ESCOLA POLITÉCNICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
DOUTORADO EM CIÊNCIA DA COMPUTAÇÃO

VINÍCIUS MEYER

INTERFERENCE-AWARE CLOUD SCHEDULING ARCHITECTURE FOR DYNAMIC LATENCY-SENSITIVE WORKLOADS

Porto Alegre
2022

PONTIFICAL CATHOLIC UNIVERSITY OF RIO GRANDE DO SUL
SCHOOL OF TECHNOLOGY
COMPUTER SCIENCE GRADUATE PROGRAM

INTERFERENCE-AWARE CLOUD SCHEDULING ARCHITECTURE FOR DYNAMIC LATENCY-SENSITIVE WORKLOADS

VINÍCIUS MEYER

Doctoral Thesis submitted to the Pontifical Catholic University of Rio Grande do Sul in partial fulfillment of the requirements for the degree of Ph. D. in Computer Science.

Advisor: Prof. César Augusto Fonticielha De Rose

Porto Alegre
2022

Ficha Catalográfica

M613i Meyer, Vinícius

Interference-Aware Cloud Scheduling Architecture for Dynamic Latency-sensitive Workloads / Vinícius Meyer. – 2022.
104.
Tese (Doutorado) – Programa de Pós-Graduação em Ciência da Computação, PUCRS.

Orientador: Prof. Dr. César Augusto Fonticielha De Rose.

1. Interference-aware Scheduling. 2. Dynamic Latency-sensitive Workloads. 3. Machine Learning. 4. Resource Management. 5. Cloud Computing. I. De Rose, César Augusto Fonticielha. II. Título.

Elaborada pelo Sistema de Geração Automática de Ficha Catalográfica da PUCRS
com os dados fornecidos pelo(a) autor(a).
Bibliotecária responsável: Loiva Duarte Novak CRB-10/2079

VINÍCIUS MEYER

INTERFERENCE-AWARE CLOUD SCHEDULING ARCHITECTURE FOR DYNAMIC LATENCY-SENSITIVE WORKLOADS

This Doctoral Thesis has been submitted in partial fulfillment of the requirements for the degree of Ph. D. in Computer Science, of the Computer Science Graduate Program, School of Technology of the Pontifical Catholic University of Rio Grande do Sul

Sanctioned on March 21, 2022.

COMMITTEE MEMBERS:

I dedicate this work to my family.

“Logic will get you from A to B. Imagination will take you everywhere.”
(Albert Einstein)

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to those who supported me throughout all my PhD years and made this dissertation possible. Initially, I would like to thank my advisor, Prof. Cesar A. F. De Rose, for creating the opportunity for me to pursue doctoral studies and providing me invaluable guidance and support during these years.

Thank you to all co-authors of my research publications: Matheus L. Da Silva, Dionatrã F. Kirchoff, Miguel G. Xavier, Uillian L. Ludwig, and Jakub Krzywda for collaboration, exchanging ideas, and providing feedback

I would also like to thank my colleagues and friends from PUCRS. The experience and knowledge acquired during the years of study at Polytechnic School proved to be very important during the development of this work.

Thank you to my mother, for her love and continuous support throughout my life.

During the doctoral period, I became the father of two wonderful children, Anna and Luca, thank you for being part of this moment and part of my life. Finally, I would like to thank my wife Nádia for her unconditional support, understanding, care, and love.

ARQUITETURA DE ESCALONAMENTO EM NUVEM CIENTE DE INTERFERÊNCIA PARA CARGAS DE TRABALHO DINÂMICAS E SENSÍVEIS À LATÊNCIA

RESUMO

Os sistemas de computação continuam a evoluir para facilitar o aumento do desempenho ao processar cargas de trabalho em grandes data centers. A virtualização é uma tecnologia que permite que vários aplicativos sejam executados em um único computador físico, gerando várias vantagens, incluindo rápido provisionamento de recursos e melhor utilização de hardware. Os provedores de computação em nuvem adotam essa estratégia para usar sua infraestrutura de forma mais eficiente, reduzindo o consumo de energia. Apesar disto, nossas pesquisas na área têm mostrado que vários serviços em nuvem competindo por recursos compartilhados são suscetíveis à interferência entre aplicativos, o que pode levar a uma degradação significativa do desempenho e, consequentemente, a um aumento de quebras no número de acordos de nível de serviço. No entanto, o escalonamento de recursos de última geração em ambientes virtualizados ainda depende principalmente da capacidade dos recursos, adotando heurísticas como o bin-packing, ignorando essa fonte de sobrecarga. Mas, nos últimos anos, o escalonamento com reconhecimento de interferência ganhou força, com a investigação de maneiras de classificar os aplicativos em relação ao seu nível de interferência e a proposta de modelos estáticos e políticas para o escalonamento de aplicativos co-hospedados em nuvem. Os resultados preliminares nesta área já mostram uma melhoria considerável na redução de quebra de SLAs, mas acreditamos fortemente que ainda existem oportunidades de melhoria nas áreas de classificação de aplicações e estratégias de escalonamento dinâmico. Portanto, o objetivo principal deste trabalho é estudar o comportamento dos perfis de interferência dos aplicativos em nuvem ao longo de todo o seu ciclo de vida e sua suscetibilidade às variações da carga de trabalho, em busca de oportunidades para melhorar o compartilhamento de recursos em ambientes virtualizados com novas estratégias de escalonamento dinâmico. Para tanto, exploramos algumas questões específicas de pesquisa relacionadas à natureza dinâmica do processo, tais como: Como classificar aplicações baseadas na interferência de recursos em tempo real? Quando as classificações devem ser executadas? Quantos níveis devem ser usados? Quando devem ser escalonados? Quais são as compensações com o custo de migração? Para responder a todas essas perguntas, criamos uma arquitetura de escalonamento com reconhecimento de interferência que integra esses tópicos mencionados para lidar com cargas de trabalho dinâmicas sensíveis à latência em ambientes virtualizados. As contribuições deste estudo são: (i) uma análise do impacto das variações da carga de trabalho no perfil de interferência de aplicativos em nuvem; (ii) uma forma precisa e otimizada de classificar aplicativos em tempo real; (iii) uma nova estratégia de escalonamento com reconhecimento de interferência dinâmica para aplicativos em nuvem; e (iv) uma arquitetura dinâmica que combina as técnicas acima para entregar um escalonamento eficiente com reconhecimento de interferência em ambientes virtualizados. Os resultados evidenciaram que nossa arquitetura melhorou em média 25% a eficiência geral de utilização de recursos quando comparada com estudos relacionados.

Palavras-Chave: Escalonamento Ciente de Interferência, Cargas de Trabalho Dinâmicas Sensíveis à Latência, Aprendizado de Máquina, Gerenciamento de Recursos, Computação em nuvem, Simulação.

INTERFERENCE-AWARE CLOUD SCHEDULING ARCHITECTURE FOR DYNAMIC LATENCY-SENSITIVE WORKLOADS

ABSTRACT

Computing systems continue to evolve to facilitate increased performance when processing workloads in large data centers. Virtualization technology enables multiple applications to be created and executed on a single physical computer, yielding various advantages, including rapid provisioning of resources and better utilization of hardware. Cloud computing providers have adopted this strategy to use their infrastructure more efficiently, reducing energy consumption. However, our research in this field has shown that multiple cloud services contending for shared resources are susceptible to cross-application interference, which can lead to significant performance degradation and consequently an increase in the number of broken service level agreements (SLA). Nevertheless, state-of-the-art resource scheduling in virtualized environments still relies mainly on resource capacity, adopting heuristics such as bin-packing, thus overlooking this source of overhead. But in recent years interference-aware scheduling has gained traction, and applications are now being classified based on their interference level and the proposal of static cost models and policies for scheduling co-hosted cloud applications. Preliminary results in this area already show a considerable improvement in the reduction of broken SLAs, yet we strongly believe that there are still opportunities to improve in the areas of application classification and dynamic scheduling strategies. Therefore, this work's primary goal is to study the behavior of cloud applications' interference profiles over their entire life cycle, and their susceptibility to workload variations, looking for opportunities to improve resource sharing in virtualized environments with novel dynamic scheduling strategies. To this end, we explored some specific research questions related to the dynamic nature of the process, such as: How can applications be classified based on resource interference in real-time? When should classifications be executed? How many levels should be used? When should they be scheduled? What are the trade-offs with migration cost? To answer all of these questions, we created an interference-aware scheduling architecture that integrates the aforementioned topics to better manage dynamic latency-sensitive workloads in virtualized environments. The contributions of this study are: (i) an analysis of the impact of workload variations in the interference profile of cloud applications; (ii) a precise and optimized way to classify applications in real-time; (iii) a novel dynamic interference-aware scheduling strategy for cloud applications; and (iv) a dynamic architecture that combines the above techniques to deliver efficient interference-aware scheduling in virtualized environments. Our results show an average 25% improvement of overall resource utilization efficiency with our architecture compared to related studies.

Keywords: Interference-aware Scheduling, Dynamic Latency-sensitive Workloads, Machine Learning, Resource Management, Cloud Computing, Simulation.

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

CAAS – Container as a Service

CMT – Cache Monitoring Technology

CPU – Central Process Unit

CRIU – Checkpoint/Restore In Userspace

DDR4 – Double Data Rate 4

EB – Emulated Browser

GB – Giga Byte

IDI – Interference Degradation Indexes

IMC – Integrated Memory Controller

INTP – Interference Profiler

IPC – Inter-process Communication

JRI – Java-R-Integration

LLC – Last Level Cache

ML – Machine Learning

OCPD – Online Change Point Detection

PC – Principal Component

PCA – Principal Component Analysis

QOS – Quality of Service

RAM – Random-access Memory

RMID – Resource Monitoring ID

RQ – Research Question

SA – Simulated Annealing

SLA – Service Level Agreement

SMP – Symmetric Multiprocessing

SVM – Support Vector Machines

TB – Tera Byte

VCORES – Virtual Cores

VM – Virtual Machine

CONTENTS

1. INTRODUCTION

Cloud computing has received a considerable amount of attention in the last decade and is widely accepted as the most promising technology for managing data utilization and resources, as well as delivering various IT services [PKK19]. Many latency-sensitive applications have begun to take advantage of cloud computing due to the promise of unlimited computing resources and the pay-as-you-go model [AKF15]. A cloud system offers capabilities through virtualization techniques, executing multiple virtual instances on each physical machine in a data center. This strategy allows cloud computing providers to use their infrastructure more efficiently, reducing energy consumption expenses [ZAL19]. However, related work [XNR$^+$13, Xav19] shows that several cloud-services contending for shared resources can generate cross-application interference, which may lead to significant performance degradation and consequently to an increase in the number of broken Service Level Agreements (SLAs).

Efficient and automatic resource scheduling strategies are essential [ZCG$^+$18] for virtualized platforms to deliver SLA guarantees for high user satisfaction. Therefore, resource scheduling is a core function of the Cloud Computing providers and a central component when coordinating all the other platform elements to deliver performance-oriented solutions [ZCV$^+$19]. Large data centers generally schedule resources through heuristics, such as bin-packing, which only considers resource capacity aspects, overlooking the original source of overhead [CRO$^+$15].

In the search for alternatives, previous work from our research group [LXK$^+$19] explored scheduling policies based on interference generated by co-allocated applications. An attraction/repulsion method built upon the workload profile of each application was proposed in order to get around the traditional concept of simply observing resource usage and capacity. Web applications were investigated, since they are a latency-sensitive category that is not well handled by proposals found in the literature and could profit from our novel strategies [IEM18].

Performance interference among applications is known to adversely impact the Quality of Service (QoS) properties and Service Level Agreements of applications [NKG10]. This becomes particularly more problematic for latency-sensitive applications. Dynamic service demands and workload profiles further increase the challenges cloud service providers face when managing resources on-demand to satisfy SLAs while minimizing operational costs [ZCB10]. Therefore, any solution that addresses these issues requires an approach that accounts for workload variability and performance interference due to the dynamic nature of the process [SAB$^+$18]. But a dynamic approach has its own challenges, such as: How to classify applications in real-time based on the interference they generate? When to execute the classification? When to schedule them and how to tradeoff migration costs?

Given the aforementioned issues, it is crucial to investigate a performance evaluation, resource configuration, and workload scheduling to reduce the SLA violations in latency-sensitive applications within virtualized data centers. It is not easy to find a solution that comprehensively covers all of the subjects above. Although recently proposed approaches in interference-aware related-studies present a significant improvement in resource usage, we strongly believe that there are still research opportunities available for interference classification and dynamic scheduling strategies. After analyzing the results of our previous work, we formulated the following thesis: dynamic interference-aware scheduling architecture, which analyzes workload variations of latency-sensitive applications over time, could further improve the utilization of consolidated resources in data centers while reducing costs and minimizing SLA violations. Our main goal in this study was to create a scheduling architecture, considering interference aspects among co-located latency-sensitive applications that have dynamic workload patterns over time. Therefore, the following research questions (RQ) must be addressed:

- **RQ1** Do the interference levels of applications with dynamic workloads change over time?
- **RQ2** Could a dynamic interference classification system lead to better resource scheduling?
- **RQ3** What architectural changes are needed to move from static to dynamic interference-aware scheduling?
- **RQ4** Is there a way to implement these changes so that the resulting overhead of a dynamic strategy will not invalidate the improved scheduling gains?

Understanding the interference behavior over time from latency-sensitive applications which have dynamic workloads (RQ1) made it possible to verify if a dynamic interference classification system would result in better resource scheduling (RQ2). Then we began to explore which architectural characteristics needed to be changed in order to move from static to dynamic interference-aware scheduling (RQ3). Consequently, an analysis was performed to determine if the resulting overhead of a dynamic strategy would invalidate the improved scheduling gains (RQ4).

1.1 Motivation and Challenges

Our research group is interested in studying cross-application interference and its impact on application performance degradation and hardware utilization. Xavier et al. [XMLDR16] analyzed the performance interference tolerated by multi-tenant e-commerce cloud databases in resource-sharing infrastructures. They concluded that multiple-different workloads (e.g. memory-/CPU-intensive, and e-commerce applications) may be consolidated with database systems to minimize performance interference and increase resource-efficiency. Recently, Ludwig et al. [LXK$^+$19] proposed a method that profiles each application workload and delivers better scheduling decisions in order to efficiently use the available resources. Although this approach improves resource usage and scheduling decisions, it uses a static classification method. Therefore, it creates a single label over the entire execution of the application and according to its interference metrics, this reduces the performance degradation across the entire system.

Following this trend, we started to move towards interference-aware dynamic scheduling. We noticed that if a static classification had the potential to substantially improve resource utilization, a dynamic one could further improve resource scheduling. After analyzing the outcomes of preliminary experiments, we confirmed that there was indeed a great research opportunity. However, changing the scheduling architecture from static to dynamic is a challenging task and some modifications are required to adjust the system due to the following issues:

- Dynamic resource scheduling needs to have a classification method that adapts its outcomes according to workload variations;
- Since dynamic workloads presents variations over time, a method must analyze time-series information to find the right time to make scheduling decisions;
- It is essential to build a manager module to coordinate and schedule all resources and application executions at runtime;
- Implementing all these features will most likely generate system overhead. We need to investigate methods to keep this overhead as low as possible so that scheduling outcomes from the proposed approach are not invalidated.

1.2 Document Organization

The rest of this document is organized as follows. Chapter 2 introduces the background concepts of this doctoral dissertation. Chapter 3 explains how applications are profiled and presents a general analysis of dynamic workloads and the interference they generate. Chapter 4 demonstrates a dynamic classification scheme and its impact on resource scheduling. Chapter 5 introduces in detail the dynamic interference-aware scheduling architecture, which is the main goal of this doctoral dissertation. Chapter 6 presents related work in the literature. Finally, Chapter 7 offers conclusions and directions for future research.

2. BACKGROUND AND STATE-OF-THE-ART

This chapter provides the essential context, key concepts, and state-of-the-art research intrinsic to this dissertation. It describes how resource scheduling is currently handled, as well as its technologies and capabilities. It is also presents performance interference aspects, interference classification methods, dynamic latency-sensitive workloads, and some limitations found in previous contemporary related studies.

2.1 Resource Scheduling in Cloud Infrastructures

Cloud computing is an emerging technology that has become increasingly popular in recent years. It allows customers to deploy its services, greatly simplifying the process of acquiring and releasing resources to run applications while only paying for the resources allocated (pay-as-you-go model). According to NIST [1] definition, "Cloud computing is a model for enabling ubiquitous, convenient, and on-demand network access to a shared pool of configurable computing resources that can be rapidly provisioned and released with minimal management effort or service provider interaction". To allow virtualized platforms to deliver SLA guarantees for high user satisfaction, efficient and automatic resource scheduling strategies are essential [ZCG+18]. Resource scheduling is a core function of cloud computing providers and a central component to coordinate all other platform features to deliver performance-oriented solutions [ZCV+19]. However, resource scheduling is a challenging task since cloud service providers must deliver a sufficient amount of resources while meeting users' QoS requirements such as deadline, execution time, and budget restrictions. This section describes how cloud providers currently handle resource scheduling. Also, it presents virtualization technologies, performance interference issues, and machine learning techniques used to support resource management decisions in cloud data centers.

2.1.1 Virtualization Technology

Orchestration systems in data centers must be highly elastic and scalable infrastructures that allow the dynamic allocation of different resources (such as compute, storage, networking, software, or a service) at the right location and in a short period of time, enabling the deployment of applications [TRA15]. The elasticity in cloud environments is obtained abstracting physical resources from an underlying layer through virtualization.

[1]http://www.nist.gov/

There are different virtualization technologies, but the two most relevant in the cloud computing landscape are *Hardware virtualization* and *System-level virtualization*:

- *Hardware virtualization* (Hypervisor) abstracts the underlying hardware layers to enable complete operating systems to run inside the Hypervisor as if they were applications. Paravirtualization solutions (Xen [2]) and hardware virtualization solutions (KVM [3]), together with hardware-specific support, integrated into a modern CPU (Intel VT-x and AMD-V), can achieve a low level of overhead due to the new layer added between the virtual instance and the hardware.

- *System-level virtualization* (Container) is based on fast and lightweight process virtualization and allows an entire application to run on every Linux distribution, with its dependencies in a virtual container. It provides its users an environment as close as possible to a standard Linux distribution. Due to the fact that containers are lighter weight than VMs (Figure 2.1), the same host can achieve higher densities with containers than with VMs. This approach has radically decreased the start-up time of instances as well as processing and storage overhead, which are typical drawbacks of Hypervisor-based virtualization [Ros14].

