5.2.2
- How effective is URMIL in assuring SLOs? §IV.5.2.3
- How much energy can URMIL save for mobile user? §IV.5.2.3

- How does URMIL compare to other algorithms? §IV.5.2.3

IV.5.1 Experimental Setup

The goal of this chapter is to evaluate URMIL on real systems. However, we did not have access to geographically distributed wireless access points that also act as micro data centers. We overcome this limitation using two techniques.

First, we experimented over the limited area available within our lab. We created a Wireless Access Point using OpenWRT 15.05.1 with Raspberry Pi 2B. These WAPs were placed over different locations and operate at a channel frequency of 2.4 GHz. We then used the Android-based client described in Section IV.2.1. The Android client includes a Motorola Moto G4 Play phone having Qualcomm Snapdragon 410 processor with a Quad-core CPU and 2 GB of memory. The battery capacity is 2800 mAh and Android version is 6.0.1. It acts as both sensor for capturing frames and actuator for providing detected object as voice feedback. The device can be set to capture the video frames at variable frames per second (fps). We capture the frames at 2 fps as this is our SLO where the user expects an update within 500 ms if the detected object changes. The user walks at a brisk walking speed (expected to be close to 1.4 mps) in the region with the wireless access points while carrying the phone We apply URMIL during the duration and evaluate its performance.

Second, we emulate a large area containing 18 WAPs and 4 of which are MDCs. We experiment with different source and destination scenarios. We apply the technique in Section IV.4.1.2 to estimate the signal strength at different segments of the entire route. We then use three OpenWRT-RaspberryPi routers to emulate the signal strengths over the route by varying the transmit power of the WAPs at the handover points, i.e. where the signal strength is exceeds or drops below the threshold of -67 dBm. We achieve this by experimentally creating a mapping of received signal strength on the client device at the current location and varying transmit power of the WAP (0 to 30 dBm).

For the client device, we use Minnowboard Turbot which has a Quad-Core Intel Atom

E3845 processor with 2GB memory. The device runs Ubuntu 16.04.3 64-bit operating system and is connected to Creative VF0770 webcam and Panda Wireless PAU06 WiFi adapter on the available USB ports. In this case too, we capture the frames at 2 fps with a frame size of 224X224. In order to measure the energy consumption, we connect the Minnowboard power adapter to Watts Up Pro power meter. We measure the energy consumption when our application is not running and on an average the power is 3.37 Watts. We then run our application, and measure the power every second. By considering power difference in both the scenarios, we derive the energy consumption per step of size 500 ms.

Table 10: Server Architectures

Conf	sockets/cores/ threads/ GHz	L1/L2/L3 Cache(KB)	Mem Type/ MHz/ GB	Count
A	1/4/2/2.8	32/256/8192	DDR3/1066/6	1
B	1/4/2/2.93	32/256/8192	DDR3/1333/16	2
C	1/4/2/3.40	32/256/8192	DDR3/1600/8	1
D	1/4/2/2.8	32/256/8192	DDR3/1333/6	1
E	2/6/1/2.1	64/512/5118	DDR3/1333/32	6
F	2/6/1/2.4	32/256/15360	DDR4/2400/64	1
G	2/8/1/2.1	32/256/20480	DDR4/2400/32	2
H	2/10/1/2.4	32/256/25600	DDR4/2400/64	1

For the server application, we use real time object detection algorithms MobileNet and Inception V3. For the local mode execution, these algorithms run on the client device. The Android device runs Tensorflow Light 1.7.1. The Linux-based client device runs Docker and the server application is containerized. We use this model so that we can easily port the application across the platforms and Docker provides near native performance [57]. We use Ubuntu 16.04.3 containers with Keras 2.1.2 and Tensorflow 1.4.1.

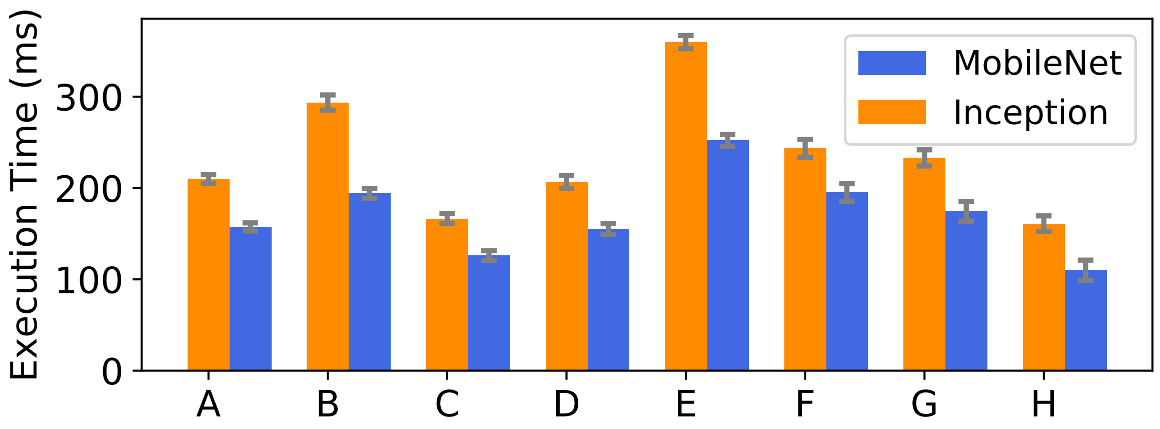
For the deployment, we use heterogeneous hardware configurations shown in Table 10. The servers have different number of processors, cores and threads. Configurations G,H and I also support hyper-threads but we disabled them in our cloud. We randomly select from a uniform distribution of the 16 servers specified in Table 10 and assign 4 of them to each MDC. In addition, for each server, the interference load and their profiles are selected randomly such that the servers have medium to high load without any resource overcommitment which is the usual load in data centers [21]. Since the MDCs are connected to each other on LAN in our experimental setup, but we need WAN latencies in order to experiment with multi-hop latencies. We experimented with http://www.speedtest.net/ on intra-city servers for ping latencies and found 32.6 ms as the average latencies. So, we added 32.6 ms ping latency with a 3 ms deviation between WAP using netem network emulator on the WAP to WAP ethernet communication links.

The Docker guest application has been assigned 2 GB memory and 4 pinned cores. For our experimentations, we use server application that listen on TCP ports for receiving the images and sending the response. Please note, our framework is independent of communication mechanism as long as we have an accurate measure of network latency for the size of data transferred. Thus, we could also support UDP (not reliable) and HTTP (higher latency). The size of a typical frame in our experiment is 30 KB. For the co-located workload that cause performance interference, we use 6 different test applications from the Phoronix test suite (http://www.phoronix-test-suite.com/) which were either CPU, memory or disk intensive and Tensorflow prediction algorithms which represent other latency sensitive applications.

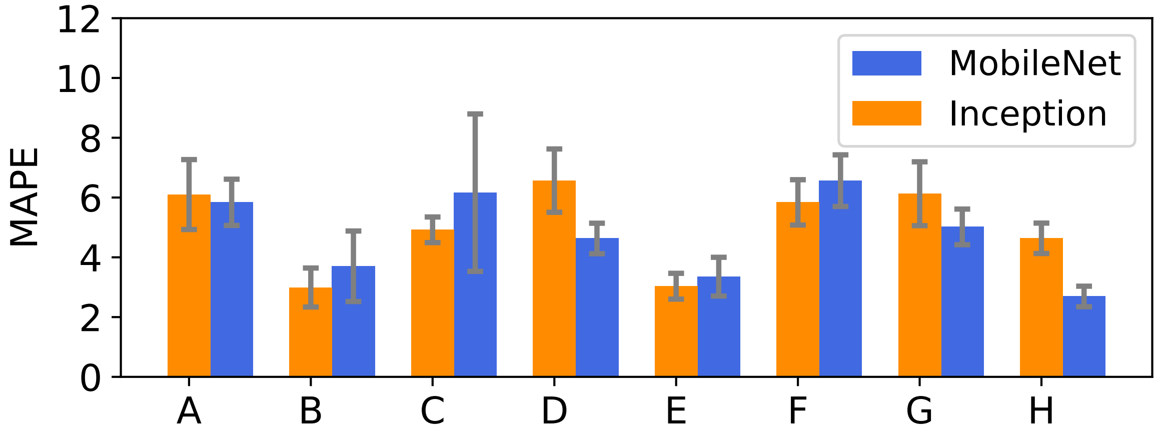
IV.5.2 Evaluations

IV.5.2.1 Evaluating the Performance Estimation Model

In Equation (IV.3), there are three main components, $t_{local}(u)$, $t_{remote}(u, s_{i,h})$ and $t_{network}(u, s_{i,h}, \ell)$ and we need accurate estimates of all three at deployment time such that we could adhere to SLO requirements. $t_{local}(u)$ has negligible variations as long as the client device is running only the target application u which is a fair assumption for the mobile devices. For the Linux client device the execution time for processing a 224X224 frame, the measured



(a) Execution time in isolation

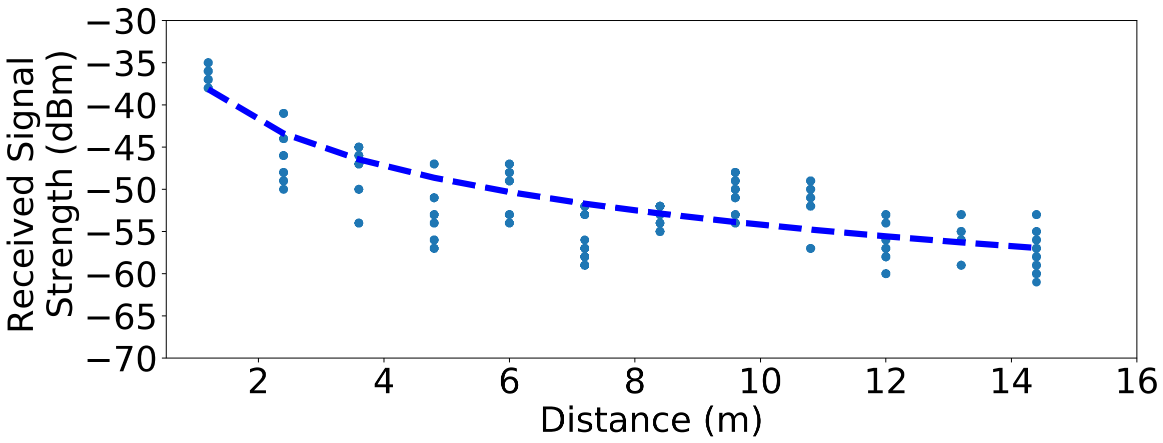


(b) Mean Absolute Percentage Error

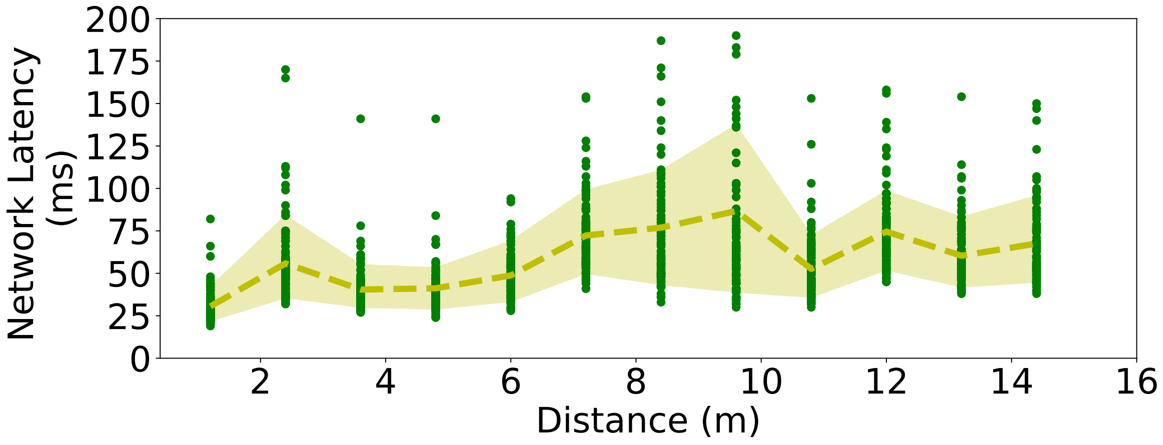
Figure 21: Performance Estimation Model Evaluations

execution time for MobileNet and Inception V3 were 434 ms mean with 8.6 ms standard deviation and 698.6 ms mean with 12.9 ms standard deviations respectively.

For $t_{remote}(u, s_{i,h})$, in addition to hardware type w, we also consider the server load. We first measured $t_{isolation}(u, w)$ for each hardware type w as shown in Figure 21a. We observe that the CPU speed, memory and cache bandwidth and use of hyper-threads instead of physical cores play significant role among other factors in application performance. Thus, the use of per hardware configuration performance model is a key requirement. In addition, we developed the performance interference profile using gradient tree boosting regression model from the enhanced version of the tool we used [149] and Figure 21b shows the estimation error on different hardwares, the hardware configurations E-H have NUMA-enabled and F-H support Intel CMT. We observe that the execution time estimation error is well within 10% and thus can be used in our response time estimations.



(a) Received Signal Strength vs Distance



(b) Response Time vs Distance

Figure 22: Signal Strength and Network Latency Variations with Distances

IV.5.2.2 Evaluating Network Connectivity and Latency Estimation

Next, we evaluate URMILA's network latency estimation module in order to calculate $t_{network}(u, s_{i,h}, \ell)$. From equation (IV.2), there are two main components to it last-hop latency, $t_{u,ap(u,\ell)}$ and WAN latency, $t_{ap(u,\ell),ap_i}$. $t_{ap(u,\ell),ap_i}$ remains stable over a duration of time [20, 159] sufficient for URMILA scenarios and we emulate those as described in Section IV.5.1. Thus, we are left with $t_{u,ap(u,\ell)}$. As received signal strength is a key factor for last hop latency, we determine γ for Equation (IV.15) for a typical access point described in section IV.5.1 for indoor environment of our lab. We used the Android device to measure signal strength and network latency for the used data transfer size. Figure 22a shows the results where we found the γ to be 1.74, inline with indoor estimates as described in Section IV.4.1.2. In Figure 22b, affirms our assertion that network latency remains constant within a range of varying received signal strength.

Next, we measure network latency for five different routes on our selected campus area with 18 emulated access points. We chose $\gamma = 2$ for outdoors and generated signal strengths for the entire routes. Based on these values, we setup the WAPs such that the client device experiences access point handovers and regions with no connectivity. Figure 23 shows the results for different routes. The shaded areas on the plots show the regions with no network connectivity and regions with different colors show connectivity to different WAPs. There are gaps in latency values which indicate the client device is performing handover to the access point. We observer from these plots that even though the mean value are low for latencies when connected to wireless network, there are large deviations. Hence, for ensuring SLOs, we need to use the required SLO percentile value from our database of network latency on user's route as we described in Algorithm 4.