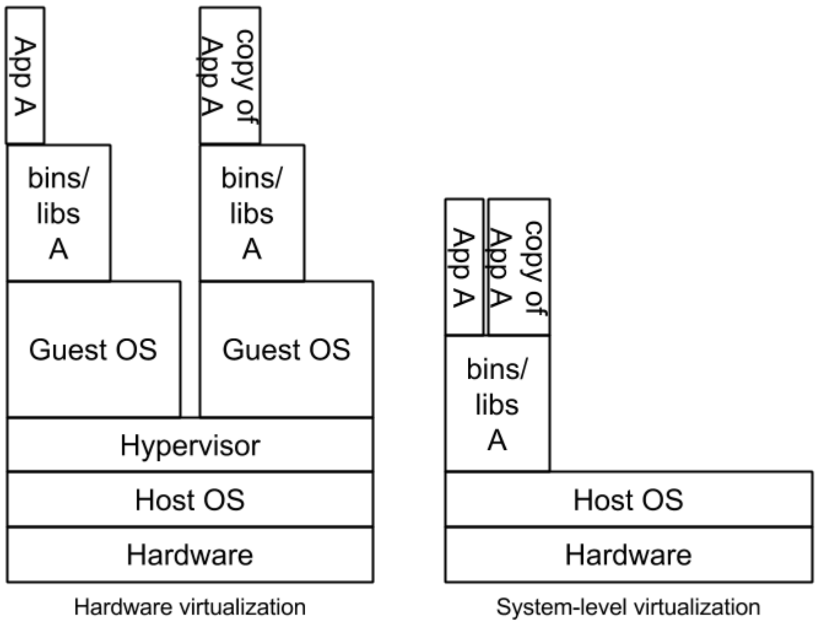

Figure 2.1 – Comparison of containers vs VMs footprint on the host system [TRA15].

Containerization is the state-of-art virtualization of the cloud platform [Mer14]. Containers only need seconds to bootstrap, initiate, versus minutes for a regular VM

[2]https://xenproject.org/
[3]https://www.linux-kvm.org/

[ZTL+19] (seen in Table 2.1). Container technologies effectively virtualize the operating system and are becoming popular in cloud computing. By encapsulating runtime contexts of software components and services, containers improve portability and efficiency for cloud application deployment [HZdLZ20]. In addition, a container can be scaled out/in within a minute, and consequently can react immediately when it encounters a possible unforeseen crash. Therefore, containers are capable of tolerating fluctuating stress and reducing overhead [Sch14], coincidentally precisely the features which auto-scaling needs.

Table 2.1 – Comparison between container and virtual machine [ZTL+19].

	Kinds of virtualization	
Performances	**Container**	**Virtual Machine**
Size	Megabytes	Hundreds Megabytes
Start time	Seconds	Minutes
Management overhead	Low	High
Portability	High	Low

A container holds self-contained, ready-to-deploy packaged parts of applications and, if necessary, middleware and business logic (in binaries and libraries) to run the applications [PBSJ19]. Containerization facilitates the step from single applications in containers to clusters of container hosts that can run containerized applications across cluster hosts [PL15]. The latter benefits from the built-in interoperability of containers. Individual container hosts are grouped into interconnected clusters. There are many well known container solutions, such as: Docker [4], Linux LXC [5], OpenVZ [6] and Linux-VServer [7].

Cloud Simulation Tools

As previously mentioned, cloud data centers have been widely adopted by companies to purchase resources like computing, storage, and networking. However, conducting research or practical tests on live cloud environments for individuals or small institutions is very difficult due to the costs involved in setting up a cloud ecosystem. To tackle this issue, the research community has developed several cloud simulation tools. A cloud simulator helps to model various kinds of cloud applications by creating data centers, virtual machines, and other capabilities that can be configured appropriately. Since we aimed to test our solution within an environment as close to a real scenario as possi-

[4]http://www.docker.com
[5]https://linuxcontainers.org/
[6]https://openvz.org/
[7]http://www.linux-vserver.org

ble, we decided to perform some analysis with a cloud simulation tool as well. This section presents cloud simulation research initiatives and explains which was adopted here.

MDCSim [LSN+09] is a commercial, comprehensive, flexible, and scalable simulation tool that is used to simulate a framework to perform detailed performance and power analysis of multi-tier data centers. It is an event-driven tool that offers IaaS to multiple clients. Performance is measured by calculating throughput and response time. The data center topology is fed as a directed graph by the MDCSim network package. MDCSim is a library and does not provide a user interface, which limits its modeling and simulation capabilities when for complex business configurations.

GreenCloud [KBK12] is an extension of the NS2 [IH12] simulator for energy-aware networking in cloud infrastructures. This tool considers specific data center components, such as servers, switches, and links, which makes it an exceptionally powerful tool for simulating power consumption because it considers the power consumed by computing and communication elements of the available data center, unlike other simulators such as MDCSim [LSN+09] and CloudSim [CRB+11] which do not. GreenCloud uses mathematical models of power consumption to simulate workload provision. The graphic user interface is limited to a certain level due to the number of the simulated cloud configurations.

ICanCloud [NLC+12] is a discrete event simulator that has been designed using the E-mc2 framework [CNLC13], a formal framework for advanced analysis of energy modelling in cloud computing. The main design aspects of iCanCloud are to provide a platform for large experiments, which is a difficult with all of the previously described simulators. iCanCloud allows the Hypervisor to be modeled, which permits flexibility for integrating any cloud brokering policy. The graphic user interface of iCanCloud is proving to be the best in terms of complex configuration, because it can convert from a single VM to large cloud computing systems composed of thousands of machines. iCanCloud is also open source and can be used by the application developer or regular cloud users.

CloudSim [CRB+11] is the most widely spread cloud simulator and also the most sophisticated by far. It was first developed as an add-on-top of the grid network simulator GridSim [Cas01]. CloudSim is a completely customizable tool which supports modeling, creating one or more VMs, and mapping tasks to the appropriate virtual machine. This gives CloudSim the ability to handle a complex simulation environment. It mainly targets application developers or testers, allowing one to configure several variables such as the number of users, data centers, and cloud resources along with the location of both users and data centers. In addition to many other studies extending CloudSim, such as [BB12, GMC+13, XRR+17, KMX+20], there is one in particular that supports Container as a Service (CaaS), namely ContainerCloudSim [PDCB17]. This extension provides a platform for modeling and simulating containerized cloud computing environments. Our decision to apply this tool in our work is two-fold. Firstly, our research group participated in its development, giving us specific knowledge about its features and usage. Moreover, it is the most appropriate current simulation tool.

2.1.2 Performance Interference

With the advent of resource sharing techniques, physical machines can host multiple applications. Even though the use of resource sharing methods, such as virtualization or containerization, provide approaches to fairly share resource between co-hosted applications, when multiple services intensively use a resource at the same time, resource contention problems will occur. This is known as performance interference, and it may lead to severe performance degradation [CRO+15].

Virtualization technologies and server consolidation are the main drivers of high resource utilization in modern data centers. Combining virtual machines into the same server may lead to severe performance degradation, known as virtual machine interference. Supporting a higher virtual machine interference may result in a higher consolidation, while strict low interference requirements may demand more resources. Jersark and Ferreto [JF16] claim that applications are affected by other virtual machines, which use the same resource intensively in the same physical machine. Furthermore, each resource is affected differently. CPU intensive applications lead to performance degradation of 14%. Whereas, memory and disk I/O intensive applications, the performance degradation was as high as 90%. Therefore, performance interference is clearly a problem, and the performance degradation varies depending on resource use.

Performance interference affects container-based environments as well. Applications with disk-intensive characteristics running over containers promote performance degradation that uses different resources intensively. Xavier et al. [Xav19] have tested several combinations of co-hosted workloads. While some of these combinations led to performance degradation up to 38%, workloads could also be combined without interference.

Cluster systems usually run several applications-often from different users concurrently, with individual applications competing for access to shared resources such as the file system or the network. Low application performance may be caused by interference from different sources. Shah et al. [SWZV13] state that mapping performance data related to shared resources into time slices can establish the simultaneity of application usage across jobs, which can be indicative of inter-application interference. In some cases, inter-application interference causes performance degradation of up to 50%.

A scheduler which considers interference issues is a common solution to minimize interference effects and improve application performance [ZT12, BRX13, ZRWZ14, WKNG19, CRO+15]. Zhu and Tung [ZT12] and Bu et al. [BRX13] present task scheduling strategies that include interference aspects, based on task performance prediction models to make better workload placement decisions. Proposed models achieve an average error of less than 8% and a speedup of 1.5 to 6.5 times for individual jobs, respectively. Zang et al. [ZRWZ14] and Wang et al. [WKNG19] developed interference-aware job scheduling algorithms to estimate the effect of interference among multiple instances of virtualized environments. Results show that the proposed scheduling algorithms reduce the execution time of tasks by an average of 6.5%.

Chen et al. [CRO+15] present CloudScope, a system for diagnosing interference for multi-tenant cloud systems. It (re)assigns virtual machines to physical machines and optimizes the Hypervisor configuration for different workloads. The interference-aware scheduler improves virtual machine performance by up to 10% of the default scheduler.

IntP: a Tool to Measure Interference

IntP [Xav19] is a tool for the quantification of per application resource sensitivity developed by our research group. IntP profiles running applications using low level kernel instrumentation, returning the utilization levels generated on each resource subsystem. Each module is responsible for each type of access on a specific resource at the infrastructure level, and produces the percentage of hardware resources utilization, per application, in an isolated fashion. This isolated measurement provides analytical information to the system to determine how much applications interfere with each other. The higher the metric, the more interference the application that is being profiled generates. More specifically, the tool determines the percentage of interference of the following metrics:

- *netp* - physical network;
- *nets* - network queue;
- *blk* - disk;
- *mbw* - memory bandwidth;
- *llcmr* - last-level cache miss rate;
- *llcocc* - last-level cache occupation;
- *cpu* - CPU utilization;

By using instrumentation techniques to infer application behaviors during runtime, IntP gives users information about how sensitive their applications are to hardware components and OS layers. Results provided by IntP can assist data center administrators create scheduling strategies to place applications that cause more noise between each other on different machines. In addition, the infrastructure becomes more balanced, since applications with different characteristics can be split up, making the data center resource efficient.

IntP extracts information from hardware in a manner that was not possible in the recent past. The advanced feature developed by Intel, called Intel Cache Monitoring Technology (CMT), makes it now entirely viable to collect information from cache utilization by running applications. This technology allows us to use an ID denominated Resource Monitoring ID (RMID) to metrify threads scheduled within the operation system. For each thread, there is one ID associated with it. Therefore, these metrics can be collected within an MSR interface. This was not possible before the creation of this technology [8].

IntP not only quantifies the interference application tasks cause on hardware resources, but also provides insights about application demands during runtime. This enables users to decide which piece of hardware is most likely to be the bottlenecked and to make the best informed decision about the queued application that fits best. Or if one application begins to affect others, it could be migrated to other machines to minimize interference and maximize performance. Figure 2.2 depicts IntP's overall architecture.

Here is an example to illustrate: suppose an application is running in a server and it is using half of its CPU and a third of its memory bandwidth capacity over its entire execution, hypothetically without variations. IntP will present, every second, the interference percentage of each resource generated by this application. Therefore, every second, IntP will provide the information presented in Table2.2. It is worth noting that, in the case of resource usage variations, these metrics will change.

Table 2.2 – IntP outcomes at a given second of an application execution.

netp	nets	blk	mbw	llcmr	llcocc	cpu
0	0	0	0.33	0	0	0.50

As mentioned before, IntP runs at the kernel layer, so it has low overhead over the system as a whole, not affecting the execution of applications. Since IntP is able to read hardware counters in an isolated manner, it can profile each application separately. Thus, this tool is considered the state-of-the-art interference profiler currently, which is why it was chosen for this work.

Methods to Classify Resource Interference

The interference classification method is the strategic basis of this work and will allow us to identify the resources utilization for each application's workload over time. Classification is a necessary step in the identification of tasks that can be scheduled at the

[8]https://github.com/intel/intel-cmt-cat

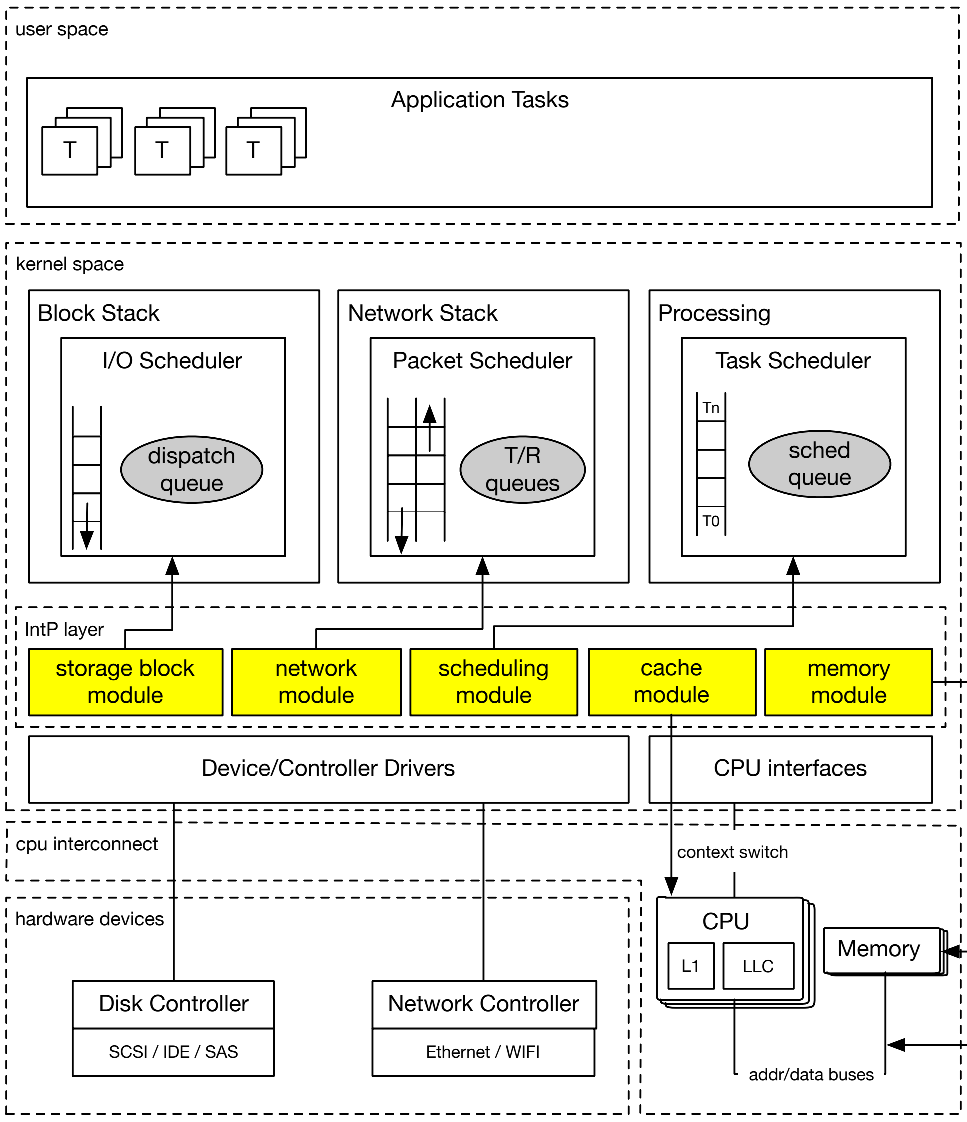

Figure 2.2 – IntP overall architecture [Xav19].

same virtual instance. An accurate classification leads to high task throughput in a virtual instance while an inaccurate classification may lead to interference among the tasks in same virtual instance which adversely affects application performance. The other major concern of classifying the tasks is how fast a task can be classified. A slow and heavy classification technique may increase the execution time of an application [KS17].

Caglar et al. [CSGK16] presents iSensitive, which is an intelligent, performance interference-aware resource management scheme for IoT cloud backends. iSensitive classifies the VMs based on their historic mean CPU, memory, and network usage features by using k-means clustering to classify the VMs in different classes. Experimental results evaluating iSensitive illustrate its advantages in deploying VMs to more aptly-suited host machines than traditional schemes, such as first-fit bin packing

Javadi and Gandhi [JG17] presents DIAL, an interference-aware load balancer for cloud environments. Interference detection is accomplished using a decision tree-based classifier to find the dominant source of resource contention. It monitors the impact of interference on user metrics such as CPU utilization, I/O wait time, etc. The model is trained by running controlled interference experiments using microbenchmarks and monitoring the metrics in each case. After training, the decision tree can classify the source of interference, even for unseen workloads, based on the observed metric values. Experimental results show that DIAL can reduce application tail latencies by as much as 70% and 48% compared to interference-oblivious and existing interference-aware load balancers, respectively.

Kumar and Setia [KS17] introduce an interference free scheduling algorithm with better performance for cloud computing applications. Random Forest [LW02] is used to classify applications into class labels: CPU intensive, Network-intensive, and Memory intensive. When each task is then recognized by the system, it is immediately classified and scheduled on the desired VM to better utilize the available resource.

In order to avoid cross-application I/O interference, Kougkas et al. [KDSL18] explore the negative effects of interference at the burst buffer layer. Their studies applied a Code-block Classifier [DKCS18] that categorizes the nodes into two classes: compute or I/O blocks. Their results claim that through better I/O scheduling, their work can triple the performance of existing state-of-the-art buffering management solutions and can lead to better resource utilization.

Ludwig et al. [LXK+19] propose placement algorithms based on resource interference profiled from applications. These algorithms apply a static classification method that produces an interference level for each resource within the application execution. The authors' approach achieved a reduction in response time of 10% when compared to related interference strategies. It is considered to be the most recent and relevant study done in the interference classification field directly related to this thesis.

2.1.3 Machine Learning Applied to Resource Scheduling

Machine learning (ML) is the discipline of teaching computers to predict outcomes or classify objects without being explicitly programmed for such tasks. One of its basic assumptions is that it is possible to build algorithms that can predict future, previously unseen, values using training data and the application of statistical techniques [DLCO19]. ML has gained increased traction and is being adopted in some critical planning and control areas [MMRB20]. Its success is mostly due to the availability of large datasets and the continuous improvements in the computational power of servers. Cloud providers rely increasingly on machine learning algorithms for prediction [AL17] and classification

[GdSTd20] purposes in real time services, lowering the cost of implementing and deploying resource scheduling solutions.

ML techniques are mainly grouped into three categories: (i) reinforcement learning, (ii) supervised, and (iii) unsupervised. Reinforcement learning allows a machine to learn behavior from the feedback it receives through interactions with an external environment. Unsupervised machine learning is used to draw conclusions from a given dataset consisting of input data without a labeled target. On the other hand, supervised machine learning techniques attempt to find out the relationship between input attributes and a target attribute. These can be further classified into two main categories: classification and regression. In regression, the output variable creates continuous values while in classification it produces class labels [AHK17]. In this study, two machine learning algorithms were adopted: (i) SVM for classification and (ii) K-Means for clustering.

SVM

Support Vector Machine is a supervised technique. It is derived from a hyperplane that maximizes the separating margin between the positive and negative classes in dimensional space. To achieve this, it considers the support vectors nearest to the minimum cost line. To accommodate curved lines or polygon regions, it scales the data into higher dimensions for predictions. Aiming to minimize performance degradation in cloud computing, Sotiriadis et al. [SBB18] introduced a virtual machine scheduling algorithm. It applied SVM to classify resource usage. Performance degradation was minimized by 19% and CPU real time was maximized by 2%. Sant'Ana et al. [SCSCC19] presented a real-time scheduling policy selection algorithm. They evaluated the use of logistic regression and SVM to perform the mapping of running queue job characteristics and machine states. Their SVM reached a classification method of up to 81%.

K-Means

Known as a clustering algorithm, K-means is an unsupervised method that attempts to split a given dataset into a fixed number of clusters. Each centroid (*k*) is an existing data point in the given input dataset. The process of classification and centroid adjustment is repeated until the values of the centroids stabilize. The final centroids will then be used to produce the final clustering. Gill et al. [GGS$^+$19] proposed a resource scheduling technique for holistic management of cloud computing resources. This method uses K-Means for clustering the workloads for execution on different set of resources. Their proposed technique was capable of reducing energy consumption by 20.1% while improving reliability and CPU utilization by 17.1% and 15.7% respectively. Xu et al. [XWHW19] formulated a generic job scheduling problem for parallel processing of big data in heterogeneous clusters and designed a K-Means based task scheduling algorithm, referred to as KMTS. Simulation results show that KMTS improves execution performance in existing models by 25% and 30% on average in single job scheduling and parallel job scheduling, respectively.

2.2 Dynamic Latency-Sensitive Workloads

Applications may present a variety of workload patterns and QoS demands in data centers. For example, non-interactive batches require completion times, whereas transactional web services are concerned with throughput guarantees. Different application workloads require distinct types and amounts of resources. Batch jobs tend to be relatively stable, while latency-sensitive jobs tend to be highly unpredictable and bursty in nature [GTGB14]. Latency-sensitive applications can also include short latency-critical user-facing tasks, such as responding to web requests. Moreover, this type of workload can be characterized by short deadlines of tens of milliseconds [CGD$^+$17].

On the other hand, multi-tenancy services need to efficiently manage resources within and among data centers taking time-varying demands into account [IEM18]. Their workload is not deferrable, meaning that every time a request is received the response must be generated immediately afterward. Consequently, these applications must perform real-time load scheduling, ensuring the quality of the request flow [TQdAB17]. This kind of application presents an unpredictable intensity variation of resource utilization at run time due to the user's different request patterns and periodicity [IEM18]. Therefore, latency-sensitive applications and multi-tenant services are ideal candidates for evaluating the interference effects suffered by dynamic workloads and are considered target applications in this work.

Garg et al. [GTGB14] creates a scheduling mechanism to guarantee that users' QoS requirements are met, according to SLA specifications. They state that it is important to be aware of different types of SLAs and the mix of workloads for better resource provisioning. Results show an improvement, reducing SLA violations. Sampaio et al. [SBP15] address the resource allocation issues running different application workload types (CPU- and network-intensive). After performing experiments with synthetic workloads, their results indicate that their strategy can fulfill contracted SLAs in real-world scenarios while reducing expenses due to energy use.

2.2.1 Benchmark Frameworks

This study aims to analyze interference generated by applications which present dynamic workloads. Therefore, four benchmarks that offer such applications capabilities have been adopted: Bench4Q, LinkBench, TPC-H, and Node-Tiers. Each tool is detailed as follows:

- Bench4Q [ZWWZ11] - This application designs a distributed architecture based on load agents for large-scale load simulation. There can be one load console and a number of load agents at the same time for benchmarking. Each load agent can have specific load settings and be controlled by the load console individually. Bench4Q has features to deduce a controllable and flexible representation of complex session-based workloads and to simulate authentic customer behavior;

- LinkBench [9] - This tool is a benchmark developed to evaluate database performance for workloads, similar to Facebook. It is highly configurable and extensible. LinkBench can be reconfigured to simulate a variety of workloads and plugins can be written for benchmarking additional database systems;

- TPC-H [10] - Also known as a decision support benchmark, TPC-H evaluates the performance of various decision support systems by executing sets of queries using a standard database under controlled conditions. It consists of a suite of business-oriented ad-hoc queries and concurrent data modifications. The queries and the data populating the database were chosen to have broad industry-wide relevance while maintaining a sufficient degree of ease of implementation.

- Node-Tiers [11] - It is a multi-tier benchmark suite that enables fine-grained personalization of resource utilization. Since Node-Tiers was built based on the Stress-ng [12] benchmark, it is able to stress the computer system in various selectable ways. This tool was designed to exercise various physical subsystems of a computer through web requests. This benchmark can be useful to observe performance changes across different operating system releases or types of hardware.

2.3 State-of-the-art Limitations

After investigating interference-aware dynamic resource scheduling aspects, we have noticed that the state-of-the-art presents the following limitations:

- Current works do not analyze the performance interference aspects from applications which have dynamic workload patterns in container-based clusters. Their interference scheduling strategies have not been tested in distributed architectures.

[9]http://github.com/facebookarchive/linkbench
[10]http://www.tpc.org/tpch/
[11]https://github.com/uillianluiz/node-tiers
[12]https://wiki.ubuntu.com/Kernel/Reference/stress-ng

In Hypervisor-based clusters, the interference comes from two main sources: co-existing tasks in the same scheduled virtual instance and co-located instances on the same physical server. This primarily happens in container environments, but the subtle difference relies on the absence of the Hypervisor layer. Shekhar et al. [SAB+18] have adopted container-based technology to analyze interference from different types of applications on a single server, performing vertical elasticity and checkpoint/restore techniques to meet QoS requirements. But they do not use a distributed architecture, such as a cluster, and that is precisely what we are interested in;

- Most of the current related-studies are limited to a simple interference classification method which considers if there is interference among applications. In addition, they only analyze interference generated by some resources: CPU [CSG14, CSGK16, JG17], memory [CSG14, CSGK16, AdAD17] and I/O aspects [JG17]. In [KDSL18], applications are classified with labels: compute or I/O blocks. Ludwig et al.[LXK+19] proposed an interference classification using levels, but they only perform a static classification over the entire application's execution. Within our research, we did not find studies which dynamic classified interference in levels from many sources at runtime;

- The state-of-the-art lacks a time series analysis to deal with interference metrics. Wang et al. [WKNG19] tackle this issue segmenting their jobs into stages and creating a prediction model for each. Since real-time applications can not be divided, this solution cannot be applied to our study. Therefore, we needed to find a solution that detects the best moment to make scheduling decisions;

- Recent studies do not address resource scheduling strategies based on cross-application interference generated only from applications with have dynamic workloads. Shekhar et al. [SAB+18] evaluate interference generated from latency-sensitive and batch-job applications, but do not consider the full range of abrupt changes in workloads of all applications. Therefore, we saw an opportunity for improvement through the implementation of our proposed interference-aware scheduling architecture.

3. INITIAL ANALYSIS OF THE INTERFERENCE PROFILE OF DYNAMIC WORKLOADS

Uncontrolled access to shared resources can cause performance variations that lead applications to fail or run unsteadily. The friction generated by the competition to access RAM, disk storage, cache, or internal busses is known as resource contention, which in this dissertation is also called interference. Many efforts have been made to alleviate it at the operating system level, ranging from better scheduling techniques in multi-core architectures [ZBF10] to dynamically addressing mapping to minimize memory contention [Pot03]. I/O contention, for instance, occurs when multiple tasks compete for a portion of disk bandwidth when the demand is higher than the available resources. The steady growth of virtual data centers has raised a concern about resource contention, and the impact it might cause in environments where performance is crucial and SLA cannot be violated, such as clouds.

In addition, performance interference may also arise due to isolation issues in the virtualization layer, which occurs when a virtual instance exceeds the amount of allocated resources. Because resource limit settings are capacity-driven (e.g. GB, VCores, etc.) rather than throughput-driven (e.g. bandwidth, IPC, etc.), even though a virtual instance receives a limited portion of a resource, there is nonetheless leakage due to uncontrolled access to operating system queues and uncore hardware components. Data center administrators have exaggerated the amount of resources allocated to sidestep contentious scenarios, thus underutilizing data centers.

Due to the systems' current complexity and users' needs, cloud computing has played a pivotal role, delivering resources on-demand on the pay-as-you-go model. However, dynamic workload demands raise additional challenges for cloud computing providers to deliver resources while still satisfying SLAs, and consequently QoS properties [NKG10]. In order to understand interference effects in dynamic workloads in a more appropriately, we first need to see how does each resource fundamentally behaves and is handled by internal devices if workloads vary. Furthermore, we may also need to look at possible problems that could be caused when contention-related issues are applied to all those types of resources. Overall, each of the main resources such as cache, CPU, memory, disk, and network, suffer the most when consolidated applications that may share the same infrastructure [XNR+13].

By starting to answer the RQ1, which intends to find out if dynamic workloads change their interference levels over time, this chapter aims to present how such applications act under different circumstances. First, we present an interference analysis, pointing out how each hardware resource behaves when the workload varies, and then the impact on the interference on response time.

3.1 Analyzing Distinct Hardware Resources Interference

To perform an in-depth analysis of interference generated on different hardware resources from applications with dynamic workloads, Node-Tiers has been adopted. As mentioned before, this tool is a multi-tier benchmark that allows fine-grained personalization of resource utilization. Node-Tiers stresses the computer system in various selectable ways and was designed to stress various physical subsystems of a computer through web requests. This tool explores the concept of web applications (client-server) and allows workload variations to be created. The server-side was performed on a server (presented in chapter 2), whereas the client-side was configured on a different computer. Both pieces of equipment were connected through a Gigabit Ethernet Network. The goal was to stress a server in many ways (distinct resources), through latency-sensitive applications, increasing the request arrival rate, and observing interference effects over the changes in workload behavior. To characterize the interference generated by each application, we used IntP [Xav19] (previously presented). All experiments were performed on a Dell PowerEdge R740xd equipped with: 2x Intel Xeon Gold 5118 Processor, 300GB DDR4 RAM Memory, 1TB Hard Drive, and 4x Gigabit Ethernet Interface. The Ubuntu Server 16.04 LTS (Xenial Xerus) operating system was adopted.

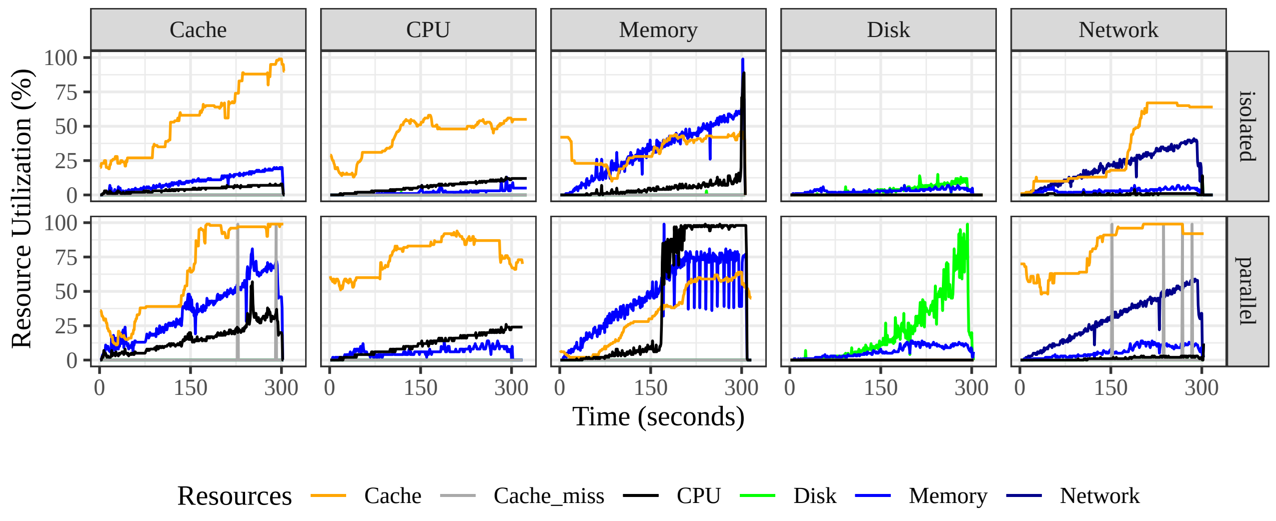

Figure 3.1 – Isolated and Parallel executions from Node-Tiers algorithms that stress when cache, CPU, memory, disk, and network resources are increased.

Firstly, we chose Node-Tiers algorithms that put added stress on a given resource. Then an increasing workload was created, varying from 0 to 300 requests per second, within 300 seconds. Each algorithm was executed two ways: in isolation and two applications' instances co-hosted, labeled here as parallel. Figure 3.1 presents the results of all experiments, depicting how each resource is affected by interference over time. Below, is a discussion and detailed analysis.

3.1.1 Cache

Last Level Cache (LLC) memory is a hardware device created to minimize the performance gap between the processing cores and the main memory [MK19]. When two or more processes are assigned to the same CPU node, threads occasionally share on-chip memory space, and it may lead to resource contention [Xav19]. The chip industry has introduced a new feature in the hardware that allows an OS to determine the usage of cache through applications running on the platform. This is the case of Intel Cache Monitoring Technology (CMT) [HVA$^+$16]. CMT provides mechanisms for an OS to indicate a software-defined ID for each of the threads that are scheduled to run on a core. This ID is called the Resource Monitoring ID (RMID). Since there are associations between threads and RMIDs, they are programmed via a thread-specific model-specific register called MSR, and can be read by system software at any time through an MSR interface.

At the isolated execution, it is possible to observe that the resource that suffers the highest interference is the cache, with increasing but non-linear behavior. Memory and CPU, which pursue a linear trend are the next most affect resources. In parallel execution, cache interference increases significantly, and some peaks in cache-miss occurred at the end of the experiment. Although the interference suffered by the memory produced a linear trend in the isolated test, it doubled its indexes in the parallel execution. The same occurred with the CPU metric.

3.1.2 CPU

When multiple co-hosted applications run and they outstrip the available amount of CPU cycles, it is called CPU contention [Xav19]. In our experiment, even though the target resource stressed was the CPU, the cache suffered the most from the interference. The CPU followed a linear trend, both in isolated and parallel execution. In this case, the CPU, on average, duplicated over time, generating the highest proportionality among all experiments. The memory rates doubled from isolated to parallel tests as well.

3.1.3 Memory

Memory contention occurs when the memory requirements for the active processes exceed the system's available physical memory, causing the it to run out of memory while dramatically decreasing performance. There are two possibilities for an operating system (OS) to overcome this problem: (i) System Paging, when the OS starts to move fractions of active processes to the disk and tries to recover physical memory and reestablish stability; and (ii) System Swapping, when the OS starts to swap an entire process to the disk to reclaim memory, causing tremendous disk overhead [ENBH13]. The IntP's memory module collects counters from the memory controller, which is a digital circuit that manages the flow of data going to and from the main memory. It is usually called an integrated memory controller (IMC). However, the problem was that the LLC_MISS counter did not include prefetch misses. This can be a huge issue when there are many prefetching activities involved (for example, when there is streaming access involved in the program). Recent CPU architectures have available counters that can be fetched from the uncore IMC, allowing for more precise observations.

In the isolated execution, the memory followed a linear tendency up to a given amount of requests. When requests hit this number, the CPU increased considerably, reaching interference rates over 90%. This abrupt CPU growth happened when the request arrival rate was more than 300 requests per second, approximately. Since workloads are co-hosted in parallel execution, the request rate was reached earlier, close to 150 seconds, creating a significant resource usage increment. So, it is possible to note that not only were many resources consumed, also the experiment that should have finished within 300 seconds, ended close to 310 seconds, generating performance degradation.

3.1.4 Disk

Disk throughput can be seen as the most volatile performance metric in a system, because it is architecture-driven and might be affected by external components, such as virtual memory, buses, and I/O controllers [MGXD18]. Although many optimization techniques have been developed, such as page caches for Writeback operations, the performance of block devices has a big impact on overall system performance. When a block request arrives into elevator scheduling queues, the scheduler sorts or merges request queues to get efficient I/O. Thus, requests are merged with others if either request ever grows large enough that they become contiguous. Then, they are sorted, not allowing a read to be moved ahead of a write or vice-versa. These optimization algorithms allow the most contiguous read/write operations to be dispatched to disks, reducing seeks and head movements in hard drives per unit of time. However, the higher the number of requests arriving at the elevator queues, the less efficient the general operation becomes, since the disk handles incoming requests at lower rates than the CPU. This overload therefore increases the queue depth (number of pending requests), which becomes even more noticeable in SMP machines, where multiple tasks contend for a single disk [MLX$^+$20].

In the isolated execution, the increase in the disk was smooth, following a linear trend. In the parallel execution, the interference that co-allocated applications generated followed an exponential trend. Since the arrival request rate doubled every second, it was possible to observe that the interference generated by the block storage contention was significantly amplified.

3.1.5 Network

Network contention appears when processes send messages that travel over the same network interface card (NIC), passing through internal buffers concurrently, thereby increasing job communication time and degrading performance [Sav19]. With the advances in CPU architectures and operating system structures, network performance has also improved in modern operating systems by changing packet receipt from interrupt-driven to polling mode. Previously, the network cards would typically fire a hardware interrupt whenever a packet arrived, suspending the executing software, affecting application performance. Current operating systems have changed the way that network packets are handled once they are pulled off the wire. They implement a polling mechanism which is periodically interrupted. While the poll method is executing, receive interrupts for the network device are disabled. Thus, the operating system can potentially drain multiple packets from the network device receive buffer, increasing throughput, and decreasing latency at the same time as reducing the interrupt overhead. In Linux operating systems, packet processing begins when the interrupt handling process (*ksoftirqd*) determines that a *softirq* is pending. It calls the *net_rx_action* driver-specific method, which begins processing all packets available in the network device ring-buffer before cpu-time is up (limited to 2 jiffies). The processing ends when the data is copied to the application-specific socket buffer. Yet, applications still suffer from throughput issues due to back-pressure caused by cross-application tasks, making either the interrupt handling mechanism unable to drain packets from the network device fast enough or the application unable to remove queued packets from the socket buffer fast enough [Xav19].

Moreover, network back-pressure grows linearly in isolated and parallel execution. When comparing the two, it is possible to note that even if the load doubles in a parallel execution, network interference does not double proportionally. Furthermore, in parallel execution, the use of cache is intensified, generating some cache-miss incidents and revealing a strong relationship between network and cache resources.