IV.5.2.3 Evaluating URMILA's Server Selection

In this section, we evaluate how effective is URMILA's server selection technique in ensuring that the SLOs are met and costs are minimized. We evaluate the system for the five routes described above and set 4 of the 18 available access points as MDCs and assigned servers as described in IV.5.1. We compare URMILA against different mechanisms. The first approach is when we perform everything locally (*Local*). Next, we compare URMILA

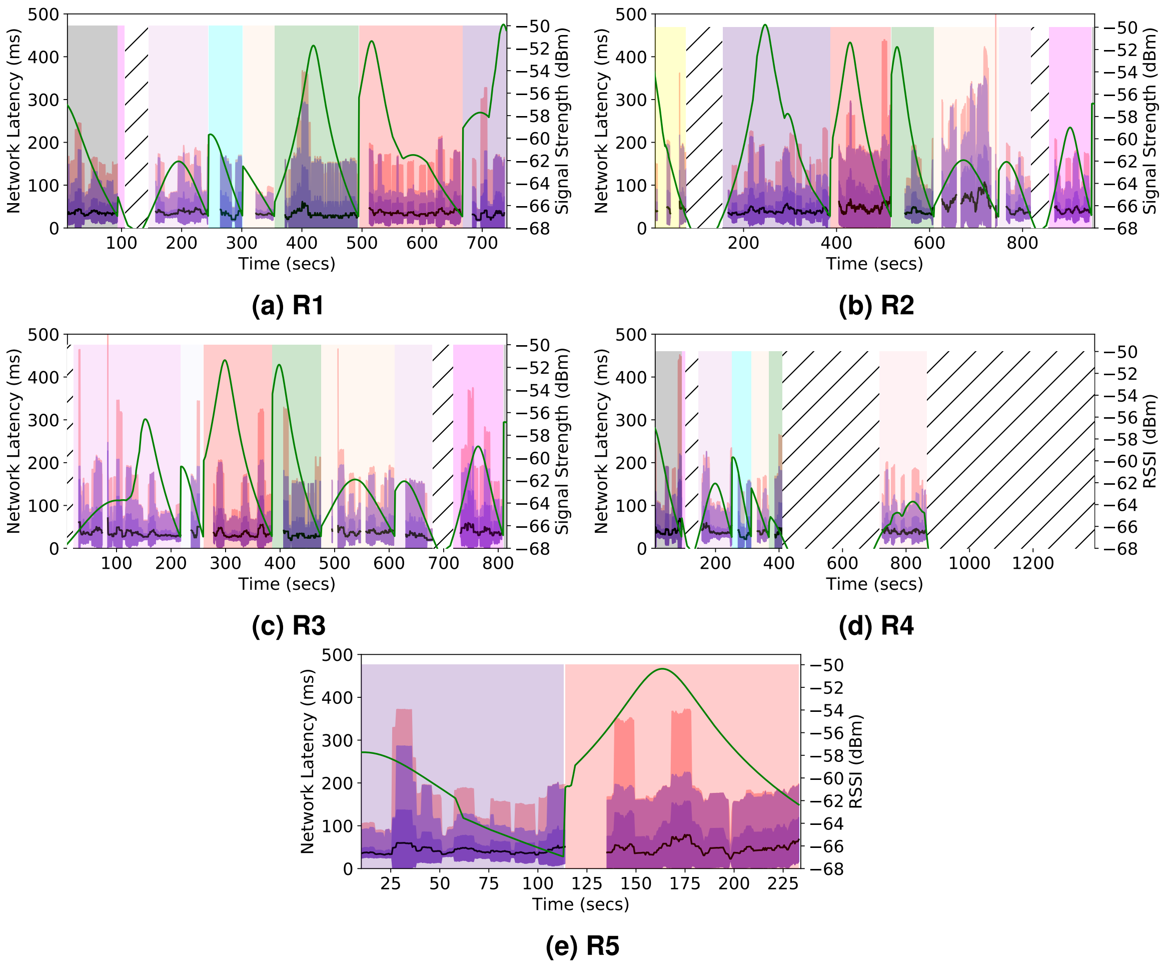


Figure 23: Observed mean, standard deviation, 95th and 99th percentile network latencies and expected received signal strengths on different emulated routes

against two commonly used techniques, namely maximum network coverage (*Max Coverage*) and least loaded server selection (*Least Loaded*). As we have already measured the efficacy of NUMA-aware deployment in figure 18, we employ NUMA-awareness in all the experimental scenarios.

For this set of experiments, we keep the deployment (Equation IV.4) and transfer (Equation IV.5) costs constant in our Algorithm 4 for all the scenarios. We also set the required SLO at 95th of desired response time of 500 ms (2 fps). We then optimize for energy consumption (Equation IV.7) while meeting the constraints (Equations IV.9-IV.14). From Figures 24 and 25 , we observe that if we run higher accuracy Inception algorithm as target

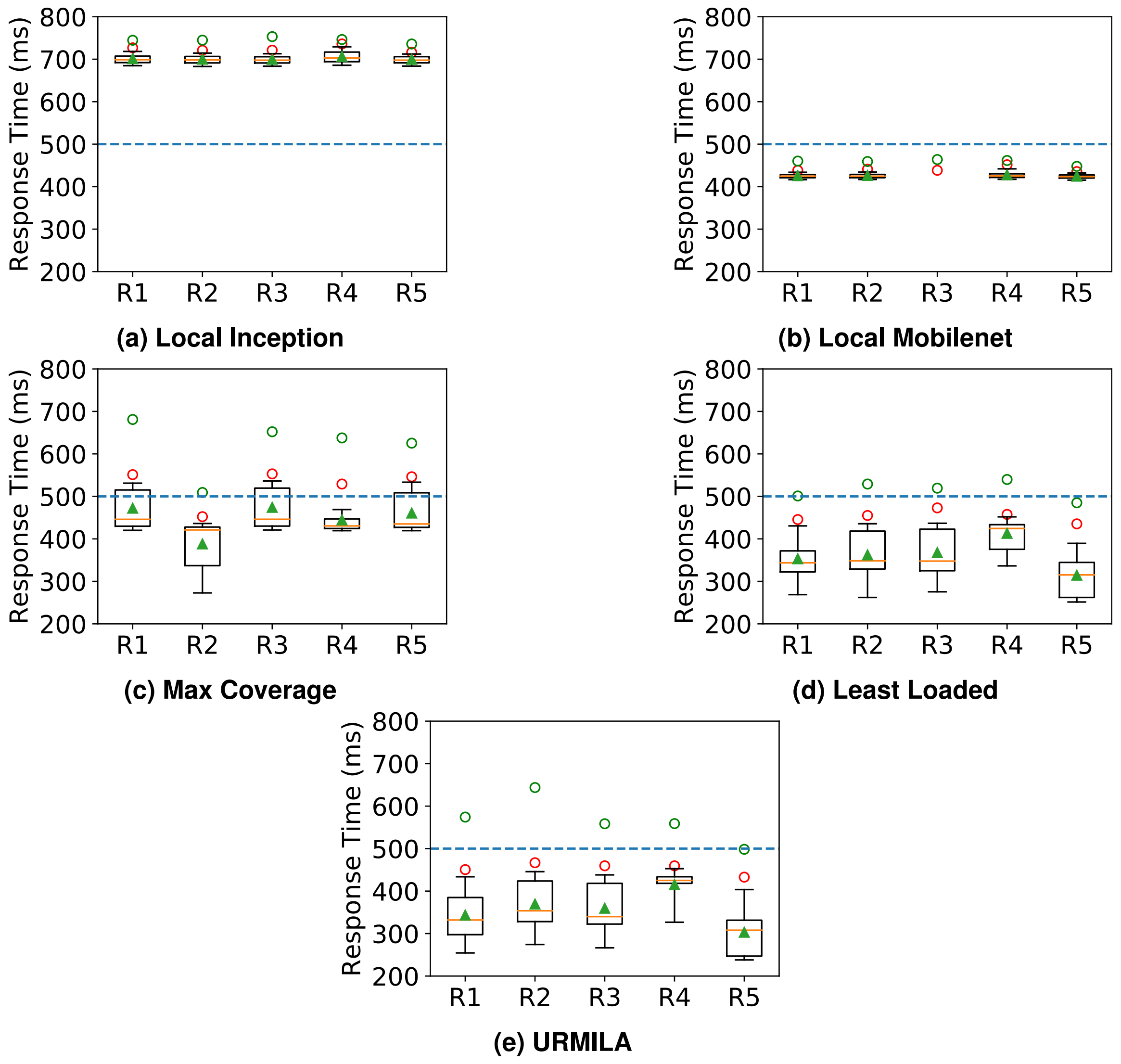


Figure 24: Response time for different techniques on the routes.

application, *Local* mode always misses the deadline of 2 fps, on the other hand for lower accuracy MobileNet always meets the deadline but wastes energy. Rest of the experiments were all done with Inception V3. The *Max Coverage* algorithm performed worse than URMILA for energy consumption and on 4 out of 5 routes for response time. For these experiments *Least loaded* performs at par with URMILA. Please note as URMILA considers both the server load and and network coverage, it will perform at least at par to the other two techniques for assuring SLOs.

We now demonstrate the scenario when URMILA performs better that *Least loaded*. In

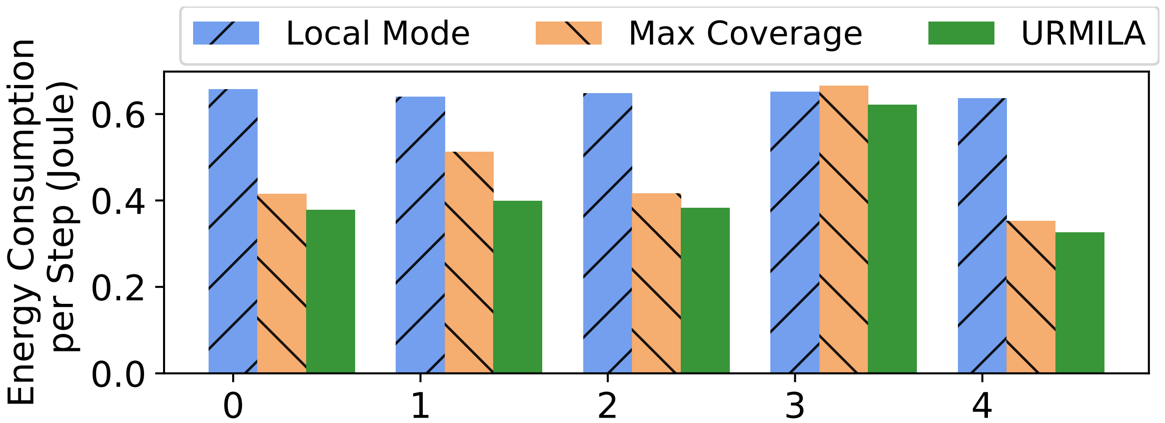


Figure 25: Energy Consumption Comparison

our current experimental setup, we considered there is similar latencies between the access points $t_{ap(u,\ell),ap_i}$ and for the last hop, $t_{u,ap(u,\ell)}$ channel utilization and connected users are less. However, this is not usually the case. Thus, we introduce use a latency value of 100.0 ms with 10% deviation for some of the access points. In real deployments, URMILA will be aware of this latency by WAP to WAP measurements. Thus, as depicted in Figure 26, for *Least Loaded*, SLOs will be violated even for best performing server due to the ignorance about the network communication delay. However, URMILA's robust runtime component is aware of the deployment plan and performs execution locally for the WAPs that cannot meet the constraints.

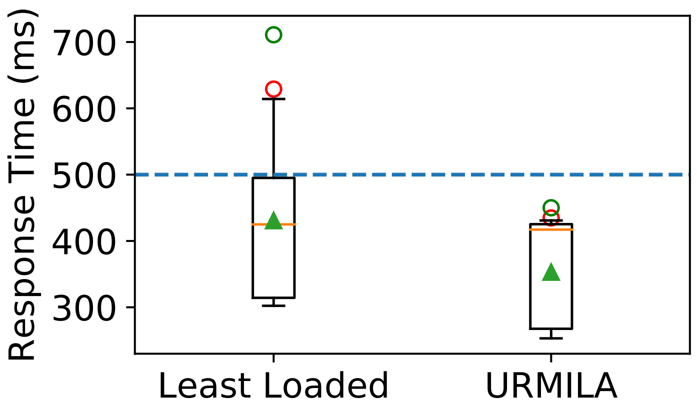


Figure 26: Response time comparison for route R5 when one of the WAP is experiencing larger latency

In the above experiments, we considered that there is sufficient gap between when the user requests the service and when she actually needs it. However, this may not be true and we need to consider the transfer and initialization costs of Equation IV.5. We setup Docker private registry and shaped the network bandwidth such that we could do the measurements for image overlays being transferred of different sizes. Table 11 depicts the same.

Table 11: Transfer and Initialization Cost Measurements

Instance Type	Size (MB)	Duration at 10 Mbps	Duration at 1 Mbps
Cached	-	13.2s	13.46s
Overlay 1	111	31.6s	127.08s
Overlay 2	440	50.26s	261.87s

IV.6 Discussion

We discuss unresolved problems and additional research opportunities in this section.

Last Hop latency: In our current approach, for un-profiled routes, we only considered received signal strength for wireless network latency estimation. However, channel utilization and connected users play a significant role in latency variations. To overcome this potential inaccuracy, we can collect these metrics from WAPs, but this will require access to their data. Other option is to use a predictive approach based on data collected for other profiled routes. We plan to explore this dimension in future.

Constant speed mobility: For the user mobility, we considered constant speed mobility, however, the user can deviate from the ideal route and stop in between. This will jeopardize our initial plan. We account for this in our server allocation, but, the runtime algorithm can further be improved to intelligently adjust the route plan based on current dynamics.

Overhead: URMILA has cost for both client device and the service provider. The service provider has to collectd metrics on each server. With monitoring tools that we used [149] this overhead is $<1\%$. We also need to maintain a database of performance metrics at each MDC and the *gm* needs to perform learning. In addition, the cost of profiling each new application can be significant, $\approx$ 30 mins. However, this is a one time cost and is required for overcoming performance interference estimation. On the client device, we made a conscience effort to not to use GPS coordinates while the user is mobile. This is because GPS has significant energy overhead and we did not want our application to be limited to navigational applications. In addition, turning on wireless and handovers are expensive. However, most mobile devices have their wireless service turned on these days, so we do not consider it as additional cost.

Applicability: Other than the image processing applications like cognitive assistance evaluated in this work, we could apply URMILA in cloud gaming (such as Pokemon GO), 3D modeling, graphics rendering etc. We could apply URMILA for energy efficient route selection and navigation. For that, we can easily modify Algorithm 4 to add another loop at line number 3 to loop through routes and find the most energy efficient route. Also, instead of using standard wireless handover policy we could design our own policy which will be more energy efficient.

Serverless Computing: Since we target containerized stateless applications, portable across the resource layers, we could potentially make our solution apt for serverless computing, wherein the same containers are shared by multiple users and the application scale as the workload varies, and are highly available.

Future Direction: Apart from what we discussed, our solution can be enhanced by controlling frame rates based on the user needs and location. We considered monolithic applications, we could allocate services with multiple components that are deployed across the spectrum optimally. Also, we could apply advanced energy optimization techniques such as [170] which consider tail state energy consumption on mobile devices to further improve our results. In future, we could address concerns related to trust, privacy, billing, fault tolerance and workload variations.

IV.7 Related Work

IV.7.1 Mobility-aware Resource Management

MOBaaS [86] is a mobile and bandwidth prediction service by leveraging dynamic Bayesian network. Sousa et al. [156] applied MOBaaS to enhance the follow-me cloud (FMC) model First, they perform mobility and bandwidth prediction with MoBaaS and then apply multiple attribute decision algorithm to place services at a suitable mobile service provider location However, the service needs a history of mobility pattern build by monitoring the users.

MuSIC defines applications as location-time workflows and optimizes their quality of service expressed as power of the mobile device, network delay and price. Nonetheless, the technique has not been validated on an actual system as the results are from simulations. In addition, the variations in network pattern are assumed without applying any prediction/ estimation methodology.

Luiz et al. [29] consider different classes of mobile applications and apply three scheduling strategies on fog resources. Wang et al. [168] account for user mobility and provide both offline and online solution for deploying service instances considering a look-ahead time-window. Both the approach do not consider edge resource for optimization which we do. ME-VoLTE [25] is an approach to offload video encoding from mobile devices to cloud for reducing energy consumption. However, the approach does not consider latency issues when offloading.

IV.7.2 Performance Interference and NUMA-aware Resource Optimization

There has been several works that account for performance interference during server selection. Bubble-Flux [180] is dynamic interference measurement framework that performs online QoS management for providing guarantees while maximizing server utilization. It uses a dynamic memory bubble to profile the sensitivity and contentiousness of the target application by pausing other co-located applications. Freeze'nSense [85] is another approach that performs a short duration freezing of interfering co-located tasks. The advantage of an online solution is that a priori knowledge of the target application is not required and it does not need additional hardware resources for benchmarking. However, pausing of co-located application is not desirable and in several cases not even possible. DeepDive [129] identifies the performance interference profile by cloning the target VM and benchmarking it when QoS violations are encountered. This is an expensive operation to be employed in a production data center.

Paragon is a heterogeneity and interference-aware data center scheduler applies analytical techniques to reduce the benchmarking workload. The paper identifies sources of interferences from a number of hardware sub-systems and micro-benchmarks them for the interference profile. SMiTe [188] considers a wide array of metrics and uses hardware performance monitoring units (PMUs) for developing interference profiling. However, these approaches do not account for virtualization based metrics that we consider. In addition, these approaches do not account for complexities arising due to NUMA architecture.

Rao et al. [136] proposed a framework that estimates the performance of applications running inside virtual machines on NUMA nodes. They modified the Xen hypervisor for NUMA awareness such that threads are dynamically migrated for minimizing the uncore penalty. We believe the frequent migration of threads is detrimental to the performance of the whole system. Besides, the approach does not consider other shared resources such as network and disk. The modification required for guest domain to communicate with the scheduler may not be a feasible solution. Liu et al. [106] identify four sources of contention on NUMA systems and presented two performance optimization strategies that involves VM memory allocation and page fault handling. Nonetheless, the metrics used for interference measurement do not provide precise performance degradation predictions, in addition to the need to modify the guest OS.

IV.8 Concluding Remarks

Although fog/edge computing have enabled low latency edge-centric applications by eliminating the need to reach the centralized cloud, solving the performance interference problem for fog resources is even harder than traditional cloud data centers. User mobility amplifies the problem further since choosing the right fog device becomes critical. Executing the applications at all times on the edge devices is not an alternative either due to their severe battery constraints. This chapter presents URMILA which is a resource management middleware to address these issues and adaptively uses edge and fog resources making trade-offs while satisfying SLOs for applications.

CHAPTER V

PERFORMANCE INTERFERENCE-AWARE VERTICAL ELASTICITY FOR CLOUD-HOSTED LATENCY-SENSITIVE APPLICATIONS

V.1 Motivation

Elastic auto-scaling is a hallmark resource management property of cloud computing which to date has focused mostly on horizontal scaling of resources wherein applications are designed to exploit distributed resources by scaling them across multiple servers using multiple instances of the application. However, spawning new virtual machines on-demand for horizontal scaling requires initialization periods that can last several minutes and the spawned instances must adhere to the cloud provider-defined instance types. This may lead to quality of service (QoS) violations (and hence violation of service level objectives – SLOs) in applications. To avoid QoS violations and to account for workload variations, cloud-hosted latency-sensitive applications, such as online gaming, cognitive assistance, and online video streaming, are often assigned more horizontal resources than they need [108]. Unfortunately, maintaining a pool of pre-spawned resources and application instances often will waste resources.

Considering the recent and emerging advances in hardware including the ever growing capacity of servers and the advent of rack-scale computing [3], vertical elasticity has become a promising area for dynamic scaling of applications and also a first choice for elastic scaling before horizontal scaling is attempted [97]. Vertical elasticity is the ability to dynamically resize applications residing in containers or virtual machines [10, 83, 157]. It not only allows fine-grained assignment of resources to the application but also enables traditional applications that were not designed for purposes of horizontal scaling, to scale vertically according to its changing resource demands stemming from workload changes.

Vertical elasticity for latency-sensitive applications is often realized by co-locating them with batch applications such that they have some slack available to scale up or down on-demand and the resources are not wasted because the batch applications can utilize the remaining resources. Cloud service providers use virtual machine or container technologies to host multiple applications on a single physical server. Each latency-sensitive application has its own configuration and dynamically allocated resources that fulfill its application-specific demands and requirements.

Despite these trends, performance interference [35, 49] between the co-located applications is known to adversely impact QoS properties and SLOs of applications [124]. Dynamic service demands and workload profiles further amplify the challenges for cloud service providers in (de)allocating resources on demand to satisfy SLOs while minimizing the cost [185]. This problem becomes even harder to address for latency-sensitive, cloud-hosted applications, which we focus on in our work. Therefore, any solution to address these challenges necessitates an approach that accounts for the workload variability and the performance interference due to co-location of applications.

To that end, we present a data-driven and predictive vertical auto-scaling framework which models the runtime behavior and characteristics of the cloud infrastructure, and controls the resource allocation adaptively at runtime for different classes of co-located workloads. Concretely, our approach uses a multi-step process where we first apply Gaussian Processes (GP) [139]-based machine learning algorithm to learn the application workload pattern which is used to forecast the dynamic workload. Next, we use K-Means [73] to cluster the system level metrics that reflect different performance interference levels of co-located workloads. Finally, we apply another GP model to learn the online performance of the latency-sensitive application using the measured data, which in turn provides real-time predictive analysis of the application performance. Our framework uses Docker container-based application deployment and control infrastructure that leverages the online predictive model in order to overcome run-time variations in workload and account for performance interference. We also periodically update the models in online fashion such that the dynamics of the target application workload and co-located applications are reflected in our predictions. Compared to our approach, prior work including ours [36] that have used machine learning-based models to address performance interference related resource management issues have either focused on horizontal elasticity or developed offline models but we have not seen prior work that provides both an online predictive model and which addresses vertical elasticity.

In summary, we make the following contributions in this chapter:

- We present an approach using K-Means clustering and Gaussian Processes-based performance modeling framework that predicts the latency sensitive application's performance under varying workload and performance interference from co-located applications.

- We describe the architecture of a resource control framework for vertical elasticity built upon the Docker container engine and Docker Swarm cluster management platform that allows resource scaling and checkpointing of applications.

- We present experimental results to validate our system.

The rest of the chapter is organized as follows: Section V.2 compares our work with related research; Section V.3 presents details of our approach; Section V.4 presents experimental evaluations; and finally Section V.5 provides concluding remarks alluding to future work.

V.2 Related Work

We surveyed literature that focus on resource allocation strategies in cloud computing along the dimensions of workload prediction, performance interference, and vertical elasticity, all of which are key pillars of our research. We provide a sampling of prior work along these dimensions and compare and contrast our work with them.

Related research based on Workload Prediction:

To model different classes of workloads and applications, the Dejavu [166] framework computes and maps the workload signature to resource allocation decisions, and then periodically clusters the workload signature using K-means algorithm storing known workload patterns in the cache for rapid resource allocation. A proactive online model for resource demand estimation techniques is proposed in K-scope [184] that utilizes a Kalman filter to estimate the queuing-network model parameter in an online fashion. K-scope has been extended for the dependable compute cloud (DC2) framework [63], which automatically uses the learned queuing-network model to predict the system performance. Likewise, [77, 98], propose an adaptive controller using Kalman filtering for dynamic allocation of CPU resources based on the fluctuating workloads without any prior information. In our prior work [141], we proposed a workload prediction model using autoregressive moving average method (ARMA). These works are based on linear models for QoS modeling; in contrast, cloud dynamics often illustrate nonlinear characteristics and incur significant uncertainty.

In [123], a non-linear, predictive controller is proposed to forecast workload using a support vector machine regression model. In [157], the authors propose a proactive online model based on monitoring application- and hypervisor-level performance metrics and a layer performance queuing model. The parameters of the performance models are updated online in a regular interval using resource demand estimation techniques. Ali et al. [127] developed an autonomic prediction suite that decides the time-series prediction model (using Neural Network or Support Vector Machine) based on incoming workload pattern. In contrast, our work uses a Gaussian Process (GP)-based model because it has relatively small number of hyper parameters so that the learning process can be achieved efficiently in an online fashion. Although some efforts [8, 147] use Gaussian processes to model and predict the query or workload performance for database appliances, they do not incorporate performance interference in their model. While most of the strategies for performance and resource management are rule-based and have static or dynamic threshold-based triggers [10], our system uses a proactive approach using GPs to learn parameters dynamically and perform timely resource adjustments for latency-sensitive applications.

Related research based on Vertical Elasticity:

Vertical elasticity adds more flexibility since it eliminates the overhead in booting up a new machine while guaranteeing that the state of the application will not be lost [10]. Several approaches are proposed to scale the CPU resources [97, 151]. Kalyvianaki et al. [83] proposed a Kalman filter-based feedback controller to dynamically adjust the CPU allocations of multi-tier virtualized servers based on past utilization. A vertical auto-scaler [183] is proposed to allocate CPU cores dynamically for CPU-intensive applications to meet their SLOs in the context of varying workloads. The authors offer a linear prediction model on top of the Xen Hypervisor to plug more CPU cores (hot-plugging) and tune virtual CPU power to provide the vertical scaling control. Controlling of CPU shares of a container based on the Completely Fair Scheduler is proposed in [122]. Vertical autoscaling techniques based on a discrete-time feedback controller for Containerized Cloud Applications are proposed in ELASTICDOCKER [10] that uses an approach to scale up and down both CPU and memory of Docker container based on resource demand. However, their decision triggering approach is reactive. In contrast, we use a more efficient Gaussian-based proactive method to trigger the scaling of resources.

Related research based on Performance Interference:

Prior research shows that model-based strategies are a promising approach which allow the cloud providers to predict the performance of running VMs or containers and to make efficient optimization decisions. DeepDive [130] is proposed to identify and manage performance interference between co-located VMs on the same physical environment. Q-Clouds [124] is a QoS framework which utilizes a feedback mechanism to model the interference interaction.

DejaVu [166] creates an interference index by comparing the estimated and actual performance. Based on matching the trained profile, it then provisions resources to meet application SLOs. Paragon [49] also classifies applications for interference and predicts which application will probably interfere co-located application performance for heterogeneous hardware based on collaborative filtering techniques. Quasar [50] extends the Paragon framework by tailoring the classification types with different workload types to improve their model. Unlike our approach, these efforts do not prioritize latency-sensitive applications due to interference from their co-located applications.

Stay-Away [135] framework mitigates the effects of performance interference on high priority applications when co-located with batch applications. If QoS of the high priority task is predicted to suffer, they throttle the batch application based on results of estimation. Bubble-flux [180] produces interference estimation by considering co-located applications on servers by continuously monitoring the QoS of latency-sensitive application and controlling the execution of batch jobs accordingly based on profiling. Heracles [109], which is a feedback controller, reduces performance interference by enabling the safe co-location of latency-sensitive applications and best-effort tasks while guaranteeing the QoS for the latency-critical application. Unlike these efforts, our GP-based model predicts the future latency in online fashion, and tunes the parameters on each iteration.

Our prior work [36] designed a performance interference-aware resource management framework that benchmarks the applications hosted on VMs. The server's performance interference level is then estimated using neural network-based regression techniques. However, hardware heterogeneity and every application's performance is not considered in the model. In another prior work [149], we benchmarked a latency-sensitive application with co-located applications on different hardware and develop its interference profile. The performance of the new application is predicted based on its interference profile which is obtained using estimators of an existing application for the same hardware specifications.

A safe co-location strategy is decided by looking up the profile, however, we did not consider dynamic vertical scaling. The the current paper, we determine the vertical scaling strategy based on our online GP prediction model.

V.3 System Design for Proactive Vertical Elasticity

This section provides details of our solution for interference-aware vertical elasticity to support SLOs of latency-sensitive applications that are co-located with batch applications.

V.3.1 System Model

We target cloud data centers comprising multiple servers that host both latency-sensitive and batch-processing applications. The latency-sensitive applications have higher priority and need assurance of their SLOs while the provider also needs to ensure the remaining resources are utilized by the co-located batch processing applications such that there is minimal to no resource wastage. Docker is a container platform for application hosting with a growing user base with cloud service providers providing their own Docker deployment services, such as Amazon EC2 Container Service, Azure Container Service and Google Container Engine. We target cloud platforms hosting Docker containers natively. However, our solution can apply to any virtualized platform that allows rapid resource reconfigurability that is needed for vertical elasticity.

V.3.2 Problem Statement and Solution Approach

Cloud providers support multi tenancy by deploying applications in virtual machines or containers in order to provide a certain level of isolation. Moreover, to assure bounded latencies, cloud-hosted latency-sensitive applications are often assigned dedicated cores with the use of CPU core pinning [132, 133] which is the ability to run a specific virtual CPU on a specific physical core, thereby increasing cache utilization and reducing processor switches. Despite all these strategies, multi tenancy still leads to performance interference causing degradation in performance for latency-sensitive applications which can be particularly severe in the case of tail latency [101], i.e., 90th, 95th, 99th or similar percentile latency values. This is due to the presence of non-partitionable or difficult-to-partition resources such as caches, prefetchers, memory controllers, translation look-aside buffers (TLBs), and disks among others. The workloads for each such resource are referred to as *sources of interference* (SoIs) [49]. A SoI helps in identifying the interference that an application can tolerate for that resource before SLO violation occurs. Exacerbating the problem is the fact that different applications incur different levels of sensitivity to co-located workloads [95, 188]. Figure 27 depicts an exemplar where the performance of a web search application is shown deteriorating significantly because of the presence of varying interference workload, even when they do not share the CPU cores. We observe that the 90th percentile latency is more than 51% worse when performance interference is present.

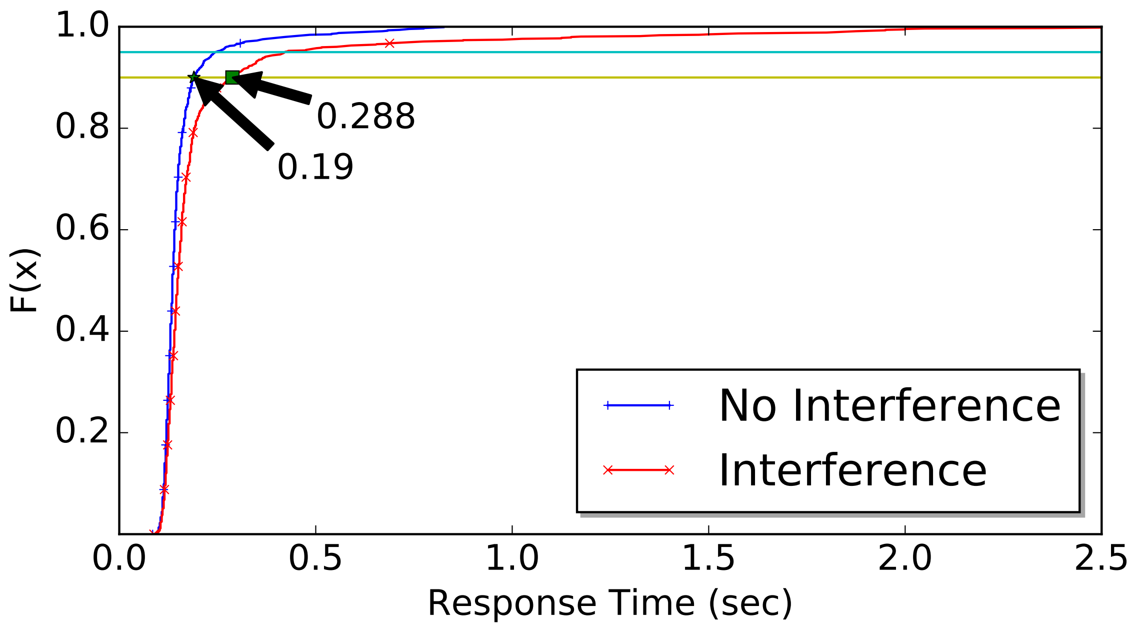


Figure 27: CDF of Response Time for CloudSuite WebSearch with 6 Cores

Addressing these concerns warrants a solution that accounts for performance degradation due to interference and the workload variations so that SLO violations can be minimized while also minimizing cost to the cloud provider. Advances in container technologies such as Docker offer promise in allowing us the control and allocation of resources to rapidly adapt to workload variations and co-location interference. Prior efforts, such as Heracles [109], account for performance interference while also rapidly scaling the application for workload variations. They use feedback controllers where the best effort batch jobs are disabled if the demand from the latency-sensitive applications increases. However, the disabled jobs are still kept in memory thereby limiting the available resources for latency sensitive applications. To overcome this limitation, one can either checkpoint or migrate the co-located batch applications and restore them once the demand from latency sensitive application reduces. Solutions such as [146, 179, 181] take the checkpointing approach. However, checkpointing and migration can take long durations, especially for memory-intensive applications where a large amount of state needs to be saved. Moreover, during this phase any additional resource allocation will not result in better performance. Reactive approaches also require very high rate of performance metric collection in order to react quickly to workload variations.

Consequently, an approach that can forecast the workload to proactively perform vertical scaling while accounting for interference imposed by co-located workloads is needed. Further, as the workload and interference level can vary dynamically, the solution should be able to forecast the required resources in an online fashion. Hence, we propose a model-predictive approach for vertical scaling which predicts the needed resources while accounting for workload variations at different levels of performance interference due to co-located workload. We use Gaussian Processes (GPs) to model the latency variations due to varying workloads forecasted using GPs We chose GPs over other learning techniques because they have relatively small number of hyper parameters. So the learning process can be achieved efficiently in online fashion. In addition, they are probabilistic models thus allowing us to model the uncertainty in the system while also being able to model nonlinear behavior of the underlying system.