3.1.6 General Findings

In general, we can conclude that each application has a specific resource usage behavior, creating different interference rates. Applications can generate interference indexes according to the workload variation, proportional to the load changes, which are the cache and CPU examples. On the other hand, the opposite also happens, when the interference generated does not follow the rising workload trend, in the cases of memory, disk, and network. Moreover, we found that these results could change if they were executed over different hardware, because they are strongly dependent on their characteristics or capabilities.

3.2 Interference Impact on Response Time

Left unmanaged, the competition for machine resources can lead to severe response-time degradation and unmet SLAs [IAK+18]. Latency-sensitivity does not only imply literal real-time applications, but also applications that have flexible limits for response times. For instance, users expect a web search to complete within a specific amount of time[SAB+18]. Hence, delays in latency-sensitive applications runtime do not always represent their performance degradation properly or demonstrate SLA violations. Therefore, to accurately evaluate such applications' deterioration, we analyzed the response time degradation through a set of experiments. With Node-Tiers, we combined all algorithms, mentioned in the previous section, in parallel executions and captured their response times. In other words, the application that uses more the cache was combined with the one that uses more the CPU, memory, disk, and network. This experiment was performed with all algorithms, combining them all. All executions took the same workload and time interval: 0 to 300 requests per second within 300 seconds. Figure 3.2 depicts all the response times collected from all tests. Note that this image combines two applications per frame, indicated in the labels on the right side (black) and on the top (gray).

It is clear that each application has its specific response time increment and none of them have a linear growth trend. For example, in the CPU-network execution, the Network application starts to increase its response time before the CPU. In the CPU-memory case, memory practically does not change its response time behavior whereas CPU does.

Furthermore, all executions with the same application, on the diagonal line (i.e. cache-cache, CPU-CPU, and so on), present the same behavior and start to increase their

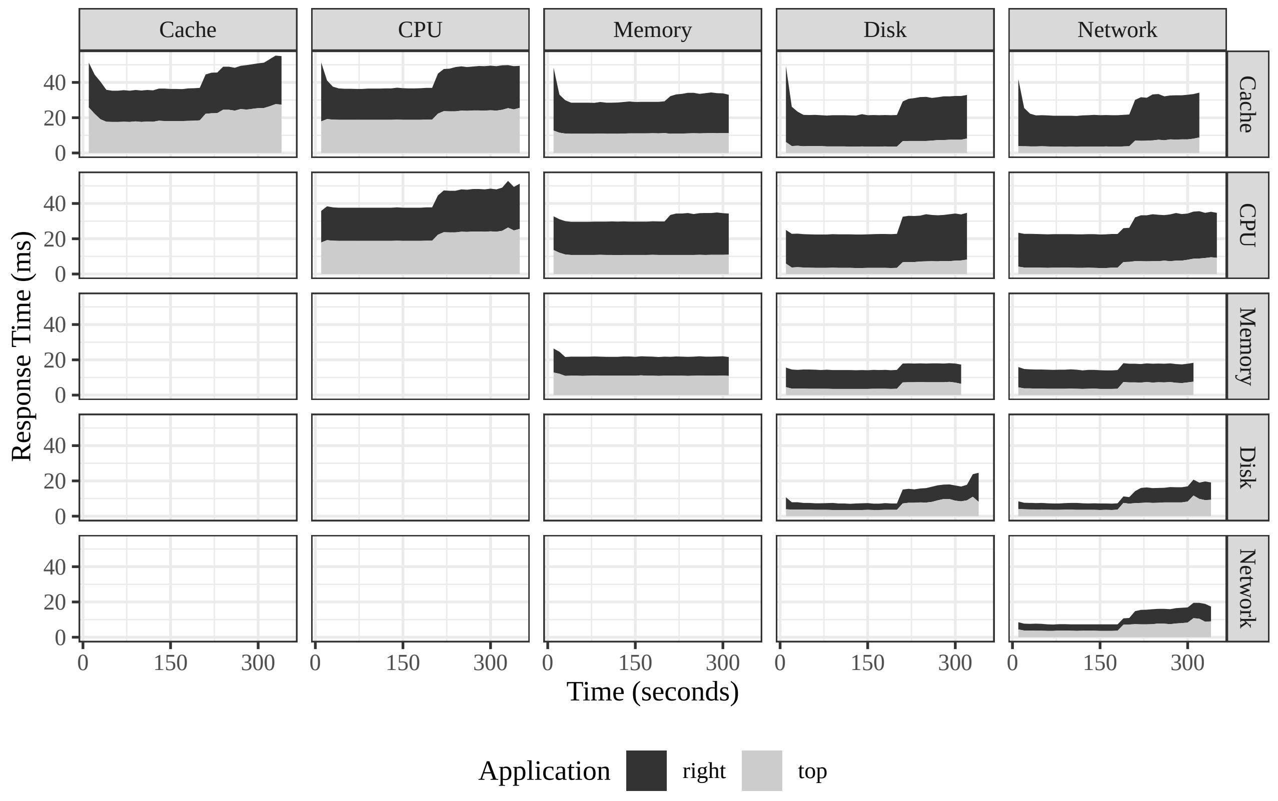

Figure 3.2 – Response time collected combining cache, CPU, memory, disk, and network algorithms from Node-Tiers. Each frame presents the combination of two applications, indicated in the labels on the right side (black color) and at the top (grey color).

response times almost at the exact same time. This is because the same resource is being used, causing approximately the same response time degradation.

It is worth mentioning that although each application presents specific characteristics and different behavior, the interference generated between the two when co-hosted, tends to proportionally affect their response times, on average. This means that response time degradation is proportional to each resource.

4. INTERFERENCE CLASSIFICATION'S IMPACT ON RESOURCE SCHEDULING

Resource scheduling can be defined as the ability of cloud infrastructures to dynamically change the amount of resources allocated to a running application. Hardware resources should be allocated according to the changing workload, enabling resource management to preserve the quality of service requirements at reduced costs [AETEK13]. The previous chapter stated that workload variations can affect the behavior of applications differently, not only in resource usage but also response time. Each application's execution might have a different hardware subsystem behavior, strongly dependent on workload variability. Therefore, an interference classification system that perceives the changes of application behavior over time becomes essential to perform interference-aware scheduling strategies in cloud computing environments.

The most challenging aspect of this problem is to find a classification system that can accurately determine how cross-application interference affects each specific resource when they are being shared over time [ZBF10].

In section 2.1.2, we discussed how interference classification has been addressed recently. In that study phase, we ran some tests with the classification method from [LXK+19], considered here as the state-of-the-art work in this field. However, we noticed that the authors' technique presented some limitations when applied to a dynamic scheduling scenario. Their solution creates only one interference label per resource over the entire application's execution, producing a static classification. Yet, as discussed in chapter 3, interference levels can change over time with dynamic workloads and static classification outcomes might be inaccurate and lead to less than ideal resource scheduling. Therefore, in this chapter, we introduce a dynamic classification system, segmenting the workload to verify if it possible to improve the efficiency of hardware resources. Moreover, this chapter presents a preliminary evaluation of how different interference classification techniques can impact interference overhead. We first discuss the influence of workload variation on the classification phase. Then, we evaluate their efficacy in reducing the overhead and improving resource utilization in these scenarios.

4.1 Transitioning from a Static to a Dynamic Classification Schema

To evaluate alternative scheduling policies, Ludwig et al. [LXK+19] created a classification method that explores levels of interference. It analyzes interference in machine resources (CPU, memory, disk, network, and cache) over the entire application's execution. The authors' technique categorizes resources with their respective interference level labels, according to Table 4.1.

Table 4.1 – Interference intervals and their respective level labels, introduced by Ludwig et al. [LXK+19].

Interval	Label
0%	Absent
1% - 20%	Low
21% - 50%	Moderate
51% - 100%	High

For example, we ran a benchmark developed to evaluate database performance for workloads, similar to those of Facebook's production, named LinkBench. First, we created an increasing workload, starting with a low load and gradually going up to a high load, and profiled it with IntP. In the LinkBench benchmark, it is possible to set the number of requests (operations) and the number of requesters (threads). The number of requests was configured into 1,000 (fixed) and the number of requests varied from 10 to 50 (by 10 to 10). The entire execution was profiled with IntP. Figure 4.1 shows interference suffered by each resource in this experiment. The top chart presents the static classification method proposed by [LXK+19], analyzing the interference levels over the application's entire life process, assigning just one label per profiled resource based on mean values. In this work, we refer to this classification format as Unique. To evaluate how well this technique deals with workload variations and its impact on the classification, we redid the same experiment segmenting the trace in four parts and applying the same static classification technique to each part. Results are shown at the bottom chart of the same figure and we refer to it as Segmented.

The top chart (Unique) shows that only disk and cache suffer low rates of interference. By classifying the application in four parts, in the bottom plot (Segmented), it is possible to notice that the overall interference generated in each resource remains the same. This means that the interference metrics do not have a significant change to modify their labels. Concluding that: (i) there are cases where interference levels do not change, even when their workload does; and (ii) in these cases, the Ludwig et al. [LXK+19] classification method works well since there is no representative variation in the interference metrics.

While there are applications with no expressive variations in the interference metrics over a given period, this is not always the case. We thereofre did the same experiment with a QoS-oriented e-commerce benchmark called Bench4Q [ZWWZ11]. Again, an increasing workload was created. Bench4Q emulates active e-commerce users through entities called Emulated Browsers (EBs). Thus, the load started with 10 simultaneous EBs and every minute we added 10 more, ending the execution with 120 EBs (720 seconds).

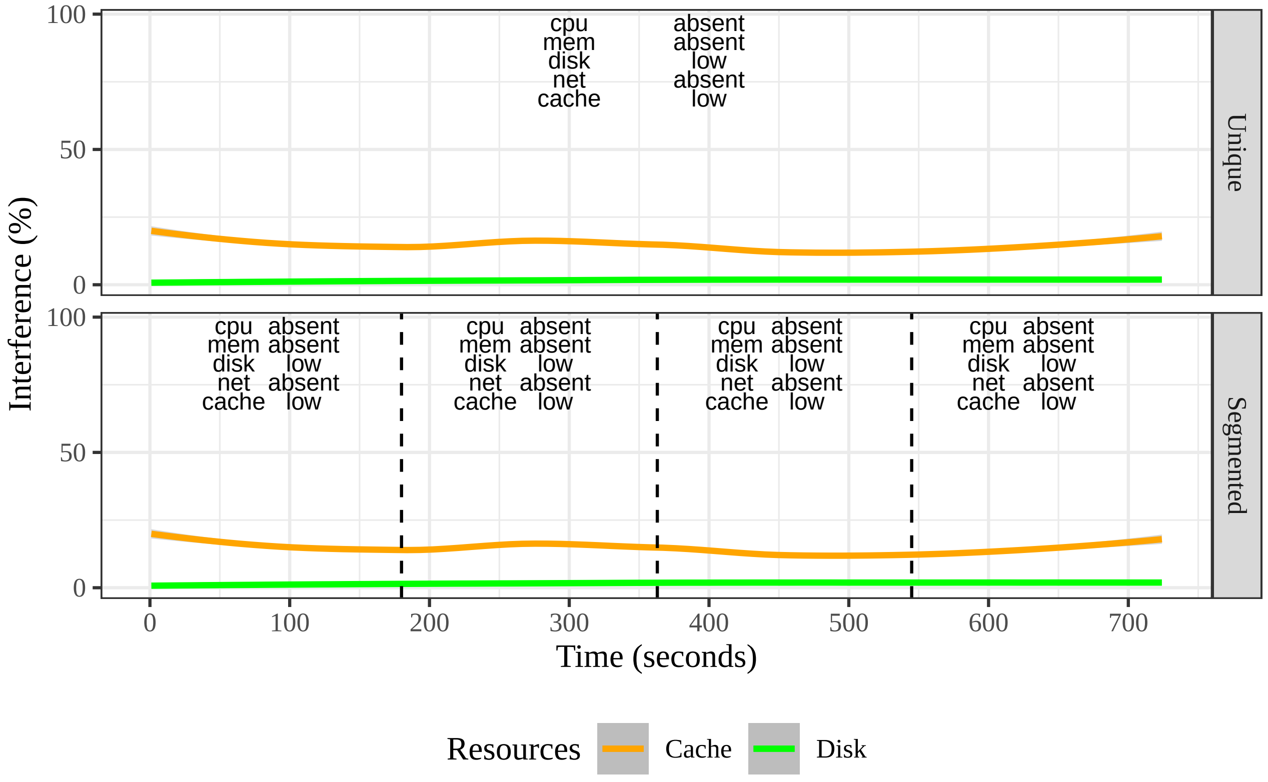

Figure 4.1 – Unique (top) and Segmented (bottom) LinkBench static interference classification. To facilitate the visualization, a Loess function was applied to smooth short-term variations in each resource. IntP metrics that do not suffer any interference in these experiments were not depicted.

Also, a single classification was performed after the entire application was executed. Then the execution was also split into four parts, and each part was classified again. Results are presented in Figure 4.2.

One can see that some resources do not change their labels, for instance, memory, cache, and network. Because their interference metrics remain at the same level, with no expressive variation on average, their labels are maintained. On the other hand, some resources do change their labels. Namely, CPU and disk. Disk's behavior smoothly decreased, from a low to an absent label, halfway through execution. In addition, the CPU had the biggest behavior change, starting low, then going to moderate levels, and ending with a high interference level.

In conclusion, LinkBench execution did not generate a significant variation in terms of interference effects, whereas Bench4Q did. Therefore, due to their dynamic workload nature, each application should be handled differently. Thus answering RQ1, confirming that applications with dynamic workloads may change their interference levels over time.

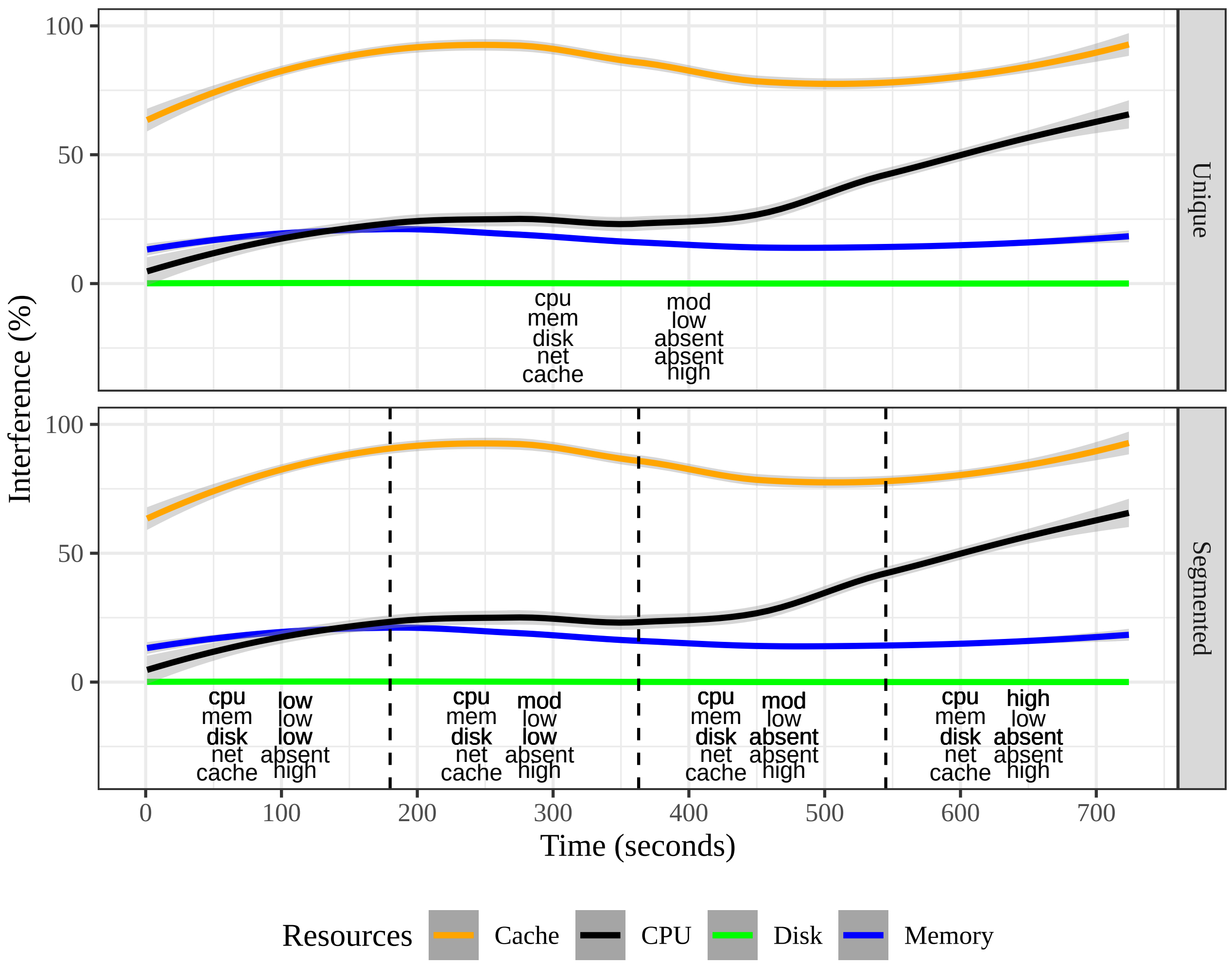

Figure 4.2 – Unique (top) and Segmented (bottom) Bench4Q static interference classification. To facilitate the visualization, a Loess function was applied to smooth short-term variations for each resource. Resources labels that changed are shown in bold in the bottom plot. IntP metrics that did not suffer any interference in these experiments were not shown.

4.1.1 Impact on Interference Overhead

Aiming to evaluate the impact of cross-application interference on the final scheduling over time, by profiling the total interference generated by running an application profiling, a tool called CIAPA [1] was used. This is a scheduling analysis tool, originally proposed by [LXK+19], that uses an interference overhead function, represented by interference set I'. The interference level for each resource is denoted as follows:

$$g(I'_{res}) = \{I \mid I \in I'_{res}, I > 1\} \tag{4.1}$$

[1] https://uillianluiz.github.io/ciapa

Where $res = \{CPU, memory, disk, cache, network\}$. The function g, denoted in Equation 4.1, returns a set of values that are greater than 1. All resource interference metrics are measured and allocated into an interval. Depending on the interval in which they are set, the interference overhead (I') index value varies according to Table 4.2.

Table 4.2 – Performance degradation generated by resource interference, introduced by Ludwig et al. [LXK+19].