V.3.3 Technique for Model-based Prediction

To optimize the resource utilization while maintaining the SLO guarantees for latency-sensitive applications, we need an accurate and online performance model of the latency-sensitive applications. There is also a need for an online model since the latency-sensitive application workload can vary dynamically. Moreover, the batch applications co-hosted on the same server can vary in their amount and nature of resource utilization. Finally, each application also incurs its own performance interference sensitivity to the co-located workload [95, 116, 176, 188]. Thus, the core component of our framework is the model predictor for which we have developed a per-application performance model in an online fashion that helps to rapidly adapt to changing levels of workload and co-location patterns.

We use Gaussian Processes (GPs) to model the performance of the latency-sensitive applications. GPs are non-parametric probabilistic models that utilize the observed data to represent the behavior of the underlying system [139]. A function $y = f(\mathbf{x}) : \mathbf{x} \in \mathbb{R}^d$ modeled by a GP can be expressed as: $f(\mathbf{x}) \sim \mathscr{GP}(m(\mathbf{x}), k(\mathbf{x},\mathbf{x}))$ where $m(\mathbf{x})$, $k(\mathbf{x},\mathbf{x})$ is the mean function and the covariance functions of the GP model. Typically, a zero mean function and squared exponential (SE) covariance kernel are used for their expressiveness.

Given the training data with n data points ($\mathscr{D} = \{(\mathbf{x}_i, y_i) | i = 1, n\}$) where $\mathbf{x}_i$ is the training inputs and y_i is the training outputs, we train the GP model to identify their hyperparameters Θ so that they best represent the training data. In other words, we optimize the hyperparameters ($\hat{\Theta}$) of the GP model to maximize the log likelihood, i.e., $\hat{\Theta} = \arg\max_{\Theta} \log p(\mathbf{y}|\Theta, \mathscr{D})$ using conjugate gradients [113, 139] optimization algorithm. We define the test input at which we want to predict the model output as $\mathbf{x}_*$. Hence, the predicted output of the GP model (y_*) can be achieved by evaluating the GP posterior distribution $p(y_*|\mathbf{x}_*, \mathbf{X}, \mathbf{y})$ which is a conditional Gaussian distribution with a mean and a variance evaluated by:

$$\begin{aligned} \mathbb{E}[y_*|\mathbf{y}, \mathbf{X}, \mathbf{x}_*] &= \mathbf{K}_*^T \beta \\ Var[y_*|\mathbf{y}, \mathbf{X}, \mathbf{x}_*] &= k_{**} - \mathbf{k}_*^T (\mathbf{K} + \sigma_\omega^2 \boldsymbol{I})^{-1} \mathbf{k}_* \end{aligned} \tag{V.1}$$

where $\mathbf{k}_* := k(\mathbf{X}, \mathbf{x}_*)$, $k_{**} := k(\mathbf{x}_*, \mathbf{x}_*)$, $\mathbf{K} := k(\mathbf{X}, \mathbf{X})$ and $\beta := (\mathbf{K} + \sigma_\omega^2 \boldsymbol{I})^{-1} \mathbf{y}$.

In this work, we initialize the model with previously benchmarked metrics and then re-learn the model in an online fashion based on a moving window technique whenever new measurements are received. Since we emphasize online learning, we reduce the input features to our model to make the learning faster. First, we reduce the application level features using Pearson correlation [27] analysis, and filtering out features with low correlation. Second, we cluster the system level metrics using K-Means, so that each cluster reflects a performance interference level caused by the co-located workloads. For each cluster, we segment its corresponding workload measurements and the container-level metrics to learn a distinct performance model of the latency-sensitive application. The cluster-based learning is very beneficial because it allows us to estimate the performance interference level caused by the co-located workloads. Moreover, it allows us to reduce the features dimension of the performance model (i.e., model input size) for fast online learning, since we learn independent models for each cluster using their corresponding workload measurements and container-level metrics as opposite to learning one model with all measurements including host-level measurements as inputs.

Figure 28 depicts the online performance model learning steps from our framework. The dashed lines indicate the learning steps and the solid lines map to prediction steps. In the first phase, we start by clustering the system-level metrics to estimate performance interference levels and use the associated workload measurements and the container-level metrics to learn a performance model, i.e., latency model using a distinct GP model for each estimated performance interference level. Furthermore, we learn a time-series GP model of the application workload, i.e., online users, so we can forecast the workload for the next time-step.

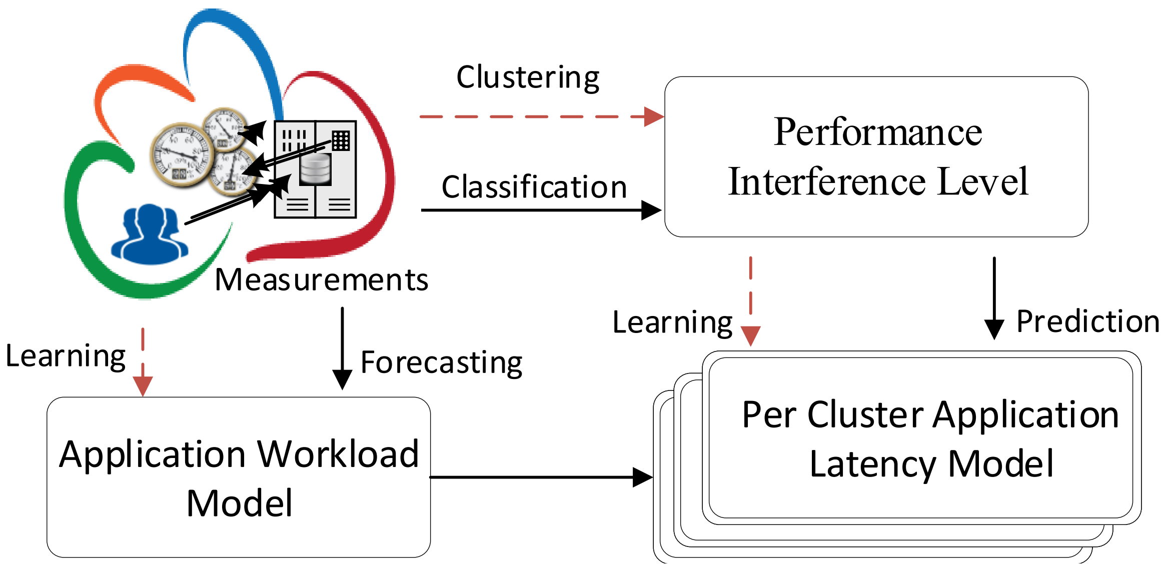

Clustering
Classification
Performance Interference Level
Measurements
Learning
Forecasting
Learning
Prediction
Application Workload Model
Per Cluster Application Latency Model

Figure 28: Performance Model Learning and Prediction Steps

After learning the model, in the prediction phase, we use the current measurement as the model input to predict the application performance in terms of latency. Note that once we learn the initial model, the performance prediction happens first and the learned model is updated next. This ensures that performance prediction does not get delayed due to model update. We start with estimating the current performance interference level by classifying the current host-level metrics. In this step, the clusters' centroid that we obtain in the model learning step are used as the classifier centroid. Then, we use the corresponding GP model to predict the latency of the system. In addition, we forecast the application workload using the workload time-series model and pass it as one of the inputs to predict the latency using the GP model associated with the estimated performance interference level as described above.

V.3.4 System Architecture and Implementation

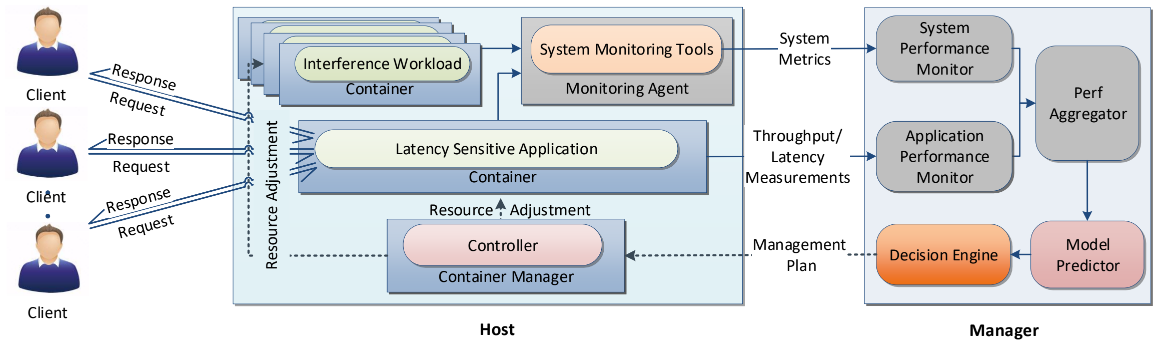


Figure 29: System Architecture

Figure 29 shows the system architecture. Our implementation comprises compute servers which host multiple containers or virtual machines. For our experiments, we used Docker as the virtualization mechanism, however, our architecture is generic enough to also include KVM or Xen-based hypervisors. The latency-sensitive application containers have dedicated cores assigned using CPU core pinning. The batch applications share the cores according to a defined overbooking ratio [33, 162]. The CPU and memory allocation are controlled using the Cgroups features supported by Docker.

The performance of the entire system is monitored using a resource usage and performance interference statistics collection framework that we have developed called FECBench [149]. The measurements include both macro and micro architectural metrics, such as CPU utilization, memory utilization, network I/O, disk I/O, context switches, page faults, cache utilization, retired instructions per second (IPS), memory bandwidth, scheduler wait time and scheduler I/O wait time. Additionally, each latency-sensitive application reports its observed workload and response time to the Application Performance Monitor residing on the framework's Manager that is deployed on a separate virtual machine. Figure 29 illustrates the monitoring agent from FECBench residing on each of the host which periodically reports metrics to the Manager.

The system performance metrics are aggregated with the latency-sensitive application workload and latency data, and passed on to the model predictor (described in Section V.3.3). The model predictor predicts the performance of the latency-sensitive application and forwards the information to a decision engine. The decision engine then decides the action which can be *add/remove cores* to the latency-sensitive application and *remove/add cores* or *checkpoint/restore* for batch applications. Our control action is based on the fact that adding more cores not only provides more resources to process additional workload, but also alleviates performance interference due to larger share of resources such as LLC and memory bandwidth [109]. The batch applications are checkpointed once they reach the overbooking ratio and the latency-sensitive applications need more resources. On the other hand, they are restored when it is found that restoring will still ensure that the latency-sensitive applications get their required resources and the overbooking ratio will not be exceeded. We leveraged Checkpoint/Restore In Userspace (CRIU) feature for Docker to achieve checkpoint and restore of the containers.

V.4 Experimental Validation

We present an experimental validation of our framework.

V.4.1 Evaluation Use Case

We consider a deployment scenario where latency-sensitive applications are co-located with batch applications whose workloads arrive in accordance to a distribution, emulated using real-world traces. Some of the popular cloud hosted latency-sensitive applications include web servers, search engines, media streaming among others.[1] For our system under test (SUT), i.e., the latency-sensitive application, we chose the CloudSuite Web-Search [131] benchmark since it fits our use case of varying workloads with low response

[1]Latency-sensitive does not imply hard real-time applications but rather applications that have soft bounds on response times beyond which users will find the application behavior unacceptable. For instance, users expect a web search to complete within a specific amount of time.

time needs. The default version of this benchmark, however, can log only the runtime statistics to a file. Since we needed the runtime statistics to be published to a remote location to make adaptive resource allocation decisions, we modified the benchmark to publish the results using RESTful APIs for our data collection and decision making.

Since the CloudSuite WebSearch benchmark also does not provide workloads for experimentation, to emulate a real web search engine workload, we used the workload pattern for Wikimedia projects from the Wikipedia traces [173]. Specifically, we collected the page view statistics for the main page in English language for the month of September 2017 and scaled the first two weeks of data to our experimental duration. We used the scaled first week data for model training and the next week data for testing.

We used two batch applications to co-locate with the SUT: the first one is the Stream benchmark from the Phoronix test suite (`http://www.phoronix-test-suite.com/`), which is a cache and memory-intensive application, and the second one is a memory-intensive custom Java application.

V.4.2 Experimental Setup

Our experimental setup consists of a compute server with the configuration as defined in Table 12. The server has Linux kernel 4.4.0-98, Docker version 17.05.0-CE and CRIU version 2.6 for checkpointing and restoring Linux containers.

Table 12: Hardware & Software Specification of Compute Server

Processor	2.1 GHz Opteron
Number of CPU cores	12
Memory	32 GB
Disk Space	500 GB
Operating System	Ubuntu 16.04.3 64-bit

We used the containerized version v3.0 of the Cloudsuite Web Search Benchmark as our SUT. The SUT is deployed on a server where it receives varying workloads over a period of time. The SUT is initially assigned 2 cores and 12 GB of memory. We vertically scale the number of assigned cores based on the output from the decision engine. The number of cores can vary from 2 to 10. One container is used to emulate the clients by varying their count as per the defined workload. This container also collects response times and throughput metrics. We deployed the client container on a separate host such that it does not have any effect on the experimental results similar to production deployment where the clients are located outside of the system. We aggregated and scaled the traces so that the number of users to the SUT change every 40 seconds.

The CloudSuite Web Search Benchmark relies on Faban [2] for time varying workload generation and statistics collection. We modified the Faban core used by CloudSuite benchmark so that it reports runtime metrics to the manager VM for model prediction and decision making. The metrics provided by CloudSuite include the throughput of the application, average latency which we use for the decision making and the 90th percentile latency which we consider as the tail latency used for measuring the efficacy of our system. Another key component of our experimentation is the FECBench framework that collects the application performance and system utilization metrics at an interval of 5 seconds and reports them to the Manager.

The manager VM is located on a separate machine which is responsible for each host's resource allocation decision making. The decision making and model update occurs every 15 sec which was chosen in order to avoid too frequent resource allocation modifications. Before deploying the system for online model prediction, we first perform offline analysis of the measured data set for the first week of the scaled Wikipedia traces. We applied the Silhouette [140] technique that determined two cluster centroids for performance interference level, which we used as the number of clusters in our online K-Means learning. For the online learning, we used 350 points as the K-Means window size and 200 points as the GP window size. We used the second week scaled data set for the experimental validation.

For the batch applications, the custom Java application is initially assigned 2 shared cores and 4GB of memory while the Stream test application is assigned 2 shared cores and 2.5 GB memory. We used Grid computing workload traces [80] to vary the batch application workload, which arrives according to a distribution to the same server as the SUT.

Our objective is to ensure that the latency-sensitive application adheres to the defined SLO guarantees while also allowing the batch applications to utilize the remaining resource slack. In order to achieve this, we need to appropriately assign resources to the latency-sensitive application and allocate the remaining resources to batch applications. While doing this, as the workload on the SUT increases, the resources allocated to batch application need to be reduced. However, this reduction in resources for batch applications will increase the overbooking ratio (i.e., degree of contention for a specified number of resources). In our experiments, when the overbooking ratio reaches 2, the batch applications must be checkpointed in a way that does not incur the limitations of prior work where memory continues to be held by these batch applications. Later, when the workload on the latency-sensitive application reduces, the checkpointed applications are restored.

V.4.3 Experimental Results

The standard practice for cloud data center resource management involves threshold-based resource allocation. Approaches such as the ones defined in [9, 75] are reactive in nature and usually have thresholds based on request rate, response time or resource utilization. Thus, we compare our model predictive framework against two threshold-based reactive approaches. In the first approach, we set the threshold based on CPU utilization of the SUT container. The objective of the approach is to keep the CPU utilization within a target range.