Level	CPU	Memory	Disk	Network	Cache
Absent	1.00	1.00	1.00	1.00	1.00
Low	1.03	1.07	1.12	1.05	1.07
Moderate	1.15	1.62	1.82	1.32	1.18
High	1.33	1.74	2.25	1.57	1.26

CIAPA tries to minimize the total interference overhead by testing all possible combinations of applications per host. Therefore, the result is finally given by the multiplication of the cost of each resource, which is calculated by using the function in Equation 4.2.

$$f_i(I') = f_{i'}(I'_{cpu}) * f_{i'}(I'_{mem}) * f_{i'}(I'_{disk}) * f_{i'}(I'_{cache}) * f_{i'}(I'_{net}) \tag{4.2}$$

To execute applications with dynamic workloads, we have chosen three different applications that can execute workload variation. The first and second were mentioned before, Bench4Q and LinkBench. The third is a decision support benchmark called TPC-H.

Different workload variations can change applications' behavior in terms of resource usage and performance. In order to evaluate their behavior, four workload patterns were set for each application. We created the following workloads: *Increasing*, *Periodic*, *Decreasing* and *Constant*. This was inspired by Iqbal et al.'s [IEM18] study, where *Increasing* starts with a low load and gradually goes to a high load. *Periodic* has continuously high-to-low and low-to-high load variations. *Decreasing* is the opposite of *Increasing*, it starts with a high load and gradually goes down to a low load. Finally, *Constant* always maintains the same workload.

To evaluate interference variations, each application was submitted to the 4 workload patterns, resulting in 12 different examples. These examples have been tested over different numbers of hosts, as follows: 4, 6, 8, 10, and 12. Then, we applied two classification formats on each workload: (i) Unique and (ii) Segmented. Unique representing a single classification over the entire application's execution, while Segmented divides each execution into four parts and runs the classification on each. Because the application was divided into four parts for the Segmented format, each part produced a set of interference level labels. Therefore, Unique format results were multiplied by for to keep both proportionals.

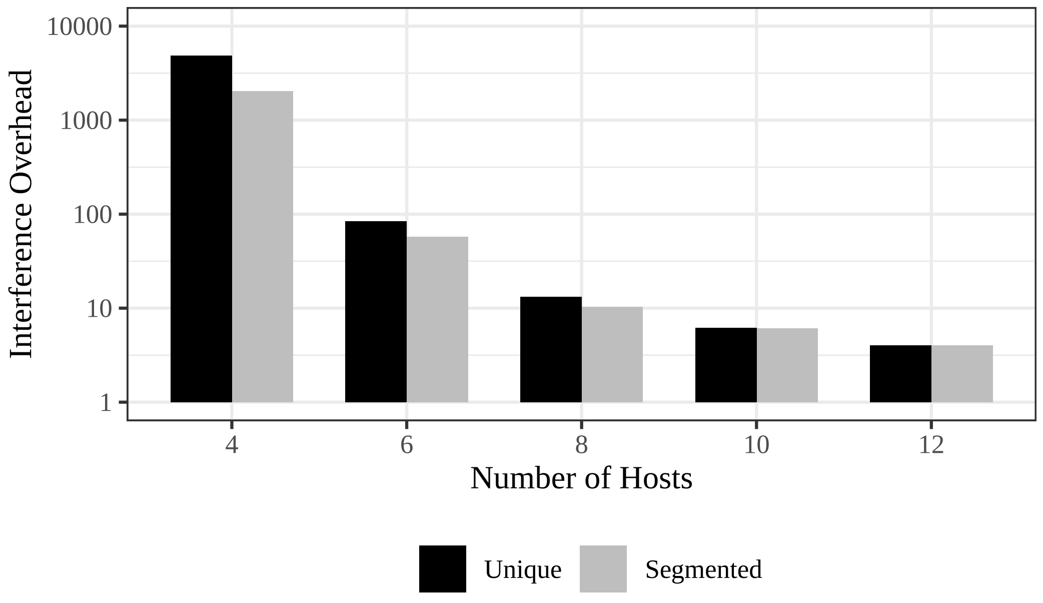

Figure 4.3 – Comparison of interference overhead.

Classification outcomes were inserted into CIAPA and the results are presented in Figure 4.3. As previously mentioned, CIAPA uses an interference overhead function that represents the total interference generated in the system. The lower the result, the better the hardware is being utilized. Thus, lower indexes present better hardware efficiency, reducing response time, makespan, etc.

Overall, the largest difference occurred with the smallest number of hosts. Namely the cases with more cross-application interference, resulted in greater performance degradation among co-located applications. As the number of hosts rose, the difference between interference overhead incidence decreased. Therefore, resource concurrency among co-hosted applications tended to decrease as well. Only with the largest number of hosts, both classification methods achieved the same values. This occured because each host only ran one application instance. Since there is no incidence of cross-application interference, the result has been left minimum.

It is interesting to note that in all experiments performed in Figure 4.3, the Segmented format reached lower interference overhead indexes than the Unique. The results demonstrate that the Segmented format improved the hardware utilization efficiency by 22%, on average, reducing resource consumption and also performance degradation at the application level. This highlights that a workload-aware fine-grained classification can reduce interference overhead while only one can lead to less efficient scheduling decisions. In addition to evaluating cross-application interference at time intervals tends to improve the performance of applications while preserving the quality of service requirements.

From this point on, *Classification Method* in this study refers to the method of how each resource receives its interference level within a given execution period, while *Clas-*

sification Format indicates how many time slices the Classification Method is applied in: Unique format for a single interference classification over the entire application execution, and Segmented for multiple classifications.

4.2 ML-driven Interference-aware Application Classifier

In the previous section, we performed experiments under high workload variations running a static interference classification only once (Unique) and several times along the execution (Segmented). Comparison results showed that the Segmented format achieved better resource utilization efficiency than the Unique. However, the thresholds of the interference levels (Table 4.1) of the applied static classification method ([LXK$^+$19]) were empirically defined. Although this still resulted in better overall results in the applications' scheduling, we observed that applying the method in applications with high workload variations could lead to an unrepresentative classification estimate. When dealing with a dynamic scenario, applications' workload could have unpredictable patterns, possibly presenting abrupt changes over time, making it necessary to run a real-time classification process. In addition, the thresholds used in the classification strategy were empirically defined, and therefore the user needed to have previous knowledge of the application's execution to perform an accurate classification.

To tackle this issue and beginning to answer RQ3, which aims at exploring what architectural changes are needed to move from a static to a dynamic scheduling, we created an interference-aware application classifier based on the combination of two well-known machine learning techniques: Support Vector Machines (SVM) and K-Means. After it is trained, the proposed classifier receives monitored metrics from applications and dynamically defines the interference level thresholds for each resource. This application classifier was first introduced in [MKdD20] with its static variant. It was then improved by introducing a dynamic version [MKDD21b], where we applied the classifier several times during the execution of an application to better react to workload variations and possible changes in interference levels.

This section outlines the both how the classifier works and evaluates its performance through different experiments. First, we explain the classifier overall, including its dependencies and capabilities. Then, we present an evaluation of the dataset and model validation. Lastly, we compare its efficiency with related work.

4.2.1 Classifier Design

Two machine learning algorithms work together to implement the proposed classifier: SVM for classification and K-Means for clustering. Initially, the SVM receives interference metrics from the target application collected each second by IntP, and those metrics are classified and stored into resource queues for their respective classes: memory, CPU, disk, network, and cache. Subsequently, K-Means quantifies values for each queue and returns their interference level for a specific period. Both machine learning algorithms use a previously defined training dataset to assist them in decision making. Figure 4.4 illustrates an overview of how the classifier works. More details about the classification method are presented in the next subsections.

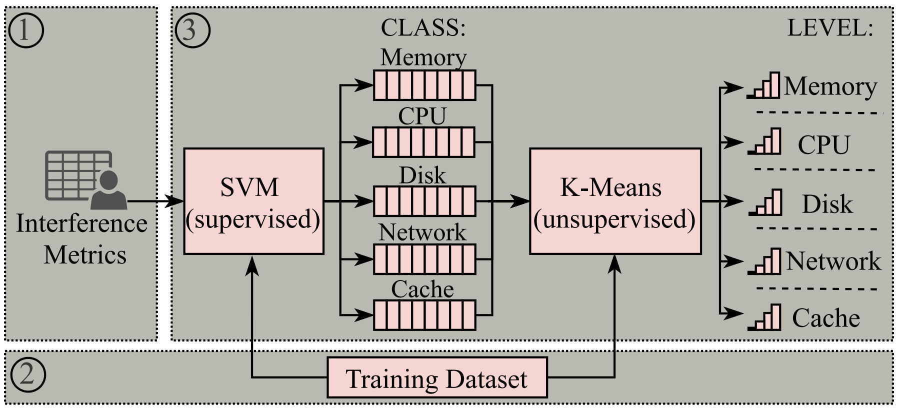

Figure 4.4 – Classifier architecture overview: Component 1 represents the collecting of interference metrics; Component 2 depicts the training dataset assisting machine learning algorithms; Component 3 illustrates the classification process and its outcomes [MKdD20].

Interference Profiling

To characterize the interference generated by each application we used IntP. The general idea is to profile applications at runtime, returning the interference for each resource used during execution: memory (mbw), CPU (cpu), disk (blk), network (netp and nets), and cache (llcocc and llcmr). IntP returns these metrics every second, based on the workload variability (higher values correspond to higher interference overhead).

Training Dataset

To take advantage of selected machine learning techniques, it is essential to have as much data as possible to use as input in order to train the models. However, no available datasets were found in the literature with cross-application interference traces. Therefore, Node-Tiers was used. This tool is able to stress the computer system in many ways, through web requests and we could generate a diversified interference dataset by performing various algorithms.

To maintain a data history from each interference class, we had to stress the main resource classes and store their interference metrics. For example, to collect CPU interference metrics, Node-Tiers was set with the *cpu* parameter, which means only the CPU was stressed. To collect the cache class, the *cache* parameter was set, and so on. We produced five major interference classes: *memory*, *CPU*, *disk*, *network* and *cache*. More precisely, 10,000 samples were collected from each interference class, resulting in a dataset with 50,000 samples.

Classification Process

The proposed ML-based interference classifier dynamically defines thresholds and assigns interference levels for each resource used by the monitored applications for a particular time period, without the need for user intervention.

The target application is monitored with the IntP tool, every second, generating data from 7 resource metrics. This data is passed to SVM as input data to be classified. SVM is a supervised technique, so it uses labeled data from a training dataset to label the new data. SVM takes these tuple 7 interference values and categorizes them into one of the 5 resource classes: memory, CPU, disk, network, or cache. The idea here is to select the class that best represents the interference generated by this application in this second and store the interference value(s) in its respective queue. Since this is done at runtime for all applications that are being executed, its a way to reduce overhead, selecting the most representative values for each class. After a system has defined the time interval, the values stored in these resource queues become K-Means input data, so that the corresponding interference level for each resource is assigned. This two-step classification process is repeated until the end of the execution, characterizing the dynamism of our approach. In fact, interference levels are reevaluated regularly so that we are able to better react to workload changes.

Inspired by [LXK$^+$19], we used four possible levels: absent, low, moderate, and high. However in our work, these thresholds were not empirically predefined, instead they varied for each different resource and were automatically defined in the K-Means training phase. When there was no interference incidence from a class (no data in the respective queue for the period), the classifier interpreted it as Absent.

K-Means, previously trained, determins the interference levels of each resource class. K represents class divisions: low, moderate, and high. Its value was set to 3 (k=3). At the beginning of the classification process, SVM trains its model and K-Means finds its centroids. Since both techniques are supported by an already created dataset, the SVM model and K-Means centroids are the same until they are retrained.

For example, we may have an IntP tuple for a running application (i.e. [0%, 0%, 3%, 15%, 5%, 10%, 80%]) as input in a particular second classified by SVM as “CPU”. This output is buffered in the respective class queue, in this case CPU. This classification phase is repeated for the duration of a time interval, and then these class queues become K-Means input to determine an application interference level for each class within the monitored interval. These queues are only used to buffer the SVM output for each class since SVM runs each second and K-Means runs for each time interval. The goal here is to reduce overhead since this is done at runtime.

The classifier was implemented using the free statistical software tool R [2]. To execute SVM and K-Means algorithms we chose *e1071* [MDH+19] and *stats* [R C19] R packages, respectively. Basically, SVMs can only solve binary classification problems. To allow for multi-class classification, `libsvm` performs the “one-against-one” technique by fitting all binary subclassifiers and finding the correct class through a voting mechanism [MDH+19]. Even though we chose R in this work, the model design is not limited to this specific tool, other software or libraries, such as Keras [3] for Python or Weka [4] for Java, could also potentially be used. One factor in choosing (or dismissing) a machine learning platform is its coverage of existing algorithms [LKRH15]. R provides flexibility for implementing several types of model architectures.

Many machine learning techniques are available in the literature, and there are different sets of parameters to be configured depending on which is chosen. In order to: (i) not over- or under-fitting the training model; (ii) to eliminate the user responsibility of setting these parameters; and (iii) to find the best set of parameters for machine learning techniques, *caret* [5] package were used. This package provides a standard syntax to execute a variety of machine learning methods, thus simplifying the process of systematically comparing different algorithms and approaches.

Since our application classifier analyzes workload behavior through interference metrics, this approach can be applied with other interference metrics and monitoring tools (like PAPI [6]). All files, including source codes and results, are available at GitHub [7] and Code Ocean [MKdD21a].

[2]https://www.r-project.org/
[3]https://keras.io/
[4]https://www.cs.waikato.ac.nz/ml/weka/index.html
[5]https://cran.r-project.org/web/packages/caret/index.html
[6]https://icl.utk.edu/papi/
[7]https://github.com/ViniciusMeyer/interference-classifier

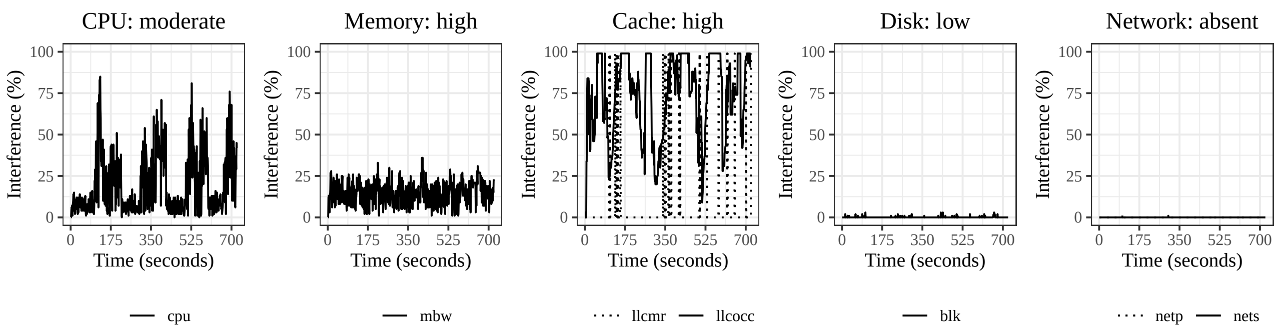

Figure 4.5 – Bench4Q execution under an oscillating workload.

To better understand the classifier functionality, Figure 4.5 presents the execution of Bench4Q under an oscillating workload. Each interference class is shown separately, together with its classification level for the elapsed time. Overall, CPU and cache are more stressed than memory, disk, and network. It is worth noting that disk and network resources had low interference rates, on average less than 2% and 5%, respectively. In this case, network classification is categorized as absent while the disk was labeled as low. Although memory values didn't seem particularly elevated (less than 25% on average), our classifier labeled this resource class as high, based on the K-Means threshold setting.

4.2.2 Classifier Evaluation

In this section, we analyze the training dataset and validate the proposed ML model presenting its quality metrics.

Dataset Analysis

In reference to data analysis, machine learning techniques process the dataset and produce a set of descriptive statistics from the analyzed data. In addition to a handful of metrics, these techniques support statistics on configurable data slices and cross-feature statistics such as the correlation between them. The correlation between a component and a variable estimates the information they share. The variables can be plotted as points in the component space using their correlation as coordinates [AW10].

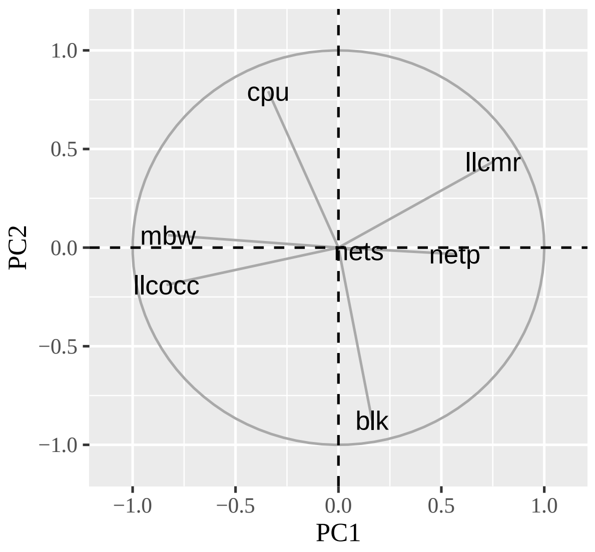

Figure 4.6 – Correlations circle of dataset interference classes.

Figure 4.6 presents the Circle of Correlations. This image shows the correlations between all of the interference metrics that were collected to develop the training dataset and the principal components (PC) are shown as coordinates. The relationships between all variables can be interpreted as follows:

- Different assets moving in the same direction are positively correlated; if they move exactly together, they are perfectly positively correlated;
- Uncorrelated returns have no relationship to each other and have a correlation coefficient of close to zero; so, they are orthogonal to each other;
- Negatively correlated returns move in opposite directions (quadrants). Series that move in exactly opposite directions are perfectly negatively correlated.

By looking at this image, observing all features, we can get important insights into the shape of the dataset. It is worth noting that there is a strong positive correlation between some interference classes, such as *llcocc* and *mbw*. On the other hand, there are those that present negative correlations, such as *cpu* and *blk*. This means that: (i) while the cache is used, memory bandwidth is used as well; and (ii) while CPU is consumed, the disk is practically not required. This information is essential for the clustering phase (K-Means) since it uses these findings to train and find the optimal arrangement of centroids (interference interval levels). Moreover, information comes from the training dataset, and if it is changes, the correlation between resources will probably behave differently (directions), strongly depending on the data.

Model Validation

Usually when a model is created with the support of machine learning techniques, a validation step is performed to find out if it has a good quality rate. A validated model is safe when a simple premise is reached: the quality measures have to achieve reasonable rates. Therefore, we chose two classification quality measures [FHOM09]: (i) Accuracy is the most common measure to evaluate a classification process. It is defined as the degree of the model's correct predictions (or conversely, the percentage of miss-classification errors); and (ii) F1-Score (or F-Measure), that correlates Precision and Recall metrics. The SVM algorithm was evaluated by repeating a 5-fold stratified cross-validation 10 times, with different randomly selected partitions. It gave this model with the highest possible validation score.