We chose CPU utilization range of 50-70%, named as $Reactive_{Utililization}$. We make this choice as we do not want the server to be either under-utilized or become saturated. Whenever the utilization grows/reduces from the target range, we add or remove a core,

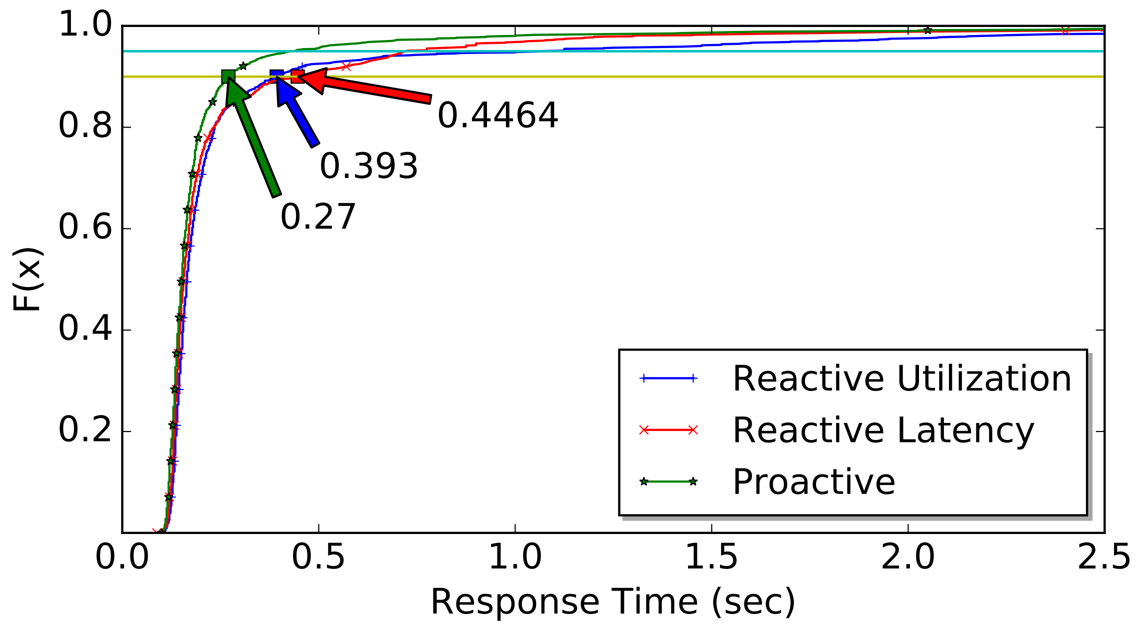


Figure 30: CDF of Response Times for 3 different scenarios

respectively. In addition, if there is a sudden gain or drop of more than 20% utilization, we add or remove two cores. In the second approach, we put the threshold on average latency, which was chosen over tail latency because the latter characterizes transient and higher fluctuations, which is not needed for control actions. The target range was set to 70-100 ms. We also had higher bounds for adding/removing two cores of 50/150 ms. The configuration is called $Reactive_{Latency}$. We used the same bound as $Reactive_{Latency}$ for our proactive approach experiments, which is called $Proactive$.

Figure 30 compares the response time of the three scenarios listed above. We observe that the tail latency (90 percentile) of our proactive approach is lowest at 270 ms. We also found the average latency to be 118, 102, 88 ms for $Reactive_{Utililization}$, $Reactive_{Latency}$ and $Proactive$, respectively. Figure 31 compares the average resource utilization for the duration of the experiment. Since our approach is a trade-off between resource utilization and the obtained latency, we observe that our proactive approach has higher resource utilization of 4.96 cores compared to 4.01 and 4.28 for the $Reactive_{Utililization}$ and $Reactive_{Latency}$

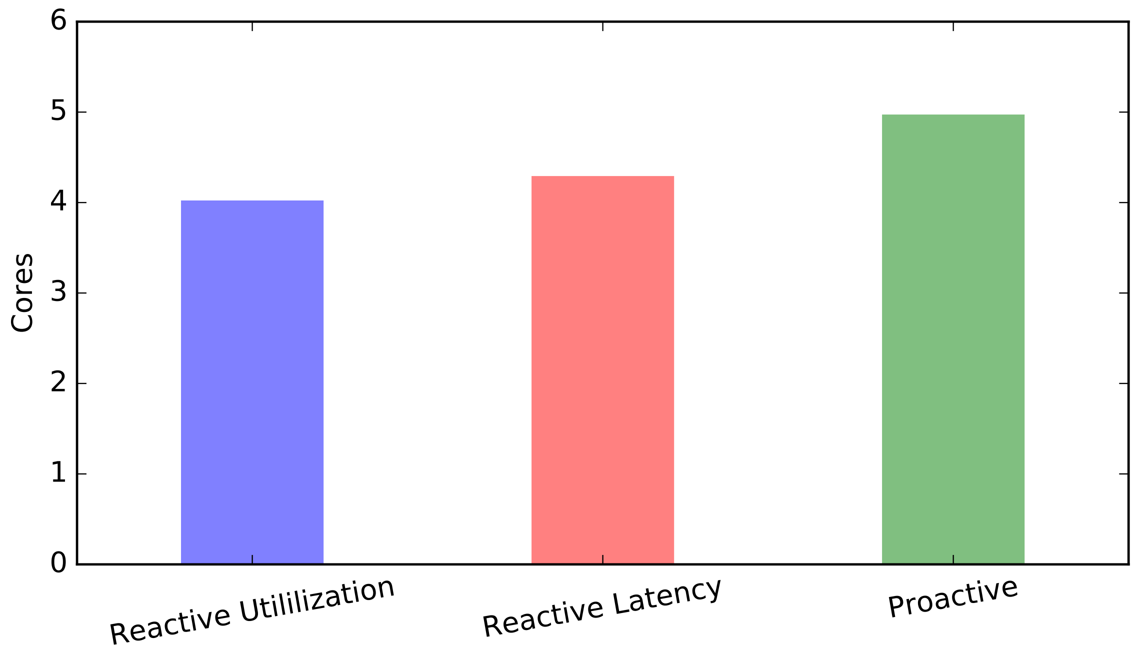


Figure 31: Resource Utilization for 3 different scenarios

approaches, respectively. Thus, compared to the two approaches, at the cost of 19.15% and 13.7% extra resources, we achieve 39.46% and 31.29% better tail latency, respectively.

We also measured the efficacy of our model prediction. Figure 32 shows the model prediction results. We achieved a mean absolute percent error of 7.56%. For interference level, we found 2 clusters for our workload. The *Co-located Workload Clusters* part of Figure 32 displays different regions of co-located workloads. The other two subfigures compare our prediction against observed latency and request rate.

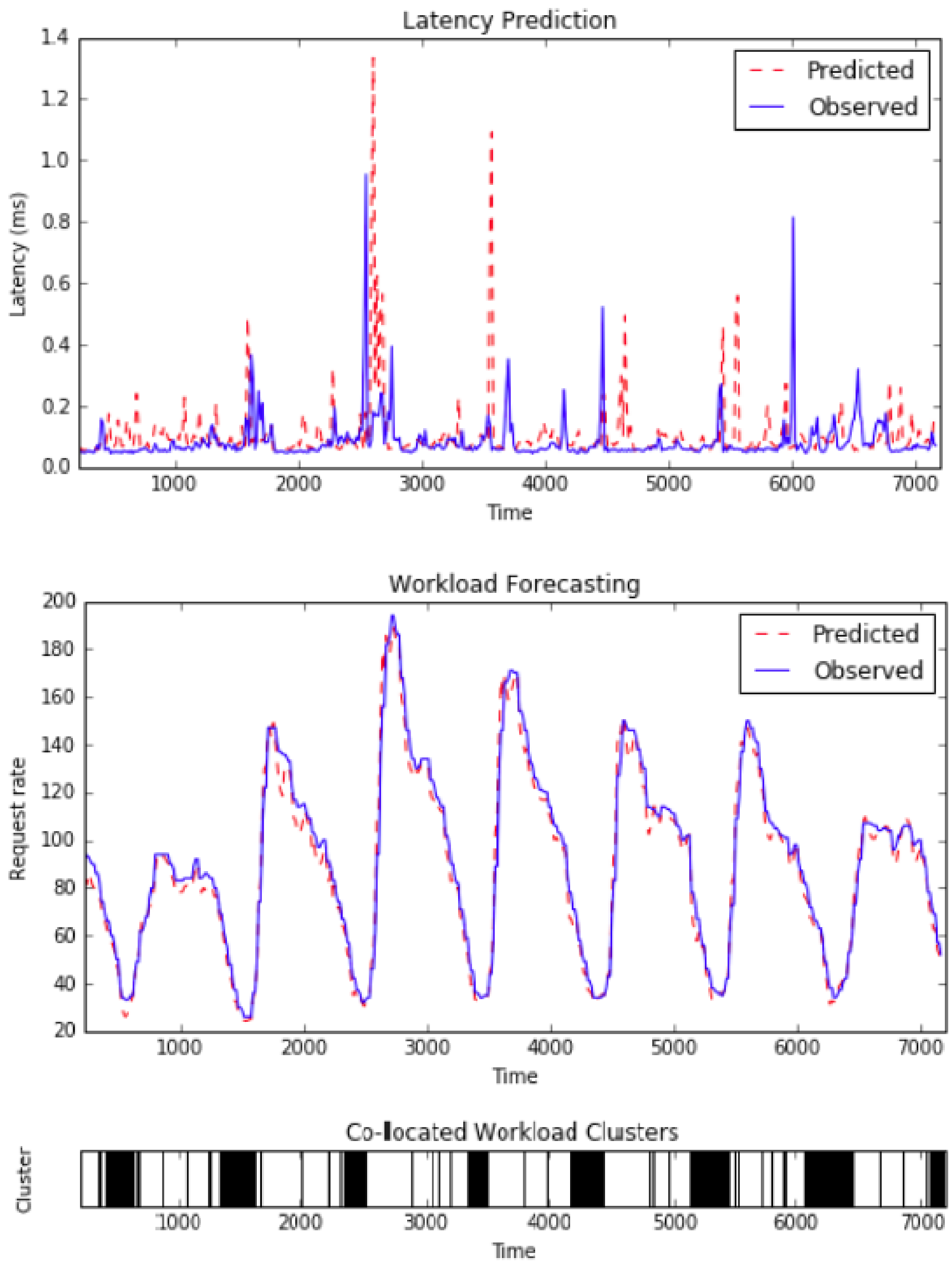


Figure 32: Model Prediction

V.5 Concluding Remarks

Dynamic vertical elasticity solutions are increasingly becoming attractive in cloud platforms as a first choice before employing horizontal elasticity strategies. To that end, this chapter presented a data-driven, machine learning techniques based on Gaussian Processes to build a runtime predictive model of the performance of the system, which can adapt itself to the variability in workload changes. This model is then used to make runtime decisions in terms of vertically scaling resources such that performance interference is minimized and QoS properties of latency-sensitive applications are met. Empirical validation of our technology on a representative latency-sensitive application reveals up to 39.46% lower tail latency than reactive approaches.

Our work provided us with the following insights and unresolved problems, which form the dimensions for our future work.

- Our work lacks finer-grained resource control such as managing CPU shares and memory, last-level cache and network allocation. To that end we are exploring the use of modern hardware advances, such as Intel's cache allocation technology and software-defined networking approaches to control network resource allocations.

- In this chapter, we viewed latency-sensitive applications to be designed as a monolithic application that can be containerized. However, with applications increasingly being designed as distributed interacting microservices, it becomes necessary to provide a distributed and coordinated vertical elasticity solutions.

- The thresholds for reactive approaches were chosen based on available literature. More experimentation is needed to compare against different thresholds. Likewise, additional experimentation is needed for different kinds of latency-sensitive applications.

- Presently, each of the clustered GP model executes inside the same VM. For future, we will perform distributed machine learning to reduce our online learning duration.

- Our future work will consider combining vertical scaling with horizontal scaling trading off along the different dimensions based on application needs and incoming workloads. Our future work will also include different categories of workloads, which can be both predictable and unpredictable.

The source code and experimental apparatus is available in open source at `https://github.com/doc-vu/verticalelasticity`.

CHAPTER VI

CONCLUDING REMARKS

The research conducted for this doctoral dissertation stemmed from having to satisfy the QoS needs of cloud data center-hosted applications. Specifically, we targeted latency-sensitive applications that reside on virtualized and multi-tenant servers but face performance interference issues due to resource contention from co-located applications and higher network latencies experienced by users due to the multiple network hops needed to reach cloud-based services. To that end, fog/edge resources offer a promising alternative to address the longer network latency issues. Specifically, we targeted latency-sensitive applications such as cognitive assistance, patient monitoring systems and online collaborations which are suitable for fog deployment because they incur strain on resources such as battery if deployed solely on the mobile edge device, or incur unacceptable network latency-imposed delays when deployed solely on the centralized cloud.

However, we found that the same cloud data center issues of performance interference extend themselves to fog and edge resources. In this context, we identified several resource management challenges across the edge-cloud spectrum and classified them under three broad categories, namely, application imposed, cloud provider related and measurement related challenges. To address these challenges, this dissertation developed several algorithms for dynamic resource management across the cloud-edge spectrum, which are codified within a common framework called *Dynamic Data Driven Cloud and Edge Systems (D^{3}CES).* The framework is composed of several components each of which provides a systematic and scientific approach to address different dimensions of challenges prevalent in the cloud-edge spectrum.

VI.1 Summary of Research Contributions

This doctoral research provided four major contributions while addressing latency related challenges in the realm of cloud-edge computing. We summarize them as follows:

- **SIMaaS:** Many seemingly simple questions that individual users face in their daily lives may actually require substantial number of computing resources to identify the right answers. For example, a user may want to determine the right thermostat settings for different rooms of a house based on a tolerance range such that the energy consumption and costs can be maximally reduced while still offering comfortable temperatures in the house. Such answers can be determined through simulations. However, some simulation models as in this example are stochastic, which require the execution of a large number of simulation tasks and aggregation of results to ascertain if the outcomes lie within specified confidence intervals. Some other simulation models, such as the study of traffic conditions using simulations may need multiple instances to be executed for a number of different parameters. Cloud computing has opened up new avenues for individuals and organizations with limited resources to obtain answers to problems that hitherto required expensive and computationally-intensive resources. We presented SIMaaS, which is a cloud-based Simulation-as-a-Service to address these challenges. We demonstrate how lightweight solutions using Linux containers (e.g., Docker) are better suited to support such services instead of heavyweight hypervisor-based solutions, which are shown to incur substantial overhead in provisioning virtual machines on-demand. Empirical results validating our claims are presented in the context of two case studies.

- **INDICES:** An increasing number of interactive applications and services, such as online gaming and cognitive assistance, are being hosted in the cloud because of its elastic properties and cost benefits. Despite these benefits, the longer and often unpredictable end-to-end network latencies between the end user and the cloud can be detrimental to the response time requirements of the applications. Although technology enablers, such as Cloudlets or Micro Data Centers (MDCs), are increasingly being leveraged by cloud infrastructure providers to address the network latency concerns, existing efforts in re-provisioning services from the cloud to the MDCs seldom focus on ensuring that the performance properties of the migrated services are met. We presented INDICES that makes three contributions to address these limitations: (a) determining when to reprovision, (b) identifying the appropriate MDC and its host from among multiple choices such that the performance considerations of the applications are met, and (c) ensuring that the cloud service provider continues to meet customer service level objectives while keeping its operational and energy costs low. Empirical results validating the claims are presented using a setup comprising a cloud data center and multiple MDCs composed of heterogeneous hardware.

- **URMILA:** The fog/edge computing paradigm is increasingly being adopted for cloud-hosted latency-sensitive services in order to reduce the uncertainties and network delays incurred while transferring messages between client device and cloud over wide area networks. Emerging research in this realm has focused mostly on offloading computation from the edge devices to the fog and cloud, but there is a dearth of solutions that take into account performance interference due to the co-location of multiple applications on the same fog server. The problem becomes even more challenging when the user's mobility is considered. We presented, URMILA, a middleware solution that takes a holistic view of the edge, fog and cloud resources with an aim of reducing the Service Level Objective (SLO) violations, while minimizing the cost to the cloud provider as well as energy consumption of the client's mobile device. The solution not only performs initial resource selection, but also assures SLOs at run-time for the mobile users. We evaluate URMILA's capabilities in the context of a real-world use case.

- **Vertical Elasticity:** Elastic auto-scaling in cloud platforms has primarily used horizontal scaling by assigning application instances to distributed resources. Owing to rapid advances in hardware, cloud providers are now seeking vertical elasticity before attempting horizontal scaling to provide elastic auto-scaling for applications. Vertical elasticity solutions must, however, be cognizant of performance interference that stems from multi-tenant collocated applications since interference significantly impacts application quality-of-service (QoS) properties, such as latency. The problem becomes more pronounced for latency-sensitive applications that demand strict QoS properties. Further exacerbating the problem are variations in workloads, which make it hard to determine the right kinds of timely resource adaptations for latency-sensitive applications. To address these challenges and overcome limitations in existing offline approaches, we present an online, data-driven approach which utilizes Gaussian Processes-based machine learning techniques to build runtime predictive models of the performance of the system under different levels of interference. The predictive online models are then used in dynamically adapting to the workload variability by vertically auto-scaling co-located applications such that performance interference is minimized and QoS properties of latency-sensitive applications are met.

VI.2 List of Publications

I have published a total of 22 peer-reviewed articles to date and two articles are in submission that can be classified as listed below:

Journal Publications

1. Shashank Shekhar, Michael Walker, Hamzah Abdelaziz, Faruk Caglar, Aniruddha Gokhale, and Xenofon Koutsoukos. A Simulation-as-a-Service Cloud Middleware. Journal of the Annals of Telecommunications, 74(3-4):93-108, 2016

2. Faruk Caglar, Shashank Shekhar, and Aniruddha Gokhale. iTune: Engineering the

Performance of Xen Hypervisor via Autonomous and Dynamic Scheduler Reconfiguration. IEEE Transactions on Services Computing (TSC), 11(1):103-116, 2018

3. Faruk Caglar, Shashank Shekhar, Aniruddha Gokhale, Satabdi Basu, Tazrian Rafi, John Kinnebrew, and Gautam Biswas. Cloud-hosted Simulation-as-a-Service for High School STEM Education. Elsevier Simulation Modelling Practice and Theory: Special Issue on Cloud Simulation, 58(2):255-273, November 2015

4. Kyoungho An, Shashank Shekhar, Faruk Caglar, Aniruddha Gokhale, and Shivakumar Sastry. A Cloud Middleware for Assuring Performance and High Availability of Soft Real-time Applications. Elsevier Journal of Systems Architecture (JSA), 60(9):757-769, December 2014

Book Chapters

1. Shashank Shekhar, Fangzhou Sun, Abhishek Dubey, Aniruddha Gokhale, Himanshu Neema, Martin Lehofer, and Dan Freudberg. Transit Hub: A Smart Decision Support System for Public Transit Operations, pages 597-612. John Wiley & Sons, Inc., 2017

2. Yi Li, Shashank Shekhar, Yevgeniy Vorobeychik, Xenofon Koutsoukos, and Aniruddha Gokhale. Simulation-based Optimization as a Service for Dynamic Data-driven Applications Systems. To Appear in Handbook of Dynamic Data Driven Applications Systems (*to appear*), 2018. Springer

Conference Publications

1. Shashank Shekhar, Hamzah Abdel-aziz, Anirban Bhattacharjee, Aniruddha Gokhale, and Xenofon Koutsoukos. "Performance Interference-Aware Vertical Elasticity for Cloud-hosted Latency-Sensitive Applications." In Cloud Computing (CLOUD), 2018 IEEE International Conference on, (*to appear*). IEEE, 2018.

2. Shashank Shekhar, Ajay Chhokra, Anirban Bhattacharjee, Guillaume Aupy, and Aniruddha Gokhale. "INDICES: exploiting edge resources for performance-aware cloud-hosted services." In Fog and Edge Computing (ICFEC), 2017 IEEE 1st International Conference on, pp. 75-80. IEEE, 2017.

3. Shashank Shekhar and Aniruddha Gokhale. "Dynamic resource management across cloud-edge resources for performance-sensitive applications." In Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, pp. 707-710. IEEE Press, 2017.

4. Prithviraj Patil, Akram Hakiri, Shashank Shekhar, and Aniruddha Gokhale. "Scalable and Adaptive Software Defined Network Management for Cloud-hosted Group Communication Applications." In 10th IEEE/ACM International Conference on Utility and Cloud Computing UCC 2017. 2017.

5. Faruk Caglar, Shashank Shekhar, Aniruddha Gokhale, and Xenofon Koutsoukos. An Intelligent, Performance Interference-aware Resource Management Scheme for IoT Cloud Backends. In 1st IEEE International Conference on Internet- of-Things: Design and Implementation, pages 95-105, Berlin, Germany, April 2016.

6. Faruk Caglar, Shashank Shekhar, and Aniruddha Gokhale. iPlace: An Intelligent and Tunable Power- and Performance-Aware Virtual Machine Placement Technique for Cloud-based Real-time Applications. In 17th IEEE Computer Society Symposium on Object/component/service-oriented real-time distributed Computing Technology (ISORC '14)., pages 48-55, Reno, NV, USA, June 2014.

7. Shashank Shekhar, Faruk Caglar, Anton Dukeman, Liyan Hou, and Aniruddha Gokhale. A collaborative k-12 stem education framework using traffic flow as a real-world challenge problem. In 2014 ASEE Annual Conference, pages 24-28. ASEE Conferences, 2014

8. Anton Dukeman, Shashank Shekhar, Faruk Caglar, and Aniruddha Gokhale. Analyzing students' computational models as they learn in stem disciplines. In 2014 ASEE Annual Conference. ASEE Conferences, 2014

9. Anton Dukeman, Faruk Caglar, Shashank Shekhar, John Kinnebrew, Gautam Biswas, Doug Fisher, and Aniruddha Gokhale. Teaching Computational Thinking Skills in C3STEM with Traffic Simulation. In Andreas Holzinger and Gabriella Pasi, editors, Human-Computer Interaction and Knowledge Discovery in Complex, Unstructured, Big Data, volume 7947 of Lecture Notes in Computer Science, pages 350-357. Springer Berlin Heidelberg, 2013

Workshop and Posters

1. Shashank Shekhar, Yogesh Barve, and Aniruddha Gokhale. "Understanding Performance Interference Benchmarking and Application Profiling Techniques for Cloud-hosted Latency-Sensitive Applications." In Proceedings of the 10th International Conference on Utility and Cloud Computing (UCC '17), pages 187-188. ACM 2017

2. Shashank Shekhar and Aniruddha Gokhale. Poster Abstract: Enabling IoT Applications via Dynamic Cloud-Edge Resource Management. In Proceedings of The 2nd ACM/IEEE International Conference on Internet-of-Things Design and Implementation (IoTDI 2017), pp. 331-332. IEEE, 2017.

3. Shashank Shekhar. Dynamic data driven cloud systems for cloud-hosted CPS. In Cloud Engineering Workshop (IC2EW), 2016 IEEE International Conference on, pages 195-197. IEEE, 2016

4. Shashank Shekhar, Faruk Caglar, Kyoungho An, Takayaki Kuroda, Aniruddha Gokhale, and Swapna Gokhale. A Model-driven Approach for Price/Performance Trade-offs

in Cloud-based MapReduce Application Deployment. In 2nd International Workshop on Model-Driven Engineering for High Performance and CLoud computing (MDHPCL) at MODELS 2013, pages 37-42, MiamiBeach, FL, September 2013

5. Faruk Caglar, Shashank Shekhar, and Aniruddha Gokhale. Performance Interference-aware Virtual Machine Placement Strategy for Supporting Soft Real-time Applications in the Cloud. In 3rd International Workshop on Real-time and Distributed Computing in Emerging Applications (REACTION), IEEE RTSS 2014, Rome, Italy, December 2014.

6. Faruk Caglar, Kyoungho An, Shashank Shekhar, and Aniruddha Gokhale. Model-driven Performance Estimation, Deployment, and Resource Management for Cloud-hosted Services. In 13th Workshop on Domain-specific Modeling (DSM '13), In conjunction with SPLASH '13, Indianapolis, IN, USA, October 2013.

7. Kyoungho An, Faruk Caglar, Shashank Shekhar, and Aniruddha S. Gokhale. A framework for effective placement of virtual machine replicas for highly available performance-sensitive cloud-based applications. In REACTION 2012, First International Workshop on Real-time and distributed computing in emerging applications, Proceedings, San Juan, Puerto Rico, December 4, 2012

REFERENCES

[1] Detecting memory bandwidth saturation in threaded applications. URL: https://software.intel.com/en-us/articles/detecting-memory-bandwidth-saturation-in-threaded-applications, Last accessed: 2018-05-21.

[2] Faban - helping measure performance. URL: http://faban.org/, Last accessed: 2018-05-21.

[3] Rack-scale computing. `https://www.microsoft.com/en-us/research/project/rack-scale-computing/`, 2018.

[4] Seeing AI Project. `http://www.pivothead.com/seeingai/`, 2018.

[5] Alessandro Abate, Joost-Pieter Katoen, John Lygeros, and Maria Prandini. Approximate Model Checking of Stochastic Hybrid Systems. *European Journal of Control*, 16(6):624–641, 2010.

[6] Alessandro Abate, Maria Prandini, John Lygeros, and Shankar Sastry. Probabilistic Reachability and Safety for Controlled Discrete Time Stochastic Hybrid Systems. *Automatica*, 44(11):2724–2734, 2008.

[7] Tim Abels, Puneet Dhawan, and Balasubramanian Chandrasekaran. An overview of xen virtualization. *Dell Power Solutions*, 8:109–111, 2005.

[8] Mumtaz Ahmad, Songyun Duan, Ashraf Aboulnaga, and Shivnath Babu. Predicting completion times of batch query workloads using interaction-aware models and simulation. In *Proceedings of the 14th International Conference on Extending Database Technology*, pages 449–460. ACM, 2011.

[9] Raja Wasim Ahmad, Abdullah Gani, Siti Hafizah Ab Hamid, Muhammad Shiraz,

Abdullah Yousafzai, and Feng Xia. A survey on virtual machine migration and server consolidation frameworks for cloud data centers. *Journal of Network and Computer Applications*, 52:11–25, 2015.

[10] Yahya Al-Dhuraibi, Fawaz Paraiso, Nabil Djarallah, and Philippe Merle. Autonomic vertical elasticity of docker containers with elasticdocker. In *10th IEEE International Conference on Cloud Computing, IEEE CLOUD 2017*, 2017.

[11] Khaldoon Al-Zoubi and Gabriel Wainer. Distributed Simulation using RESTful Interoperability Simulation Environment (RISE) Middleware. In *Intelligence-Based Systems Engineering*, pages 129–157. Springer, 2011.

[12] Atif Alamri, Wasai Shadab Ansari, Mohammad Mehedi Hassan, M Shamim Hossain, Abdulhameed Alelaiwi, and M Anwar Hossain. A Survey on Sensor-cloud: Architecture, Applications, and Approaches. *International Journal of Distributed Sensor Networks*, 2013, 2013.

[13] Rajeev Alur and George Pappas. *Hybrid Systems: Computation and Control: 7th International Workshop, HSCC 2004, Philadelphia, PA, USA, March 25-27, 2004, Proceedings*, volume 7. Springer, 2004.

[14] Kyoungho An, Shashank Shekhar, Faruk Caglar, Aniruddha Gokhale, and Shivakumar Sastry. A Cloud Middleware for Assuring Performance and High Availability of Soft Real-time Applications. *Elsevier Journal of Systems Architecture (JSA)*, 60(9):757–769, December 2014.

[15] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The Internet of Things: A Survey. *Computer networks*, 54(15):2787–2805, 2010.

[16] Youakim Badr, Salim Hariri, Youssif AL-Nashif, and Erik Blasch. Resilient and

trustworthy dynamic data-driven application systems (dddas) services for crisis management environments. *Procedia Computer Science*, 51:2623 – 2637, 2015.

[17] Victor Bahl. Cloud 2020: Emergence of micro data centers (cloudlets) for latency sensitive computing (keynote). *Middleware 2015*, 2015.

[18] Rajesh Balan, Jason Flinn, Mahadev Satyanarayanan, Shafeeq Sinnamohideen, and Hen-I Yang. The case for cyber foraging. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, pages 87–92. ACM, 2002.

[19] Mircea Bardac, Razvan Deaconescu, and Adina Magda Florea. Scaling Peer-to-Peer Testing using Linux Containers. In *Roedunet International Conference (RoEduNet), 2010 9th*, pages 287–292. IEEE, 2010.

[20] Sean Kenneth Barker and Prashant Shenoy. Empirical evaluation of latency-sensitive application performance in the cloud. In *Proceedings of the first annual ACM SIGMM conference on Multimedia systems*, pages 35–46. ACM, 2010.

[21] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture*, 8(3):1–154, 2013.

[22] Yogesh D. Barve, Himanshu Neema, Aniruddha Gokhale, and Sztipanovits Janos. Model-driven automated deployment of large-scale cps co-simulations in the cloud (poster). In *ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems*, Austin, TX, 09/2017 2017.

[23] Yogesh D Barve, Prithviraj Patil, Anirban Bhattacharjee, and Aniruddha Gokhale. Pads: Design and implementation of a cloud-based, immersive learning environment for distributed systems algorithms. *IEEE Transactions on Emerging Topics in Computing*, 6(1):20–31, 2018.

[24] Yogesh D Barve, Prithviraj Patil, and Aniruddha Gokhale. A cloud-based immersive learning environment for distributed systems algorithms. In *Computer Software and Applications Conference (COMPSAC), 2016 IEEE 40th Annual*, volume 1, pages 754–763. IEEE, 2016.

[25] Michael Till Beck, Sebastian Feld, Andreas Fichtner, Claudia Linnhoff-Popien, and Thomas Schimper. Me-volte: Network functions for energy-efficient video transcoding at the mobile edge. In *Intelligence in Next Generation Networks (ICIN), 2015 18th International Conference on*, pages 38–44. IEEE, 2015.

[26] Michael Behrisch, Laura Bieker, Jakob Erdmann, and Daniel Krajzewicz. SUMO-Simulation of Urban MObility-an Overview. In *SIMUL 2011, The Third International Conference on Advances in System Simulation*, pages 55–60, 2011.

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

[28] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: characterization and architectural implications. In *17th international conference on Parallel architectures and compilation techniques*, pages 72–81. ACM, 2008.

[29] Luiz F Bittencourt, Javier Diaz-Montes, Rajkumar Buyya, Omer F Rana, and Manish Parashar. Mobility-aware application scheduling in fog computing. *IEEE Cloud Computing*, 4(2):26–35, 2017.

[30] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pages 13–16. ACM, 2012.

[31] Robert Bradford, Evangelos Kotsovinos, Anja Feldmann, and Harald Schiöberg.

Live wide-area migration of virtual machines including local persistent state. In *3rd international conference on Virtual execution environments*, pages 169–179. ACM, 2007.

[32] Cache allocation technology improves real-time performance. `http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/cache-allocation-technology-white-paper.pdf`.

[33] Faruk Caglar, Shashank Shekhar, and Aniruddha Gokhale. iPlace: An Intelligent and Tunable Power- and Performance-Aware Virtual Machine Placement Technique for Cloud-based Real-time Applications. In *17th IEEE Computer Society Symposium on Object/component/service-oriented real-time distributed Computing Technology (ISORC '14).*, pages 48–55, Reno, NV, USA, June 2014. IEEE.

[34] Faruk Caglar, Shashank Shekhar, Aniruddha Gokhale, Satabdi Basu, Tazrian Rafi, John Kinnebrew, and Gautam Biswas. Cloud-hosted Simulation-as-a-Service for High School STEM Education. *Elsevier Simulation Modelling Practice and Theory: Special Issue on Cloud Simulation*, 58(2):255–273, November 2015.

[35] Faruk Caglar, Shashank Shekhar, Aniruddha Gokhale, and Xenofon Koutsoukos. An Intelligent, Performance Interference-aware Resource Management Scheme for IoT Cloud Backends. In *1st IEEE International Conference on Internet-of-Things: Design and Implementation*, pages 95–105, Berlin, Germany, April 2016. IEEE.

[36] Faruk Caglar, Shashank Shekhar, Aniruddha Gokhale, and Xenofon Koutsoukos. Intelligent, performance interference-aware resource management for iot cloud backends. In *2016 IEEE First International Conference on Internet-of-Things Design and Implementation (IoTDI)*, pages 95–105. IEEE, 2016.

[37] Rodrigo N Calheiros, Marco AS Netto, César AF De Rose, and Rajkumar Buyya. EMUSIM: An Integrated Emulation and Simulation Environment for Modeling, Evaluation, and Validation of Performance of Cloud Computing Applications. *Software: Practice and Experience*, 43(5):595–612, 2013.

[38] Rodrigo N Calheiros, Rajiv Ranjan, Anton Beloglazov, César AF De Rose, and Rajkumar Buyya. CloudSim: A Toolkit for Modeling and Simulation of Cloud Computing Environments and Evaluation of Resource Provisioning Algorithms. *Software: Practice and Experience*, 41(1):23–50, 2011.