For clustering, we defined Rand Index [MA19] (or Rand Measure) as the quality measure. Rand Index is a measure of the similarity between two sets of clustered data. It has become the index of choice in comparing the agreement between two separate partitions of the same data set. It adjusts for chance agreement and is not restricted to comparing partitions with the same number of segments. Complete independence between the two divisions yields a Rand Index of essentially zero. Complete association yields an index of 1.0. From a mathematical standpoint, this index is related to accuracy, but is also applicable even when class labels are not used.

All quality measures range between 0 and 1. The higher the measured value, the better the quality. All metrics mentioned above are presented in Table 4.3.

Table 4.3 – Quality measures of machine learning techniques. (- not applicable)

Measure	SVM	K-Means
Accuracy	0.97	-
F1-Score	0.98	-
Rand Index	-	0.82

In all experiments, the quality metrics presented acceptable rates. This means that both machine learning techniques used in the classifier prompt good quality training.

4.2.3 Comparison with State-of-the-Art

To evaluate the proposed dynamic ML classifier, we compared it to two static classification approaches (**Ludwig et al**), using customized classification intervals and a variation that uses proportional intervals (**Proportional**) to verify our claim that we would perform better with applications that have dynamic workloads. We also compared it to the state of the art in round-robin scheduling as a baseline (**Even**). More details about these techniques are presented below:

- **Even** implements the *EvenScheduler* algorithm, which is the Apache Storm [8] default scheduler. This algorithm distributes computation tasks across nodes in a Round-Robin manner [ASZ20]. When tasks are scheduled, this approach counts all available slots on each node and places application instances to be scheduled one at a time to each node while keeping the order of nodes constant. We have decided to use this method because Apache Storm is a well-known framework that processes real-time data, such as cloud multi-tenant systems, which are the target applications in this work. Furthermore, we chose *Even* as a baseline since it is the less optimized approach.

- **Ludwig et al.** [LXK+19] evaluate the profile of application workloads and use interference classification levels. This approach was introduced with details in subsection 4.1 and was chosen because it is the work most closely related to ours. The difference between this work and ours lies in the fact that the classification is static, done only one time in the beginning and the definition of interference levels thresholds are fixed and empirically defined.

- **Proportional** is similar to [LXK+19] but categorizes the interference from each profiled resource through a proportional division of the interference levels ranges (1/3 for each level, low [0%-33%], medium [34%-66%], high [67%-100%]). This technique was chosen because defining fixed thresholds is commonly adopted in the resource management field [ZLT+16, KMX+20].

One of the main challenges when classifying the interference levels of an application is to define thresholds at each level for a specific resource (for example, is 30% CPU interference low or moderate? And 60% memory interference, moderate or high?). *Even* uses an “in order” scheduling strategy, and therefore does not take interference classification aspects into account. Thus, it does not need to define interference levels. *Ludwig et al.* (A) and *Proportional* (B) are similar approaches, interference classification based on fixed thresholds. In our approach, we use variable thresholds automatically defined for each resource using ML. This is one of the main contributions of this work. To better visualize these differences, Figure 4.7 shows how the definition of interference levels for each technique. It is worth mentioning that the two broken lines (close to the points 30 and 70, on the x-axis) in our classification approach (C), are for illustrative purposes only, and the arrows indicate that they can vary depending on the workload.

[8]https://storm.apache.org/

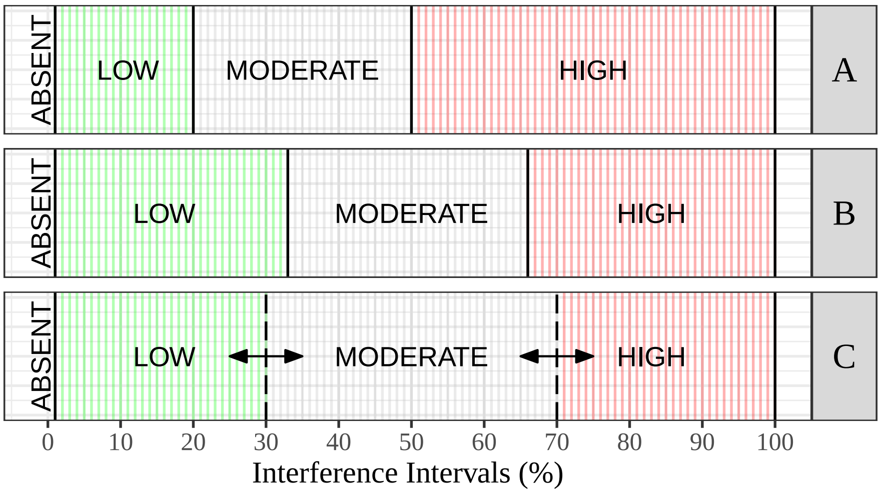

Figure 4.7 – Interference levels used in each classification method and their respective intervals. Ludwig et al. (A) and Proportional (B) use static thresholds for determining their interference classes while in our classifier (C), they are variable, automatically defined without user intervention.

To perform the comparison, we took the same workloads, with the same patterns (increasing, periodic, decreasing, and constant) and adopted the same applications (Bench4Q, LinkBench, and TPC-H).

As we saw in the experiments in section 4.1, employing the Segmented format to classify workload interference levels can reduce interference overhead, since a classification scheme that better represents workload variations tends to use resources more efficiently. Therefore, in this experiment, we expanded this approach into dynamic classification methods that reclassify the workload after regular time intervals. More specifically, the monitored period was set to 180 seconds, that is a common interval found in related work that dynamically reevaluates classifications at runtime, such as [ZT12]. Classification outcomes were inserted into CIAPA over different numbers of hosts, as follows: 4, 6, 8, 10, and 12. The results are presented in Figure 4.8.

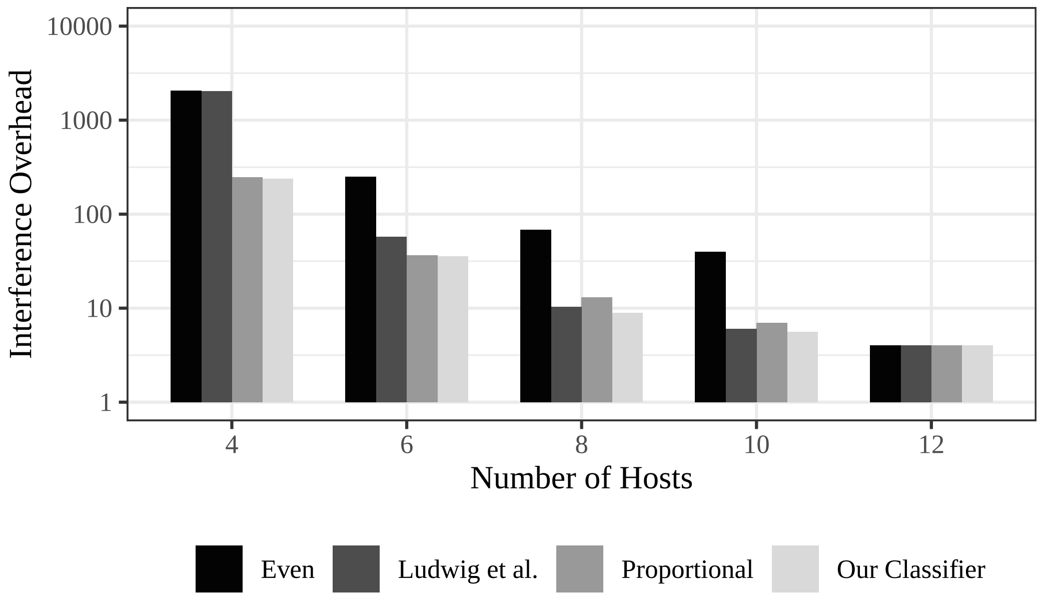

Figure 4.8 – Comparison of interference overhead with state-of-the-art.

In this figure, it is possible to observe that, the *Even* method presented the worst results (higher indexes) in all executions, which was already expected since this method ignores interference among applications. In general, our solution demonstrated the best placement results, presenting a 27% improvement of resource utilization efficiency, on average, compared to the other strategies from related work. The only exception appears with 12 hosts. In this case, each host handled only one application, producing no interference rate and generating the lowest possible interference overhead. As the number of hosts decreased, interference overhead indexes became higher. Therefore, the resource concurrency among co-hosted applications tended to increase as well. With 4 hosts, the highest indexes occurred, revealing the case with more cross-application interference incidences and greater performance degradation.

Preliminary results with different workloads, have confirmed that resource interference has a high impact on application performance, which was already demonstrated by related work. All these conclusions guide us to answer RQ2, confirming that a dynamic interference classification system, which better represents the variability of workloads over time, indeed lead to better resource scheduling decisions.

5. DYNAMIC INTERFERENCE-AWARE SCHEDULING ARCHITECTURE

Performance interference is known to adversely impact applications' QoS properties and dynamic service demands. In fact, workload profiles create even more challenges for cloud service providers when it comes to managing resources on-demand to satisfy SLAs while simultaneously minimizing operational costs [NKG10]. Therefore, any solution that addresses these challenges must account for the workload variability and performance interference [SAB+18]. Due to the dynamic nature of the process, some questions arise. Specifically: How to classify applications in real-time based on the interference they generate? When to execute classification? When to schedule? And how to tradeoff migration costs?

Finding a solution that comprehensively covers all the issues mentioned is not a straightforward task. Recently approaches have been proposed that present significant improvements concerning interference classification and dynamic scheduling strategies. However, there are still gaps in the state-of-the-art. One example is the lack of a complete scheduling architecture that automatically handles dynamic workloads, finding the best intervals to classify and schedule applications among cluster nodes. Furthermore, the architecture should be able to address performance interference aspects without earlier workload know-how and also without user mediation.

Based on the concept that dynamic scheduling algorithms based on interference, which analyzes workload variations over time, could further improve resource utilization, and consequently reduce SLA violations, in this chapter we propose IADA. It is a full-fledged interference-aware scheduling architecture for dynamic workloads in clouds. This architecture aims to efficiently schedule applications based on the interference they generate, without user intervention and with no knowledge of the previous workload.

In the next sections, we introduce the proposed architecture, describing its capabilities and tools in detail.

5.1 A Novel Interference-aware Architecture Design

In general, interference-aware task schedulers are created by combining three main steps [CH11, ZT12, BRX13, ZRWZ14, Xav19, WKNG19]: (i) profiling queued tasks based on their resource needs; (ii) predicting the performance interference; and (iii) scheduling the task on the best-suited node, which is the node that causes the lowest performance interference effects. Since we are interested in scheduling real-time applications based on workload variability, we decided to use a reactive approach. Thus, we adjusted the prediction step by splitting in two: (ii-A) classifying interference and (ii-B) analyzing the best time intervals from applications at runtime in order to perform the scheduling (next) step.

Hence, to build a dynamic interference-aware scheduling architecture, we used four main components, presented as follows: (i) a profiler that reads hardware metrics; (ii) a technique to determine significant workload changes, based on profiling data at runtime; (iii) an interference classification method supported by a combination of machine learning techniques; and (iv) a scheduling algorithm that interprets all data generated by previous components and makes efficient placement decisions.

To perform the proposed architecture, all these aforementioned components were assigned in a node which works over the entire computational environment analyzing and executing scheduling decisions, referred to here as Node Manager. Also, an IntP profiler module is executed inside each cluster node, profiling all applications and sending all data to the Node Manager. First, these metrics are received and analyzed by Data Analyzer component, which is responsible for examining and finding abrupt changes in the behavior of the applications. Then, these metrics are sent to Classifier component that is responsible for classifying each application in a given period, defined by the previous component, into interference levels. Finally, the Scheduler module makes scheduling decisions by running an algorithm based on information that was generated by the two previous components. Figure 5.1 presents an overview of the proposed architecture, distinguishing each layer.

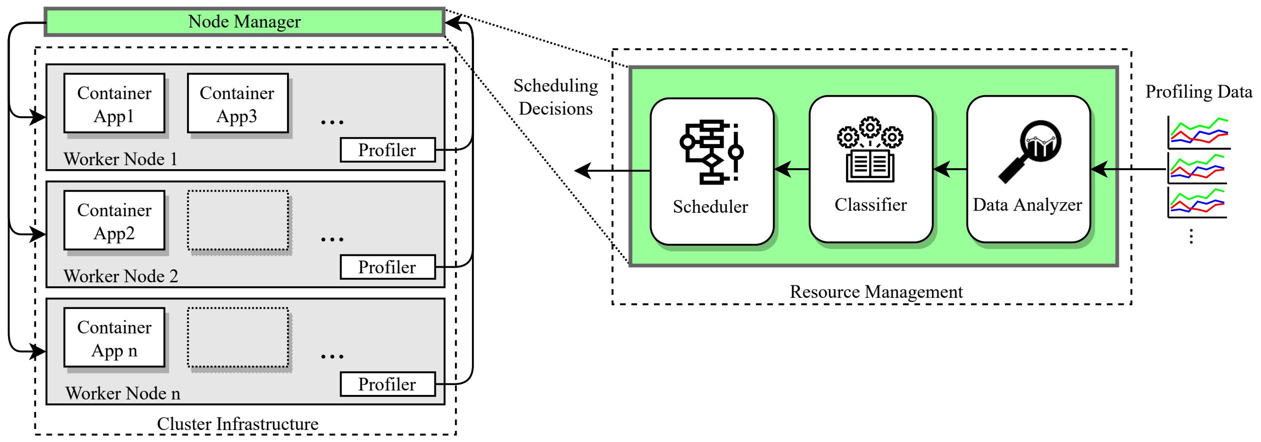

Figure 5.1 – Architecture general overview.

The Node Manager is continually monitoring and analyzing information that could potentially interfere from the cluster infrastructure. It is worth noting that the Profiler module is always monitoring the entire infrastructure while feeding the Data Analyzer and Classifier components. While both modules analyze and classify the data received, when they find there is room to make scheduling decisions, they send that information to the Scheduler module to apply it over the cluster infrastructures. Figure 5.2 depicts the architecture data flow, where it is possible to observe how collected metrics are processed through each component.

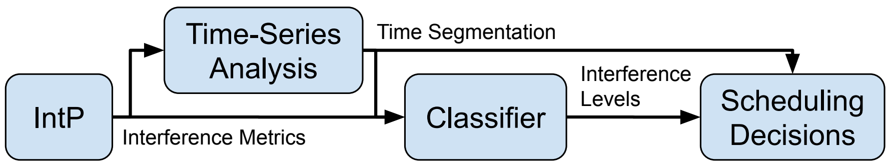

Figure 5.2 – Architecture data flow.

This cycle will always run while more than one application is running in the cluster. To clarify, let us look an example: suppose that an application1 starts on Node1, together with application2. At a certain time, application1 and application2 contend for CPU (or another resource, i.e.) and the Node Manager perceives this contention and decides to migrate the container which runs application2 to Node2. This scheduling action aims to use the resources more efficiently, improving QoS and consequently reducing SLA violations. In the next subsections, each component is presented in detail.

5.1.1 Interference Profiler

Profiling runtime applications is not a straightforward task, given that different tasks may stress arbitrary resources, causing variations in resource consumption. In addition, an intrusive profiler can induce the performance of applications and compromise reliability. The literature presents works that address resource contention aspects among applications in a simple way, either present or not present [LXK$^+$19]. Also, a number of application profiling mechanisms, ranging from kernel-based [Yag02] to runtime [USR03] profiling that use especially linked libraries, have been proposed in the past.

In this study we adopted, the previously mentioned IntP tool [Xav19], which profiles the application at execution time, returning the interference the application generates on each resource subsystem. IntP allows us to profile applications at runtime (every second), so it is possible to analyze what resources receive the most interference generated by target applications. This gives us interesting information to perform an analysis of applications' interference behavior changes over time.

5.1.2 Time-Series Analysis

We aim to evaluate the influence of application interference over time, but dealing with dynamic workloads at runtime is a challenging task. Time is an important factor that must now be considered in our model, regardless of the online trend. For example, to perform dynamic scheduling actions on-the-fly, it is necessary to define when our architecture will execute. As already mentioned, in previous work [MKDD21b], we moved a step forward and created a static-defined time interval scheme to start analyzing segmented scheduling, and preliminary results presented a considerable improvement in hardware efficiency. Since we are interested in accomplishing automatic scheduling decisions based on interference levels generated across applications, we needed to carry out a statistical time-series analysis to address the profiled data and point to patterns. However, there are some specific aspects that arise when working with time series, such as: Is this data stationary? Is there seasonality? To work around these questions, we performed an online change point analysis [Pag19] aiming at determining the time points with the most significant behavior changes, considered crucial for analyzing and classifying the profiled data, and subsequently, performing scheduling actions.

Change points are abrupt variations in the generative parameters of a data sequence. Online detection of change points is useful in modelling and predicting time series in application areas such as finance, biometrics, and robotics [Pag19]. A time series consists of multiple assessments of a specific outcome measure, at a group level, at regularly spaced time intervals. The “interruption” or “change point” of the time series is an identifiable real-world event.

Since IntP profiles each application in an isolated manner and provides multiple metrics (different resources) from each, we first had to reduce dimensionality. Therefore, we applied Principal Component Analysis (PCA) [R C19] over each application. PCA is a method to reduce dimensionality that is often used to reduce the dimensionality of datasets, by transforming a set of variables into a smaller one that still contains most of the valuable information in the large dataset. In our case, we decided to reduce the seven metrics profiled in IntP to only one for each application. Depending on the order the algorithm sorts the metrics, the PCA outcome changes. Thus, to determine how to best arrange the interference metrics, we performed several tests and decided to place the metrics in order of performance degradation priority. In previous work [MKDD21b], we introduced this priority order, arguing that when there is resource contention, some hardware components present more elevated performance degradation indexes than others. Thus, we decided to apply PCA with the following resource order: disk, memory, cpu, cache, and network. Because performance degradation caused by disk resource contention is bigger than that caused by network, for example. If there is no disk usage, PCA takes the next resource in the queue order, memory in this case. If there is no memory utilization, the next resource will be considered as the main one, and so on.

After reducing the profiled data from each application to a single dimension, observing its performance degradation priority, we applied the Online Change Point Detection (OCPD) function, from R Package [Pag19], over all applications’ metrics. This technique implements the Bayesian online change point detection to handle multivariate data, computing the set of change points with the highest probability online (updating the results with each incoming point). This method outputs a list of change points over time (x-axis) while running the model, in an online fashion. The entire process of reducing and analyzing profiled data is depicted in Figure 5.3.