[39] Rodrigo N Calheiros, Christian Vecchiola, Dileban Karunamoorthy, and Rajkumar Buyya. The Aneka Platform and QoS-driven Resource Provisioning for Elastic Applications on Hybrid Clouds. *Future Generation Computer Systems*, 28(6):861–870, 2012.

[40] Henri Casanova, Arnaud Giersch, Arnaud Legrand, Martin Quinson, and Frédéric Suter. SimGrid: A Sustained Effort for the Versatile Simulation of Large-scale Distributed Systems. *arXiv preprint arXiv:1309.1630*, 2013.

[41] Ranveer Chandra and Paramvir Bahl. Multinet: Connecting to multiple ieee 802.11 networks using a single wireless card. In *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, volume 2, pages 882–893. IEEE, 2004.

[42] Changbeom Choi, Kyung-Min Seo, and Tag Gon Kim. Dexsim: an experimental environment for distributed execution of replicated simulators using a concept of single simulation multiple scenarios. *Simulation*, 90(4):355–376, 2014.

[43] Sharon Choy, Bernard Wong, Gwendal Simon, and Catherine Rosenberg. The brewing storm in cloud gaming: A measurement study on cloud to end-user latency.

In *Proceedings of the 11th annual workshop on network and systems support for games*, page 2. IEEE Press, 2012.

[44] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. Clonecloud: elastic execution between mobile device and cloud. In *Proceedings of the sixth conference on Computer systems*, pages 301–314. ACM, 2011.

[45] Andrei Croitoru, Dragos Niculescu, and Costin Raiciu. Towards wifi mobility without fast handover. In *NSDI*, pages 219–234, 2015.

[46] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. Maui: making smartphones last longer with code offload. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, pages 49–62. ACM, 2010.

[47] Frederica Darema. Dynamic data driven applications systems (dddas)–a transformative paradigm. In *Computational Science–ICCS 2008*, pages 5–5. Springer, 2008.

[48] Christina Delimitrou and Christos Kozyrakis. ibench: Quantifying interference for datacenter applications. In *Workload Characterization (IISWC), 2013 IEEE International Symposium on*, pages 23–33. IEEE, 2013.

[49] Christina Delimitrou and Christos Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. In *ACM SIGPLAN Notices*, volume 48, pages 77–88. ACM, 2013.

[50] Christina Delimitrou and Christos Kozyrakis. Quasar: resource-efficient and qos-aware cluster management. In *ACM SIGPLAN Notices*, volume 49, pages 127–144. ACM, 2014.

[51] Shuo Deng, Ravi Netravali, Anirudh Sivaraman, and Hari Balakrishnan. Wifi, lte,

or both?: Measuring multi-homed wireless internet performance. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, pages 181–194. ACM, 2014.

[52] Tharam Dillon, Chen Wu, and Elizabeth Chang. Cloud computing: issues and challenges. In *Advanced Information Networking and Applications (AINA), 2010 24th IEEE International Conference on*, pages 27–33. Ieee, 2010.

[53] Cong Ding, Yang Chen, Tianyin Xu, and Xiaoming Fu. Cloudgps: a scalable and isp-friendly server selection scheme in cloud computing environments. In *IEEE 20th International Workshop on Quality of Service*, page 5. IEEE Press, 2012.

[54] Trinh Minh Tri Do and Daniel Gatica-Perez. Contextual conditional models for smartphone-based human mobility prediction. In *Proceedings of the 2012 ACM conference on ubiquitous computing*, pages 163–172. ACM, 2012.

[55] Paul J Drongowski and Boston Design Center. Basic performance measurements for amd athlonâĎć 64, amd opteronâĎć and amd phenomâĎć processors. *AMD whitepaper*, 25, 2008.

[56] Jane Elith, John R Leathwick, and Trevor Hastie. A working guide to boosted regression trees. *Journal of Animal Ecology*, 77(4):802–813, 2008.

[57] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An updated performance comparison of virtual machines and linux containers. In *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On*, pages 171–172. IEEE, 2015.

[58] Debessay Fesehaye, Yunlong Gao, Klara Nahrstedt, and Guijun Wang. Impact of cloudlets on interactive mobile cloud applications. In *Enterprise Distributed Object Computing Conference (EDOC), 2012 IEEE 16th International*, pages 123–132.

IEEE, 2012.

[59] Florian Forster. Collectd - The System Statistics Collection Daemon. `http://collectd.org`, 2017.

[60] David A Forsyth and Jean Ponce. *Computer vision: a modern approach*. Prentice Hall Professional Technical Reference, 2002.

[61] Richard M Fujimoto. Parallel Discrete Event Simulation. *Communications of the ACM*, 33(10):30–53, 1990.

[62] Richard M Fujimoto, Asad Waqar Malik, and A Park. Parallel and Distributed Simulation in the Cloud. *SCS M&S Magazine*, 3:1–10, 2010.

[63] Anshul Gandhi, Parijat Dube, Alexei Karve, Andrzej Kochut, and Li Zhang. Adaptive, model-driven autoscaling for cloud applications. In *11th International Conference on Autonomic Computing (ICAC 14)*, pages 57–64, 2014.

[64] Yue Gao, Yanzhi Wang, Sandeep K Gupta, and Massoud Pedram. An Energy and Deadline Aware Resource Provisioning, Scheduling and Optimization Framework for Cloud Systems. In *Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, page 31. IEEE Press, 2013.

[65] Marisol García-Valls, Tommaso Cucinotta, and Chenyang Lu. Challenges in Real-Time Virtualization and Predictable Cloud Computing. *Journal of Systems Architecture*, 2014.

[66] Sriram Govindan, Jie Liu, Aman Kansal, and Anand Sivasubramaniam. Cuanta: quantifying effects of shared on-chip resource interference for consolidated virtual machines. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*,

page 22. ACM, 2011.

[67] Chrispin Gray, Robert Ayre, Kerry Hinton, and Rodney S Tucker. Power consumption of iot access network technologies. In *Communication Workshop (ICCW), 2015 IEEE International Conference on*, pages 2818–2823. IEEE, 2015.

[68] Kiryong Ha, Yoshihisa Abe, Zhuo Chen, Wenlu Hu, Brandon Amos, Padmanabhan Pillai, and Mahadev Satyanarayanan. Adaptive vm handoff across cloudlets. Technical report, Technical Report CMU-CS-15-113, CMU School of Computer Science, 2015.

[69] Kiryong Ha, Padmanabhan Pillai, Grace Lewis, Soumya Simanta, Sarah Clinch, Nigel Davies, and Mahadev Satyanarayanan. The impact of mobile multimedia applications on data center consolidation. In *Cloud Engineering (IC2E), 2013 IEEE International Conference on*, pages 166–176. IEEE, 2013.

[70] Mohammad Hajjat, David Maltz, Sanjay Rao, Kunwadee Sripanidkulchai, et al. Dealer: application-aware request splitting for interactive cloud applications. In *8th international conference on Emerging networking experiments and technologies*, pages 157–168. ACM, 2012.

[71] Mordechai Haklay and Patrick Weber. Openstreetmap: User-generated Street Maps. *Pervasive Computing, IEEE*, 7(4):12–18, 2008.

[72] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, Bob Lantz, and Nick McKeown. Reproducible Network Experiments using Container-based Emulation. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, pages 253–264. ACM, 2012.

[73] John A Hartigan and Manchek A Wong. Algorithm as 136: A k-means clustering algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*,

28(1):100–108, 1979.

[74] Sylvain Hellegouarch. *CherryPy Essentials: Rapid Python Web Application Development*. Packt Publishing Ltd, 2007.

[75] Nikolas Roman Herbst, Nikolaus Huber, Samuel Kounev, and Erich Amrehn. Self-adaptive workload classification and forecasting for proactive resource provisioning. *Concurrency and computation: practice and experience*, 26(12):2053–2078, 2014.

[76] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[77] Yazhou Hu, Bo Deng, Fuyang Peng, and Dongxia Wang. Workload prediction for cloud computing elasticity mechanism. In *Cloud Computing and Big Data Analysis (ICCCBDA), 2016 IEEE International Conference on*, pages 244–249. IEEE, 2016.

[78] Yun Chao Hu, Milan Patel, Dario Sabella, Nurit Sprecher, and Valerie Young. Mobile edge computingâĂŤa key technology towards 5g. *ETSI white paper*, 11(11):1–16, 2015.

[79] Junxian Huang, Feng Qian, Alexandre Gerber, Z Morley Mao, Subhabrata Sen, and Oliver Spatscheck. A close examination of performance and power characteristics of 4g lte networks. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pages 225–238. ACM, 2012.

[80] Alexandru Iosup and Dick Epema. Grid computing workloads. *IEEE Internet Computing*, 15(2):19–26, 2011.

[81] Mohammad Shahedul Islam, Matt Gibson, and Abdullah Muzahid. Fast and qos-aware heterogeneous data center scheduling using locality sensitive hashing. In *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 74–81. IEEE, 2015.

[82] Michael Jarschel, Daniel Schlosser, Sven Scheuring, and Tobias Hoßfeld. Gaming in the clouds: Qoe and the usersâĂŹ perspective. *Mathematical and Computer Modelling*, 57(11):2883–2894, 2013.

[83] Evangelia Kalyvianaki, Themistoklis Charalambous, and Steven Hand. Adaptive resource provisioning for virtualized servers using kalman filters. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 9(2):10, 2014.

[84] Melanie Kambadur, Tipp Moseley, Rick Hank, and Martha A Kim. Measuring interference between live datacenter applications. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 51. IEEE Computer Society Press, 2012.

[85] Alexander Kandalintsev, Dzmitry Kliazovich, and Renato Lo Cigno. Freeze'nsense: estimation of performance isolation in cloud environments. *Software: Practice and Experience*, 2016.

[86] Morteza Karimzadeh, Zhongliang Zhao, Luuk Hendriks, Ricardo de O Schmidt, Sebastiaan la Fleur, Hans van den Berg, Aiko Pras, Torsten Braun, and Marius Julian Corici. Mobility and bandwidth prediction as a service in virtualized lte systems. In *Cloud Networking (CloudNet), 2015 IEEE 4th International Conference on*, pages 132–138. IEEE, 2015.

[87] Rajat Kateja, Nimantha Baranasuriya, Vishnu Navda, and Venkata N Padmanabhan. Diversifi: Robust multi-link interactive streaming. In *Proceedings of the 11th ACM*

Conference on Emerging Networking Experiments and Technologies, page 35. ACM, 2015.

[88] Claire Kenyon et al. Best-Fit Bin-Packing with Random Order. In *SODA*, volume 96, pages 359–364, 1996.

[89] Hyunjoo Kim, Yaakoub El-Khamra, Ivan Rodero, Shantenu Jha, and Manish Parashar. Autonomic Management of Application Workflows on Hybrid computing Infrastructure. *Scientific Programming*, 19(2):75–89, 2011.

[90] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the Linux virtual machine monitor. In *Proceedings of the Linux Symposium*, volume 1, pages 225–230, 2007.

[91] Donald E Knuth. The Art of Computer Programming, Vol. 2: Seminumerical Algorithms, Revised Edition, 1969.

[92] Younggyun Koh, Rob Knauerhase, Paul Brett, Mic Bowman, Zhihua Wen, and Calton Pu. An analysis of performance interference effects in virtual environments. In *Performance Analysis of Systems & Software, 2007. ISPASS 2007. IEEE International Symposium on*, pages 200–209. IEEE, 2007.

[93] Ron Kohavi, Randal M Henne, and Dan Sommerfield. Practical guide to controlled experiments on the web: listen to your customers not to the hippo. In *13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 959–967. ACM, 2007.

[94] S. Kosta, A. Aucinas, Pan Hui, R. Mortier, and Xinwen Zhang. ThinkAir: Dynamic Resource Allocation and Parallel Execution in the Cloud for Mobile Code Offloading. In *INFOCOM, 2012 Proceedings IEEE*, pages 945–953, March 2012.

[95] Wei Kuang, Laura E Brown, and Zhenlin Wang. Modeling cross-architecture co-tenancy performance interference. In *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on*, pages 231–240. IEEE, 2015.

[96] Minseok Kwon, Zuochao Dou, Wendi Heinzelman, Tolga Soyata, He Ba, and Jiye Shi. Use of network latency profiling and redundancy for cloud server selection. In *2014 IEEE 7th International Conference on Cloud Computing*, pages 826–832. IEEE, 2014.

[97] Ewnetu Bayuh Lakew, Cristian Klein, Francisco Hernandez-Rodriguez, and Erik Elmroth. Towards faster response time models for vertical elasticity. In *Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, pages 560–565. IEEE Computer Society, 2014.

[98] Ewnetu Bayuh Lakew, Alessandro Vittorio Papadopoulos, Martina Maggio, Cristian Klein, and Erik Elmroth. Kpi-agnostic control for fine-grained vertical elasticity. In *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 589–598. IEEE Press, 2017.

[99] Patrick Lardieri, Jaiganesh Balasubramanian, Douglas C. Schmidt, Gautam Thaker, Aniruddha Gokhale, and Tom Damiano. A Multi-layered Resource Management Framework for Dynamic Resource Management in Enterprise DRE Systems. *Journal of Systems and Software: Special Issue on Dynamic Resource Management in Distributed Real-time Systems*, 80(7):984–996, July 2007.

[100] Roman Ledyayev and Harald Richter. High Performance Computing in a Cloud Using OpenStack. In *CLOUD COMPUTING 2014, The Fifth International Conference on Cloud Computing, GRIDs, and Virtualization*, pages 108–113, 2014.

[101] Jialin Li, Naveen Kr Sharma, Dan RK Ports, and Steven D Gribble. Tales of the tail:

Hardware, os, and application-level sources of tail latency. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–14. ACM, 2014.

[102] Yongbo Li, Yurong Chen, Tian Lan, and Guru Venkataramani. Mobiqor: Pushing the envelope of mobile edge computing via quality-of-result optimization. In *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on*, pages 1261–1270. IEEE, 2017.

[103] Yongbo Li, Fan Yao, Tian Lan, and Guru Venkataramani. Poster: Semantics-aware rule recommendation and enforcement for event paths. In *International Conference on Security and Privacy in Communication Systems*, pages 572–576. Springer, 2015.

[104] Yongbo Li, Fan Yao, Tian Lan, and Guru Venkataramani. Sarre: semantics-aware rule recommendation and enforcement for event paths on android. *IEEE Transactions on Information Forensics and Security*, 11(12):2748–2762, 2016.

[105] Zengxiang Li, Xiaorong Li, TA Duong, Wentong Cai, and Stephen John Turner. Accelerating Optimistic HLA-based Simulations in Virtual Execution Environments. In *Proceedings of the 2013 ACM SIGSIM conference on Principles of advanced discrete simulation*, pages 211–220. ACM, 2013.

[106] Ming Liu and Tao Li. Optimizing virtual machine consolidation performance on numa server architecture for cloud workloads. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pages 325–336. IEEE, 2014.

[107] Xiaocheng Liu, Xiaogang Qiu, Bin Chen, and Kedi Huang. Cloud-based simulation: the state-of-the-art computer simulation paradigm. In *Proceedings of the 2012 ACM/IEEE/SCS 26th Workshop on Principles of Advanced and Distributed Simulation*, pages 71–74. IEEE Computer Society, 2012.

[108] David Lo, Liqun Cheng, Rama Govindaraju, Luiz André Barroso, and Christos

Kozyrakis. Towards energy proportionality for large-scale latency-critical workloads. In *ACM SIGARCH Computer Architecture News*, volume 42, pages 301–312. IEEE Press, 2014.

[109] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving resource efficiency at scale. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 450–462. ACM, 2015.

[110] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Improving resource efficiency at scale with heracles. *ACM Transactions on Computer Systems (TOCS)*, 34(2):6, 2016.

[111] David G Lowe. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2):91–110, 2004.

[112] LXC. Linux Container, 2014. Last accessed: 10/11/2014.

[113] David JC MacKay. *Information theory, inference and learning algorithms*. Cambridge university press, 2003.

[114] Guoqiang Mao, Brian DO Anderson, and Barış Fidan. Path loss exponent estimation for wireless sensor network localization. *Computer Networks*, 51(10):2467–2483, 2007.

[115] Ming Mao and Marty Humphrey. A performance study on the vm startup time in the cloud. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 423–430. IEEE, 2012.

[116] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible

co-locations. In *44th annual IEEE/ACM International Symposium on Microarchitecture*, pages 248–259. ACM, 2011.