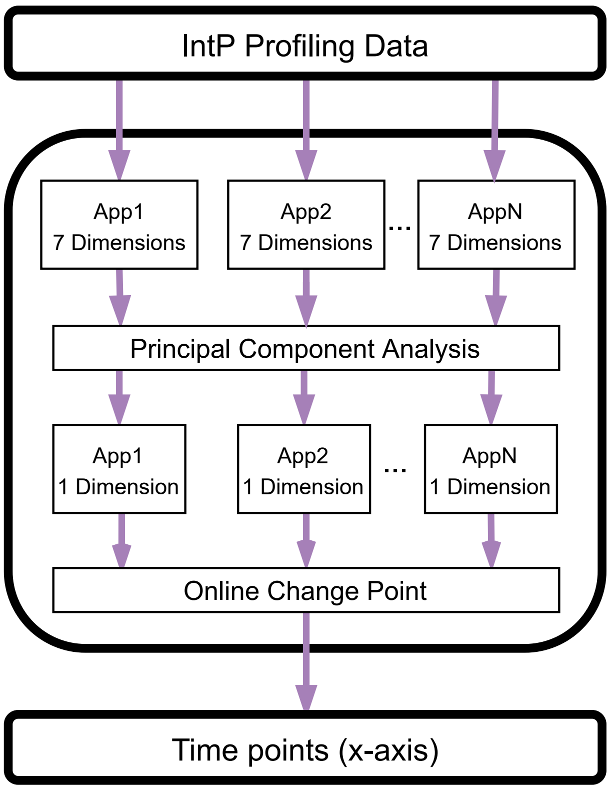

Figure 5.3 – Data scheme of data profiling (IntP), dimensionality reduction (PCA), and discovering change points over time (OCPD).

To present a simple use case example, we ran an experiment adopting Node-Tiers. First, we chose two memory-intensive applications from the Node-Tiers suite, then we created a synthetic workload for each one. Each workload purposely produced an interval with a high-load request rate: (A) between 60 and 120 seconds; and (B) between 180 e 240 seconds, accordingly Table 5.1.

Table 5.1 – A and B applications' workloads behavior.

Intervals (s)	A (req/s)	B (req/s)
0 - 60	100	100
61 - 120	200	100
121 - 180	100	100
181 - 240	100	200
241 - 300	100	100

Both applications (A and B) were executed together while profiled with IntP, and the results are presented at the top of Figure 5.4.

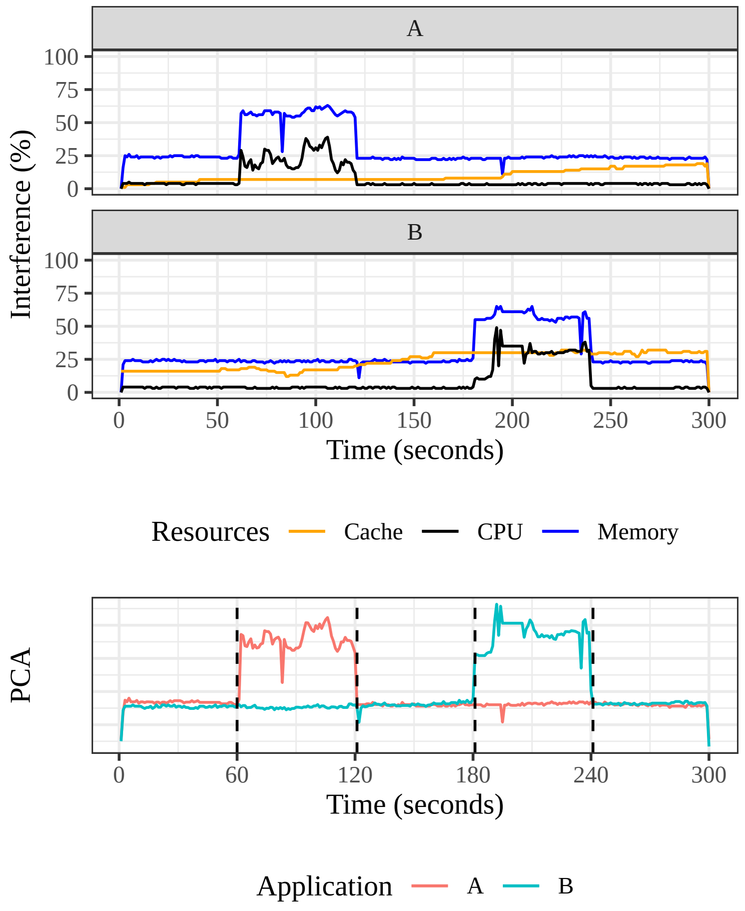

Figure 5.4 – Profiled data (IntP) from A and B execution (top); PCA resulting data along with found OCPD change points (bottom).

The metrics collected passed through the PCA phase and then produced A and B data results, shown at the bottom of the same Figure. Finally, this data was submitted to the OCPD function, returning the moments where both applications presented abrupt behavior changes (60s, 120s, 180s, and 240s), seen in the same image. It is possible to observe that OCPD is able to handle multiple applications due to its *multivariate* characteristics, making it a good candidate tool for our architecture.

5.1.3 Interference Classification

A number of techniques have been proposed regarding interference classification, such as: collaborative filtering [DK13], decision-tree [MYXW13, JG17], major interference source [KS17, DKCS18] and resources historic mean [CSG14, CSGK16]. The authors from [LXK+19], the most closely-related to our study, developed a scheduling model that considers interference levels among applications to increase resource usage. Even though the authors' approach increases the state-of-the-art in the scheduling resource field, their classification was developed with fixed thresholds that are empirically defined.

In order to find alternatives to minimize interference overhead effects on scheduling decisions, we proposed a classifier that quantifies cross-application interference in levels over time in the previous chapter. It is an interference classifier method that better represents the workload variability and improves hardware utilization. The main purpose of our classification method is to return the hardware resources' that the interference produced by applications had caused, within a time slice, to a given degree. This is achieved by exploiting the combination of two different machine learning algorithms: (i) SVM for classification and (ii) K-Means for clustering. Initially, SVM receives interference data from applications, collected each second by IntP, and those metrics are classified and stored into resource queues for their respective classes: *memory*, *CPU*, *disk*, *network*, and *cache*. Subsequently, K-Means quantifies values for each queue and returns their interference level for a specific period. In addition, we adopted four interference levels: (i) *absent*, when there is no interference incidence; (ii) *low*; (ii) *moderate*; and (iv) *high*. Both machine learning algorithms use a previously defined training dataset to assist them in making decisions.

The proposed ML-based interference classifier dynamically defines thresholds and assigns interference levels for each resource used by the monitored applications for a particular time slice, without the need for user intervention. This classification process is repeated until the end of the execution, characterizing the dynamicity of our approach. In fact, interference levels are reevaluated regularly, accordingly to the OCPD function, to ensure that we are able to better react to significant workload changes.

To present an example, we will use a decision support benchmark called TPC-H. We created an increase workload, starting with a low load and gradually moving up to a high load. This workload execution was profiled with IntP, arbitrarily divided into four segments, and each one was classified by our approach. The classification result is shown in Figure 5.5.

There are resources that do not change their labels, for instance, memory, disk, and network. Since they keep their interference metrics at the same level, on average, with no expressive variation, their labels are maintained. On the other hand, there are some resources that do change their labels, which are the CPU and Cache.

The CPU has a smooth increase in its behavior, moving from moderate to high. Cache keeps its high level label while executing, then changing to moderate and then back again to high interference levels. This highlights that, due to the dynamic nature of the workload, the application presents different interference labels during its execution.

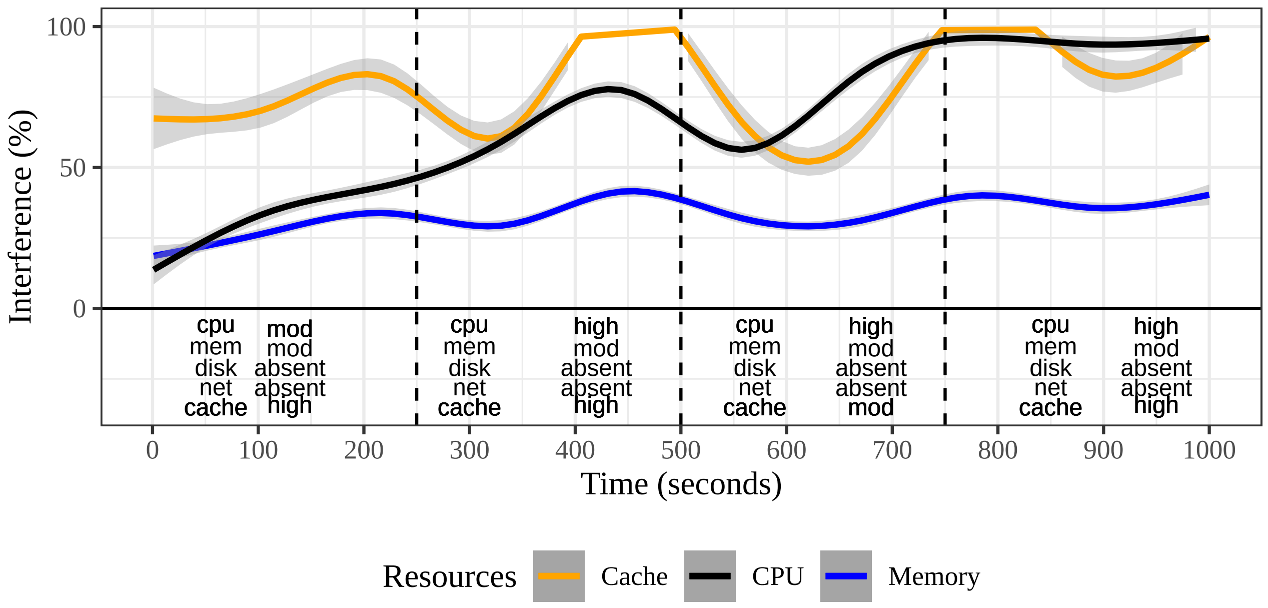

Figure 5.5 – Segmented TPC-H static interference classification. To facilitate the visualization, a Loess function was applied to smooth short-term variations in each resource. Resources labels that changed are shown in bold in the bottom plot. IntP metrics that do not suffer any interference were not depicted.

5.1.4 Scheduling Algorithm

Scheduling consists of ordering running jobs across available computational resources [HZdLZ20, TVN^{+}20]. To do this with interference awareness, the Node Manager first pulls the task into the available node slot. It then profiles the interference from each application and, based on the information generated from previous components, assigns them on the best candidate nodes to minimize the overall performance interference. IADA is an architecture that relies manly on a reactive approach, so that the applications are constantly profiled and when OCPD technique finds significant workload variations, the most recent interval data is used to perform scheduling decisions.

The applications start at zero and are monitored continuously every second. The time-series analysis evaluates the data and returns X, which is the point found by the OCPD function with the greatest relevance in the workload variation of the applications. When X is found, the classification module generates an interference label for each application resource running in each container. The interval between $X_{(n-1)}$ and X_n is defined as ΔT_n. When a ΔT_n is found, the scheduling is performed based on the most recent data, which means, the last ΔT_n outcome.

The traditional view for real-time scheduling problems focuses on how to find a feasible schedule for an application set. However, the scheduling of a given application set is not straightforward. With the rapid increase in the use of powerful cloud systems, an efficient task scheduling policy, which deals with the assignment of tasks to resources, is required to reduce performance degradation. Task scheduling is an established NP-Hard optimization problem that can be effectively tackled with meta-heuristic algorithms [CSK20]. Taking this statement into account, we decided to use a heuristic algorithm to solve our problem. Ludwig et al. [LXK^{+}19] tested many heuristics to schedule applications with interference awareness and concluded that Simulated Annealing (SA) presented the best overall results. Therefore we decided to apply a modified SA algorithm that addresses interference-aware aspects in this work.

The SA algorithm is an optimization method which mimics the slow cooling of metals, which is characterized by a progressive reduction in the atomic movements that reduce the density of lattice defects until a lowest-energy state is reached [KGV83]. In a similar manner, the simulated annealing algorithm generates a new potential solution to the problem by altering the current state, according to predefined criteria. The new state solution is then based on the satisfaction criteria, and may be accepted even if they do not lead to an improvement in the objective function.Table 5.2 shows the summary of notations that we used along this work.

Table 5.2 – Notations for the problem formulation.

Symbol	Meaning
X	The point found by the OCPD function with the greatest relevance in the workload variation of the applications.
$X_{(n-1)}$	Start of the interval analyzed by the scheduler.
X_n	End of the interval analyzed by the scheduler.
ΔT_n	The interval between $X_{(n-1)}$ and X_n.
S	Initial set of applications that IADA is handling.
P	Set of physical machines (hosts) that IADA is handling.
$S_{modified}$	New placement solution suggests every iteration to compare with the best one.
k	Index of hosts' summation, ranging from 1 to Nh.
Nh	Total number of hosts in the environment.
j	Index of applications' summation, ranging from 2 to Na.
Na	Total number of applications in each cluster node.
L	Interference level index in each resource.

Since our architecture moves applications among cluster nodes at runtime, we developed an algorithm based on SA to find the best arrangement of applications in order to minimize performance degradation. The algorithm 1 presents how our architecture scheduling policy works.

```
Data: P, A, temperature, coolingRate
Result: $solution_{best}$
s = roundRobin(P,A);
bestsolution = s;
while (temperature > 1) do
    newsolution = randomFunction(s);
    bestscore = bestsolution.getInterferenceScore();
    newscore = newsolution.getInterferenceScore();
    if (newscore < bestscore ) then
        if (bestsolution.getMig() < newsolution.getMig() ) then
            bestsolution = newsolution;
        end
    end
end
```

Algorithm 1: Optimized Simulated Annealing

Initially, the algorithm creates an application set *S*, in which each container receives one application instance to execute. All containers are then distributed among cluster nodes by a *RoundRobin* function that receives a set of physical machines *P* an a set of applications *A* to be executed. Every SA iteration generates one new solution $S_{modified}$ that is compared to the best solution at that point. This new solution is generated by the Random Swap Function, presented in Algorithm 2.

```
Data: solution
Result: $S_{modified}$
p = Math.random();
if (p < 0.5) then
    $app_1$ = getRandomApp(solution);
    $app_2$ = getRandomApp(solution);
else
    $app_1$ = getHigherScoreApp(solution);
    $app_2$ = getLowerScoreApp(solution);
end
swap($app_1$, $app_2$, solution);
```

Algorithm 2: Random Swap Function

This function relies on a randomized approach, in which the function has a 50% chance of swapping random applications in the cluster and 50% chance of swapping the application of the cluster node with highest score to the cluster node with lowest score.

After finding the new solution, if $S_{modified}$ presents an interference degradation index lower than the current one, the algorithm replaces the current (best) solution with the new one. To compare both solutions, we created a function *InterferenceScore()* that analyzes the interference levels in ΔT_n and returns a total interference score, which is calculated by using the function seen in Equation 5.1.

$$TotalIntScore_{\Delta T} = \sum_{k=1}^{Nh} IntScore_{Host}, \quad \forall k \in s \quad | \quad k \geq 1. \tag{5.1}$$

The total interference score is the result of the sum of all interference scores from each cluster node, where k represents the index of hosts' summation in the environment, ranging from 1 to Nh (total number of hosts). Each cluster node has its own interference score as well, this is calculated with a function demonstrated by the Equation 5.2.

$$IntScore_{Host} = \begin{cases} \prod_{j=1}^{Na} IntScore_{App}, & \text{if } j \geq 2 \\ 0, & \text{otherwise} \end{cases} \tag{5.2}$$

Where j denotes the index of applications' summation in each cluster node, ranging from 2 to Na (total number of applications). If there are less than two applications running in a cluster node, it will not generate an interference incidence in that specific node and consequently will return a zero-score, since only one or no one application does not cause interference. Finally, the application interference score is calculated by the Equation 5.3.

$$IntScore_{App} = cpu(L) \times mem(L) \times disk(L) \times net(L) \times cache(L) \tag{5.3}$$

All resource interference metrics (*cpu*, *memory*, *disk*, *network*, and *cache*) were measured and allocated into a level L. Depending on the level they are assigned, the interference overhead index value varies, according to Figure 5.6.

To find these Interference Degradation Indexes (IDI), first we ran applications with each resource-intensive (e.g. cpu-intensive, memory-intensive and so on) in isolation and took the average response time. Then we ran each one again co-hosted with one more application instance at a specific level (low, moderate, and high), according to the classifier method, and found the average response time from both. Based on these metrics, we discovered how much each resource degraded at each interference level by using the Equation 5.4.

$$IDI = \frac{ResponseTime_{(level+absent)}}{ResponseTime_{absent}} \tag{5.4}$$

To illustrate this scenario for memory, lets take an example: We executed a memory intensive application in isolation, which resulted in $ResponseTime_{absent}$ = 23.2ms.

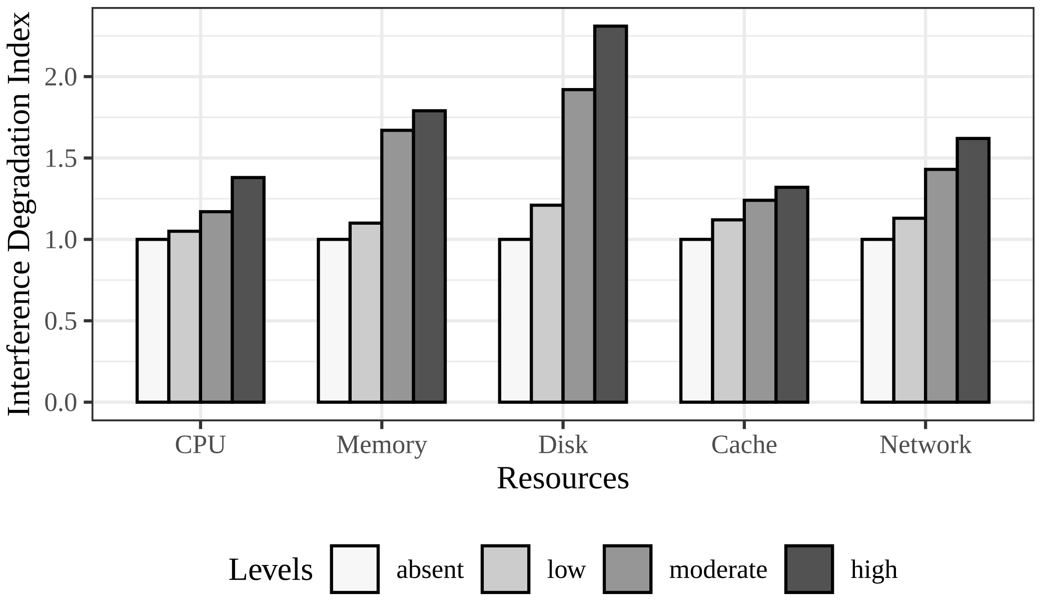

Figure 5.6 – Interference degradation index by resource.