[117] Viktor Mauch, Marcel Kunze, and Marius Hillenbrand. High Performance Cloud Computing. *Future Generation Computer Systems*, 29(6):1408–1416, 2013.

[118] Peter Mell and Tim Grance. The nist definition of cloud computing. *National Institute of Standards and Technology*, 53(6):50, 2009.

[119] Paul B Menage. Adding generic process containers to the linux kernel. In *Proceedings of the Linux Symposium*, volume 2, pages 45–57. Citeseer, 2007.

[120] Dirk Merkel. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux J.*, 2014(239), March 2014.

[121] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, 2014.

[122] Jose Monsalve, Aaron Landwehr, and Michela Taufer. Dynamic cpu resource allocation in containerized cloud environments. In *Cluster Computing (CLUSTER), 2015 IEEE International Conference on*, pages 535–536. IEEE, 2015.

[123] Laura R Moore, Kathryn Bean, and Tariq Ellahi. Transforming reactive auto-scaling into proactive auto-scaling. In *Proceedings of the 3rd International Workshop on Cloud Data and Platforms*, pages 7–12. ACM, 2013.

[124] 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*, pages 237–250. ACM, 2010.

[125] Khang T Ngyuen. Intel's cache monitoring technology software-visible interfaces. `https://software.intel.com/en-us/blogs/2014/12/11/`

`intel-s-cache-monitoring-technology-software-visible-interfaces.`

[126] Anthony J Nicholson, Yatin Chawathe, Mike Y Chen, Brian D Noble, and David Wetherall. Improved access point selection. In *Proceedings of the 4th international conference on Mobile systems, applications and services*, pages 233–245. ACM, 2006.

[127] Ali Yadavar Nikravesh, Samuel A Ajila, and Chung-Horng Lung. An autonomic prediction suite for cloud resource provisioning. *Journal of Cloud Computing*, 6(1):3, 2017.

[128] Anastasios Noulas, Salvatore Scellato, Neal Lathia, and Cecilia Mascolo. Mining user mobility features for next place prediction in location-based services. In *Data mining (ICDM), 2012 IEEE 12th international conference on*, pages 1038–1043. IEEE, 2012.

[129] 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.

[130] Dejan Novakovic, Nedeljko Vasic, Stanko Novakovic, Dejan Kostic, and Ricardo Bianchini. Deepdive: Transparently identifying and managing performance interference in virtualized environments. In *Proceedings of the 2013 USENIX Annual Technical Conference*, number EPFL-CONF-185984, 2013.

[131] Tapti Palit, Yongming Shen, and Michael Ferdman. Demystifying cloud benchmarking. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 122–132, April 2016.

[132] Andrej Podzimek, Lubomír Bulej, Lydia Y Chen, Walter Binder, and Petr Tuma. Analyzing the impact of cpu pinning and partial cpu loads on performance and energy efficiency. In *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on*, pages 1–10. IEEE, 2015.

[133] Xing Pu, Ling Liu, Yiduo Mei, Sankaran Sivathanu, Younggyun Koh, Calton Pu, and Yuanda Cao. Who is your neighbor: Net i/o performance interference in virtualized clouds. *IEEE Transactions on Services Computing*, 6(3):314–329, 2013.

[134] Massimiliano Rak, Antonio Cuomo, and Umberto Villano. Mjades: Concurrent simulation in the cloud. In *Complex, Intelligent and Software Intensive Systems (CISIS), 2012 Sixth International Conference on*, pages 853–860. IEEE, 2012.

[135] Navaneeth Rameshan, Leandro Navarro, Enric Monte, and Vladimir Vlassov. Stay-away, protecting sensitive applications from performance interference. In *15th International Middleware Conference*, pages 301–312. ACM, 2014.

[136] Jia Rao, Kun Wang, Xiaobo Zhou, and Cheng-Zhong Xu. Optimizing virtual machine scheduling in numa multicore systems. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pages 306–317. IEEE, 2013.

[137] Theodore S Rappaport et al. *Wireless communications: principles and practice*, volume 2. prentice hall PTR New Jersey, 1996.

[138] Carl Edward Rasmussen and Christopher K. I. Williams. *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press, 2005.

[139] Carl Edward Rasmussen and Christopher K. I. Williams. *Gaussian Processes for Machine Learning*. The MIT Press, 2006.

[140] Peter J Rousseeuw. Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *Journal of computational and applied mathematics*, 20:53–65, 1987.

[141] N. Roy, A. Dubey, and A. Gokhale. Efficient autoscaling in the cloud using predictive models for workload forecasting. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 500–507. IEEE, 2011.

[142] Mahadev Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The Case for VM-Based Cloudlets in Mobile Computing. *Pervasive Computing, IEEE*, 8(4):14–23, 2009.

[143] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Caceres, and Nigel Davies. The case for vm-based cloudlets in mobile computing. *IEEE pervasive Computing*, 8(4), 2009.

[144] Mahadev Satyanarayanan, Zhuo Chen, Kiryong Ha, Wenlu Hu, Wolfgang Richter, and Padmanabhan Pillai. Cloudlets: at the leading edge of mobile-cloud convergence. In *Mobile Computing, Applications and Services (MobiCASE), 2014 6th International Conference on*, pages 1–9. IEEE, 2014.

[145] Nishanth Shankaran, John S Kinnebrew, Xenofon D Koutsoukas, Chenyang Lu, Douglas C Schmidt, and Gautam Biswas. An Integrated Planning and Adaptive Resource Management Architecture for Distributed Real-time Embedded Systems. *Computers, IEEE Transactions on*, 58(11):1485–1499, 2009.

[146] Prateek Sharma, David Irwin, and Prashant Shenoy. Portfolio-driven resource management for transient cloud servers. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 1(1):5, 2017.

[147] Muhammad Bilal Sheikh, Umar Farooq Minhas, Omar Zia Khan, Ashraf Aboulnaga,

Pascal Poupart, and David J Taylor. A bayesian approach to online performance modeling for database appliances using gaussian models. In *Proceedings of the 8th ACM international conference on Autonomic computing*, pages 121–130. ACM, 2011.

[148] Shashank Shekhar, Yogesh Barve, and Aniruddha Gokhale. Understanding performance interference benchmarking and application profiling techniques for cloud-hosted latency-sensitive applications. In *Proceedings of the10th International Conference on Utility and Cloud Computing*, UCC '17, pages 187–188, New York, NY, USA, 2017. ACM.

[149] Shashank Shekhar, Ajay Chhokra, Anirban Bhattacharjee, Guillaume Aupy, and Aniruddha Gokhale. INDICES: Exploiting Edge Resources for Performance-Aware Cloud-Hosted Services. In *IEEE 1st International Conference on Fog and Edge Computing (ICFEC)*, pages 75–80, Madrid, Spain, May 2017.

[150] Shashank Shekhar, Michael Walker, Hamzah Abdelaziz, Faruk Caglar, Aniruddha Gokhale, and Xenofon Koutsoukos. A Simulation-as-a-Service Cloud Middleware. *Journal of the Annals of Telecommunications*, 74(3-4):93–108, 2016.

[151] 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*, page 5. ACM, 2011.

[152] Shipyard. Shipyard Project, 2014. Last accessed: 10/11/2014.

[153] Francisco Airton Silva, Paulo Maciel, and Rubens Matos. Smartrank: a smart scheduling tool for mobile cloud computing. *The Journal of Supercomputing*, pages 1–24, 2015.

[154] Stephen Soltesz, Herbert Pötzl, Marc E Fiuczynski, Andy Bavier, and Larry Peterson. Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 275–287. ACM, 2007.

[155] Thamarai Selvi Somasundaram and Kannan Govindarajan. CLOUDRB: A Framework for Scheduling and Managing High-Performance Computing (HPC) Applications in Science Cloud. *Future Generation Computer Systems*, 34:47–65, 2014.

[156] Bruno Sousa, Zhongliang Zhao, Morteza Karimzadeh, David Palma, Vitor Fonseca, Paulo Simoes, Torsten Braun, Hans Van Den Berg, Aiko Pras, and Luis Cordeiro. Enabling a mobility prediction-aware follow-me cloud model. In *Local Computer Networks (LCN), 2016 IEEE 41st Conference on*, pages 486–494. IEEE, 2016.

[157] Simon Spinner, Samuel Kounev, Xiaoyun Zhu, Lei Lu, Mustafa Uysal, Anne Holler, and Rean Griffith. Runtime vertical scaling of virtualized applications via online model estimation. In *Self-Adaptive and Self-Organizing Systems (SASO), 2014 IEEE Eighth International Conference on*, pages 157–166. IEEE, 2014.

[158] Kaixin Sui, Mengyu Zhou, Dapeng Liu, Minghua Ma, Dan Pei, Youjian Zhao, Zimu Li, and Thomas Moscibroda. Characterizing and improving wifi latency in large-scale operational networks. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '16, pages 347–360, New York, NY, USA, 2016. ACM.

[159] Srikanth Sundaresan, Walter De Donato, Nick Feamster, Renata Teixeira, Sam Crawford, and Antonio Pescapè. Broadband internet performance: a view from the gateway. In *ACM SIGCOMM computer communication review*, volume 41, pages 134–145. ACM, 2011.

[160] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2818–2826, 2016.

[161] F Tao, L Zhang, VC Venkatesh, Y Luo, and Y Cheng. Cloud manufacturing: a computing and service-oriented manufacturing model. *Proceedings of the Institution of Mechanical Engineers, Part B: Journal of Engineering Manufacture*, 225(10):1969–1976, 2011.

[162] Luis Tomás and Johan Tordsson. Improving cloud infrastructure utilization through overbooking. In *Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference*, CAC '13, pages 5:1–5:10, New York, NY, USA, 2013. ACM.

[163] Franco Travostino, Paul Daspit, Leon Gommans, Chetan Jog, Cees De Laat, Joe Mambretti, Inder Monga, Bas Van Oudenaarde, Satish Raghunath, and Phil Yonghui Wang. Seamless live migration of virtual machines over the man/wan. *Future Generation Computer Systems*, 22(8):901–907, 2006.

[164] Ruben Van den Bossche, Kurt Vanmechelen, and Jan Broeckhove. Cost-efficient Scheduling Heuristics for Deadline Constrained Workloads on Hybrid Clouds. In *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on*, pages 320–327. IEEE, 2011.

[165] Kurt Vanmechelen, Silas De Munck, and Jan Broeckhove. Conservative Distributed Discrete Event Simulation on Amazon EC2. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, pages 853–860. IEEE Computer Society, 2012.

[166] Nedeljko Vasić, Dejan Novaković, Svetozar Miučin, Dejan Kostić, and Ricardo

Bianchini. Dejavu: accelerating resource allocation in virtualized environments. In *ACM SIGARCH computer architecture news*, volume 40, pages 423–436. ACM, 2012.

[167] Steve Vinoski. Advanced message queuing protocol. *IEEE Internet Computing*, 10(6):87–89, 2006.

[168] Shiqiang Wang, Rahul Urgaonkar, Ting He, Kevin Chan, Murtaza Zafer, and Kin K Leung. Dynamic service placement for mobile micro-clouds with predicted future costs. *IEEE Transactions on Parallel and Distributed Systems*, 28(4):1002–1016, 2017.

[169] Y Angela Wang, Cheng Huang, Jin Li, and Keith W Ross. Estimating the performance of hypothetical cloud service deployments: A measurement-based approach. In *INFOCOM, 2011 Proceedings IEEE*, pages 2372–2380. IEEE, 2011.

[170] Yimeng Wang, Yongbo Li, and Tian Lan. Capitalizing on the promise of ad prefetching in real-world mobile systems. In *2017 IEEE 14th International Conference on Mobile Ad Hoc and Sensor Systems (MASS)*, pages 162–170. IEEE, 2017.

[171] Yingzi Wang, Nicholas Jing Yuan, Defu Lian, Linli Xu, Xing Xie, Enhong Chen, and Yong Rui. Regularity and conformity: Location prediction using heterogeneous mobility data. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1275–1284. ACM, 2015.

[172] Patrick Wendell, Joe Wenjie Jiang, Michael J Freedman, and Jennifer Rexford. Donar: decentralized server selection for cloud services. *ACM SIGCOMM Computer Communication Review*, 40(4):231–242, 2010.

[173] wikipedia.org. Page view statistics for Wikimedia projects, 2017. URL: https://dumps.wikimedia.org/other/analytics/, Last accessed: 2017-11-22.

[174] Timothy Wood, KK Ramakrishnan, Prashant Shenoy, and Jacobus Van der Merwe. Cloudnet: dynamic pooling of cloud resources by live wan migration of virtual machines. In *ACM Sigplan Notices*, volume 46, pages 121–132. ACM, 2011.

[175] Miguel G Xavier, Marcelo Veiga Neves, Fabio D Rossi, Tiago C Ferreto, Timoteo Lange, and Cesar AF De Rose. Performance Evaluation of Container-based Virtualization for High Performance Computing Environments. In *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*, pages 233–240. IEEE, 2013.

[176] Chi Xu, Xi Chen, Robert P Dick, and Zhuoqing Morley Mao. Cache contention and application performance prediction for multi-core systems. In *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*, pages 76–86. IEEE, 2010.

[177] Fengyuan Xu, Chiu C Tan, Qun Li, Guanhua Yan, and Jie Wu. Designing a practical access point association protocol. In *INFOCOM, 2010 Proceedings IEEE*, pages 1–9. IEEE, 2010.

[178] Hongfa Xue, Yurong Chen, Fan Yao, Yongbo Li, Tian Lan, and Guru Venkataramani. Simber: Eliminating redundant memory bound checks via statistical inference. In *IFIP International Conference on ICT Systems Security and Privacy Protection*, pages 413–426. Springer, 2017.

[179] Ying Yan, Yanjie Gao, Yang Chen, Zhongxin Guo, Bole Chen, and Thomas Moscibroda. Tr-spark: Transient computing for big data analytics. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, pages 484–496. ACM, 2016.

[180] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers. In

ACM SIGARCH Computer Architecture News, volume 41, pages 607–618. ACM, 2013.

[181] Youngseok Yang, Geon-Woo Kim, Won Wook Song, Yunseong Lee, Andrew Chung, Zhengping Qian, Brian Cho, and Byung-Gon Chun. Pado: A data processing engine for harnessing transient resources in datacenters. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 575–588. ACM, 2017.

[182] Fan Yao, Yongbo Li, Yurong Chen, Hongfa Xue, Tian Lan, and Guru Venkataramani. Statsym: vulnerable path discovery through statistics-guided symbolic execution. In *Dependable Systems and Networks (DSN), 2017 47th Annual IEEE/IFIP International Conference on*, pages 109–120. IEEE, 2017.

[183] Lenar Yazdanov and Christof Fetzer. Vertical scaling for prioritized vms provisioning. In *Cloud and Green Computing (CGC), 2012 Second International Conference on*, pages 118–125. IEEE, 2012.

[184] Li Zhang, Xiaoqiao Meng, Shicong Meng, and Jian Tan. K-scope: Online performance tracking for dynamic cloud applications. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*, pages 29–32, 2013.

[185] Qi Zhang, Lu Cheng, and Raouf Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of internet services and applications*, 1(1):7–18, 2010.

[186] Wuyang Zhang, Yi Hu, Yanyong Zhang, and Dipankar Raychaudhuri. Segue: Quality of service aware edge cloud service migration. In *8th IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 2016.

[187] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. Cpi2: Cpu performance isolation for shared compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13,

pages 379–391, New York, NY, USA, 2013. ACM.

[188] 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 *47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 406–418. IEEE Computer Society, 2014.

[189] Bowen Zhou, Amir Vahid Dastjerdi, Rodrigo N Calheiros, Satish Narayana Srirama, and Rajkumar Buyya. A context sensitive offloading scheme for mobile cloud computing service. In *Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on*, pages 869–876. IEEE, 2015.

[190] Xiaomin Zhu, Huangke Chen, Laurence T Yang, and Shu Yin. Energy-Aware Rolling-Horizon Scheduling for Real-Time Tasks in Virtualized Cloud Data Centers. In *High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC), 2013 IEEE 10th International Conference on*, pages 1119–1126. IEEE, 2013.

[191] Paolo Zuliani, André Platzer, and Edmund M Clarke. Bayesian Statistical Model Checking with Application to Stateflow/Simulink Verification. *Formal Methods in System Design*, 43(2):338–367, 2013.