While co-hosted with a low-intensive memory application, the runtime increased to $ResponseTime_{(low+absent)}$ = 24.4ms, which produced IDI of 1.10. When co-hosted with a Moderate-intensive application, the response time increased to $ResponseTime_{(moderate+absent)}$ = 39.2ms, resulting in an IDI of 1.69. Finally, when co-hosted with a High-intensive memory application, the response time increased to $ResponseTime_{(high+absent)}$ = 41.5ms, resulting in an IDI of 1.79.

Another important aspect that is analyzed in SA algorithm, is the number of migrations with the new generated solution. If the number of migrations performed in the new solution is higher than the best solution, this new solution is eliminated and another one is considered. The migrations number is found with the help of the $getMig()$ function, as seen in algorithm 1.

5.2 Evaluation and Results

In this section, we describe how the experiments were conducted, the scope, and the limits of the project. Also, the details about workload, application, and the computational environment adopted in this work are discussed.

5.2.1 Application and Workload

To investigate applications that present dynamic workloads (unpredictable load variation) by stressing different hardware resources, Node-Tiers was adopted. This tool explores the latency-sensitive application's concept (client-server) and enables the creation of workload variations. The goal is to stress hardware resources in many ways (distinct resources), through many latency-sensitive applications from this suite benchmark, increasing and decreasing the request arrival rate, and executing scheduling decisions at runtime, handling changes in the workload.

To create a most realistic scenario, we evaluated our architecture using three real-world workload traces. The first one was from NASA [1] dataset, consisting of all the requests made to the 1998 World Cup Web site between April 30, 1998, and July 26, 1998. The second one was from the Wikimedia project, found in Wikipedia [2] traces. Specifically, we collected the page view statistics for the main page in the English language for the month of January 2021. The last one is from Alibaba Open Cluster Trace [3], this one is sampled from one of our production clusters. There are both online services and batch workloads, and we collected only information from Sigma, the online service scheduler.

5.2.2 Experiments Scenarios

To explore the efficiency of our architecture, we dived all experiments into two phases: first, we used a real-scenario with a small number of machines to ensure all chosen technologies worked together correctly and to guarantee the simulation phase outcomes were real, reflecting reliable results; and second, based on the previous phase, we built a simulated environment, to test our architecture with a bigger number of cluster nodes, and consequently, more applications. In the next sections we describe how each phase was performed and the results.

5.2.3 Practical Experiments

To run our experiments within a practical testbed, we used the Pantanal cluster from the LAD Laboratory [4] from PUCRS. This cluster has Dell PowerEdge R740xd nodes, each one equipped with: 2x Intel Xeon Gold 5118 Processor, 300GB DDR4 RAM Memory,

[1]ftp://ita.ee.lbl.gov/html/contrib/
[2]https://dumps.wikimedia.org/other/analytics/
[3]https://github.com/alibaba/clusterdata/tree/master/cluster-trace-v2018
[4]https://www.pucrs.br/ideia/lablad/

1TB Hard Drive, and 4x Gigabit Ethernet Interface. Also, as the Node Manager, we used one Dell Optiplex 990 outside of the cluster, equipped with 8GB of RAM, and one Core i5 processor.

To stress different resource subsystems (CPU, memory, disk, network, and cache), we used different applications from the Node-Tiers suite. The server-side was performed over the cluster, while the client-side was configured on a single computer, using Artlillery [5], a load stress testing tool. The server-side applications were executed inside containers, more specifically one container per application. Since containers present many benefits compared to traditional virtual machines, we decide to adopt LXC/LXD containers as a target virtualization technology, allowing us to schedule the applications with live migration across the cluster nodes with Checkpoint/Restore In Userspace (CRIU [6]) functions. All pieces of equipment were connected through a Gigabit Ethernet Network.

We ran four applications inside each cluster node, and each one was submitted to two hours of a workload trace (randomly chosen), mixing the chosen datasets and creating greater variation among application workloads.

To evaluate the proposed architecture efficiency, we compared it to three schedulers found in the literature: (i) EVEN, the state-of-the-art round-robin scheduler, also considered a baseline in this work; (ii) CIAPA, considered the most closely-related work that applies interference levels to make placement decisions; and (iii) Segmented, a scheduling scheme from our previous work. More details are presented below:

- **EVEN** implements the *EvenScheduler* algorithm, which is the Apache Storm [7] default scheduler. This algorithm distributes computation tasks across nodes in a Round-Robin manner [ASZ20]. When tasks are scheduled, this approach counts all available slots on each node and places application instances to be scheduled one at a time to each node while keeping the order of nodes constant. We decided to use this method because Apache Storm is a well-known framework that processes real-time data, like cloud multi-tenant systems, which are the target applications in this work. Moreover, we consider EVEN as a baseline since it is the less optimized approach. In this case, the applications are placed into the cluster nodes in a round-robin fashion, meaning that they are not moved while the experiment is executing.

- **CIAPA** [LXK$^+$19] evaluates the profile of the application workloads and uses an interference classification in levels. This approach was chosen because it is the work most closely related to ours. The difference between this work and ours lies in the fact that the classification is static, done only one-time at the beginning of the execution, and the definition of interference levels thresholds are fixed and empirically

[5]https://artillery.io/
[6]https://criu.org/
[7]https://storm.apache.org/

defined. Firstly, the applications are placed in a round-robin manner and then the collected data is analyzed in 10-minute intervals, and only one scheduling movement is done. The placement of the application is not changed after that.

- **Segmented** [MKDD21b] applies an interference classification using levels, similar to CIAPA. The difference is that the Segmented scheduler arbitrarily divides the applications' executions into four parts (proportional), and based on that division, it classifies the generated interference in levels per segment. The goal of this approach is to find an alternative to classify applications' interference considering workload variability, not only using a simple average over the entire execution.

For this experiment phase, we used four nodes of the Pantanal cluster, each one executing four Node-Tiers applications, totaling 16 applications. When using latency-sensitive applications, the response time (latency) metric quantifies how long the user must wait for a response to a query, regardless of the quality of the response. Together with data quality metrics, latency metrics provide the best indication of the end-user experience under normal conditions and during outages [Bro04]. For this reason, we decided to use the *Average Response Time* as the main performance metric in these experiments, which represents the total latency for the test divided by the number of requests submitted to the server-side by the users-side. The response time was collected from each application during the entire experiment with Artillery, and their total average is presented in Figure 5.7.

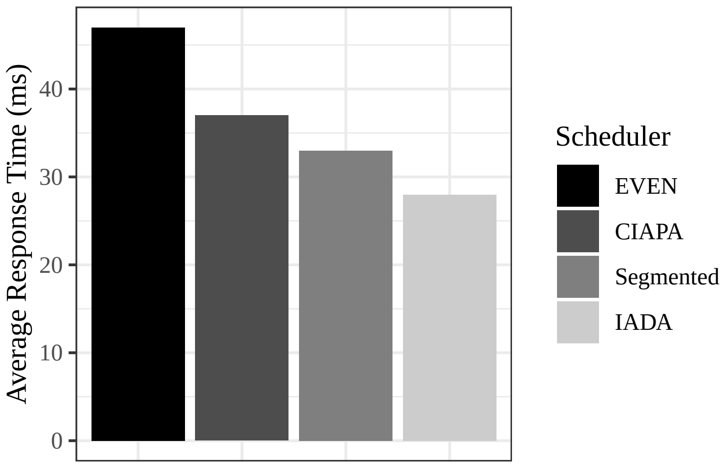

Figure 5.7 – Average Response Time from practical experiments phase.

It is worth noting that in all experiments, our proposed architecture presented the best results, improving the average response time by 26% when compared to other solutions. When compared to EVEN scheduler, our proposed architecture reached a 40% average reduction of response time. EVEN scheduler reached the highest response time indexes (worst results), as predicted since this scheduling strategy is not interference-optimized. Compared to the CIAPA approach, IADA reduced the average response time by 24%. In contrast to Segmented, IADA obtained a reduction of 15% of the average response time.

By running 16 applications with a mix of workload variations, IADA detected 12-time points with expressive (global) behavior change. Consequently, 12 periods were classified and each one provoked scheduling actions. As mentioned before, the Data Analyzer component uses a bayesian change point detection to find a workload behavior modification, but this does not imply that all applications, in all analyzed intervals, had their interference labels modified, changing their interference degree (levels). The applications only have their labels modified if the workload variation has an abrupt alteration.

To give an example of how much the interference in a node is affected by scheduling actions, we collected the average interference generated in Node1 and Node2 within a given period, every second, while a scheduling rearrangement was performed. This is presented in Figure 5.8.

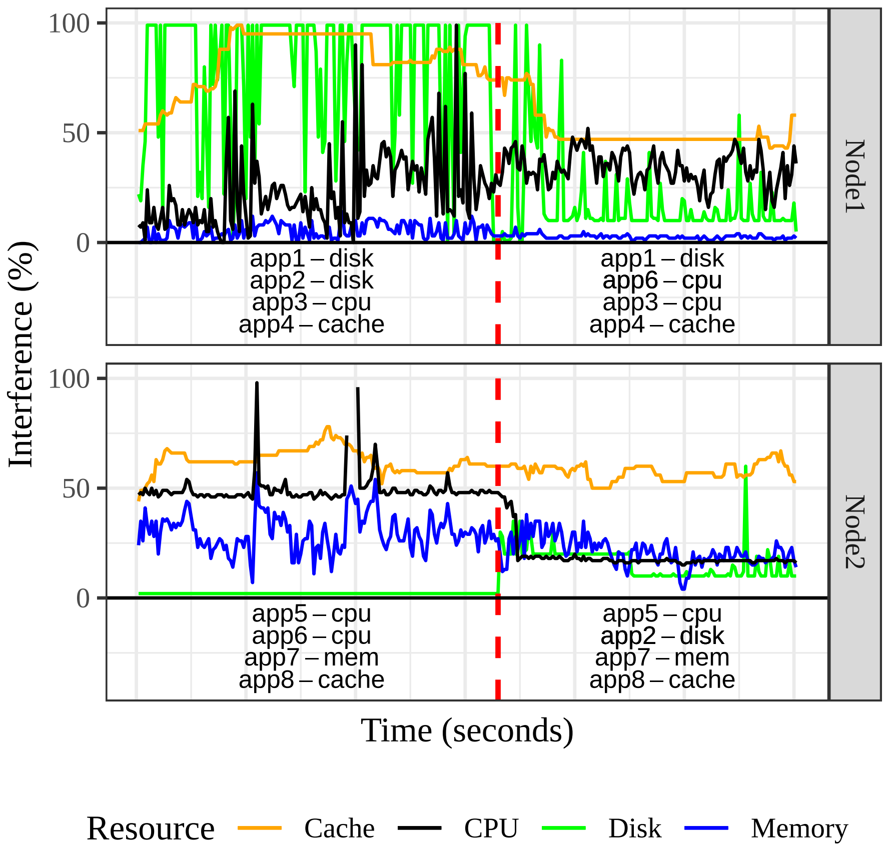

Figure 5.8 – Average interference indexes in Node1 and node2 while performing a scheduling action. Applications which have migrated across nodes are shown in bold.

This image illustrates the average interference generated by the 8 applications running in Node1 and Node2 before and after a scheduling change. The red broken line demonstrates the exact moment the scheduling was executed.

By looking at the interference measures, it is possible to perceive two interesting facts: (i) after the scheduling, Node1 and Node2 exchanged app2 (disk intensive) and app6 (cpu intensive) applications (in bold at the image); and (ii) after this rearrangement, Node1 had its disk interference ratios considerably reduced while Node2 had its disk ratios increased. Also, Node2 had its cpu interference indexes reduced while Node1 had its overall cpu usage increased. In general, it is possible to observe that this scheduling operation provided a balance across interference indexes, improving resource usage and reducing the response time of the applications.

Therefore, these results show that a technique that frequently analyzes the generated interference over time is able to reduce the overall system's overhead, using the infrastructure more efficiently, and consequently, improving QoS requirements. Also, these experiments show that the proposed architecture presents interesting and trustworthy outcomes, and they will be used to calibrate and perform the simulation experiments, presented in the next section.

5.2.4 Simulated Experiments

In order to carry out experiments closer to a real scenario, a large physical machine set is necessary. To have more flexibility to perform different host arrangements, we decided to scale our approach through simulation as well. First, we searched in the literature for tools that simulate cloud infrastructures [LSN+09, KBK12, NLC+12, CRB+11]. After exploring each simulation tool, we concluded that none of them offerd an environment able to handle interference aspects from applications. Then, we confirmed that CloudSim [CRB+11] is the most widely spread cloud simulator and by far the most sophisticated. It was developed as an add-on-top of the grid network simulator GridSim [Cas01]. CloudSim is a completely customizable tool that supports modeling, creation of one or more VMs, and mapping tasks to appropriate virtual machines. This gives CloudSim the ability to handle a complex simulation environment. It mainly targets application developers or testers as it gives the ability to configure several variables such as the number of users, data centers, and cloud resources along with the location of both users and data centers. Though there are many studies extending CloudSim, such as [BB12, GMC+13, XRR+17, KMX+20], one in particular supports Container as a Service (CaaS), namely ContainerCloudSim [PDCB17]. This extension provides a platform for modeling and simulating containerized cloud computing environments. Therefore, it is the most fitting simulation tool to use nowadays and the one we chosen to extend with application interference features. We developed the CloudSimInterference version, which is a trace-driven simulation tool. Thus, to perform our simulated experiments, first each application execution was performed in a physical machine and all IntP metrics were kept and used as input for our workload simulations.

To implement our simulation plugin, we made many class modifications in ContainerCloudSim. First, we extended the *containerCloudlet()* class to *InterferenceContainerCloudlet()*. This class is responsible for representing the application behavior, and we included all Interference metrics measured with IntP, each keeping the information from each application trace generated from the real execution. Another major modification, was the integration with the R algorithms to perform the OCPD and ML functions, presented in the sections 5.1.2 and 5.1.3. To perform this integration, we included the JRI (Java-R-Integration) library on the java side and the rJava library on the R side. Also, we extended *containerDatacenter()* class to *InterferenceContainerDatacenter()*, including several functions to handle the modifications done with interference metrics utilization. All source codes are available in a GitHub repository[8].

To generate a considerable number of applications (*InterferenceContainerCloudlets*) for the simulation experiments, we executed several hours of each workload trace (seen in section 5.2.1) with five application instances from Node-Tiers suite, stressing the main hardware resources (cpu, memory, disk, net, and cache). Then, we randomly divided those execution traces collected with IntP into two-hour segments to use as input data in our simulation experiments.

To run the simulated experiments, we used four different arrangements of cluster node numbers and application instances, presented in Table 5.3.

Table 5.3 – Hosts/applications arrangements used in simulated experiments.

Hosts	Applications
6	24
12	48
24	96
48	192

As mentioned before, we used the real experiments as a base to calibrate our simulator. To produce more reliable results and as close as possible to real case scenarios, we continued to use four applications in each node as well. Figure 5.9 presents the results from the simulated experiment phase.

It is noteworthy that in all experiments, IADA achieved the best results (lower indexes). When compared to EVEN scheduler, our proposed architecture reached a 37%

[8]https://github.com/ViniciusMeyer/CloudSimInterference

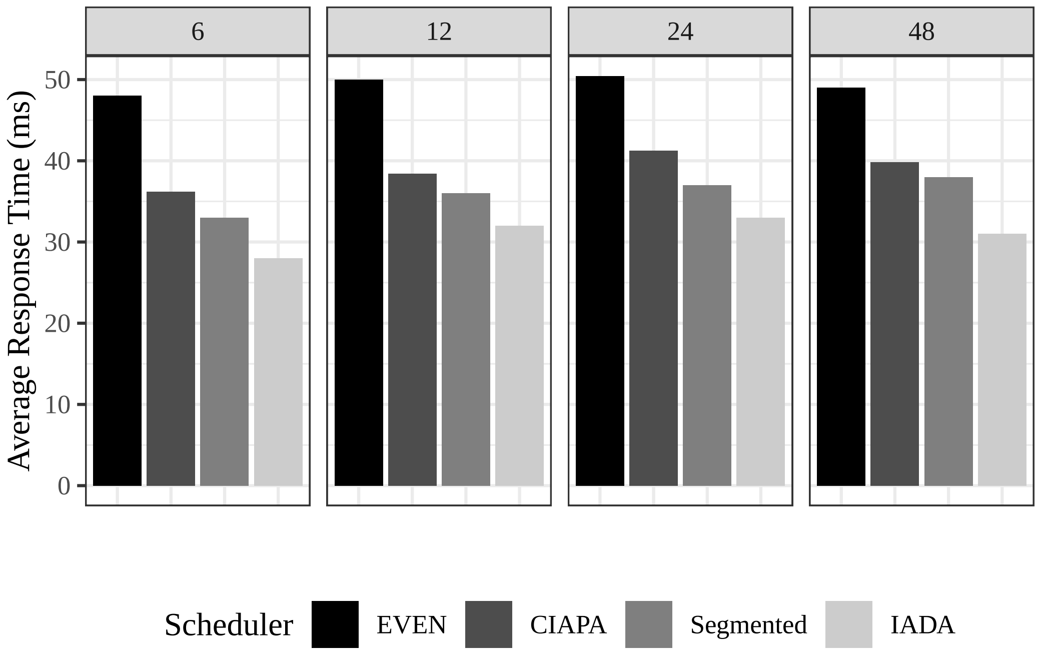

Figure 5.9 – Average Response Time from simulated experiments phase in each host arrangement (6, 12, 24, and 48).

average reduction of response time. Not surprisingly, EVEN had the worst results as well, similar to the real experiments, because this scheduler was not developed based on interference-awareness. Compared to the CIAPA approach, IADA reduced the average response time by 21%. In contrast to Segmented, the state-of-the-art strategy, IADA obtained a reduction of 14% of the average response time.

To analyze the variation of response time, we took the average response times in each scheduled interval, during the experiments running with 24 nodes (96 applications), and compared them with EVEN, CIAPA, and Segmented schedulers' results. These results are presented in Figure 5.10.

It is interesting to observe that EVEN scheduler places the applications at the beginning of the execution and they are not moved later. That is the reason its representation in the image is a straight line, depicting the total average in the entire execution. Something similar happens with the CIAPA strategy, within the first 300 seconds the interference metrics are collected and analyzed, and after that just one placement decision is made, not executed again, and it is represented by a straight line as well in the total average over the entire execution. In the Segmented approach, the application execution was divided into four segments, and at the end of each one, scheduling actions were taken. This strategy improved the overall response time compared to EVEN and CIAPA strategies because it adjusted the infrastructure to applications, taking into account the variability of workloads [MKDD21b], even considering only a few segments (four in this case).

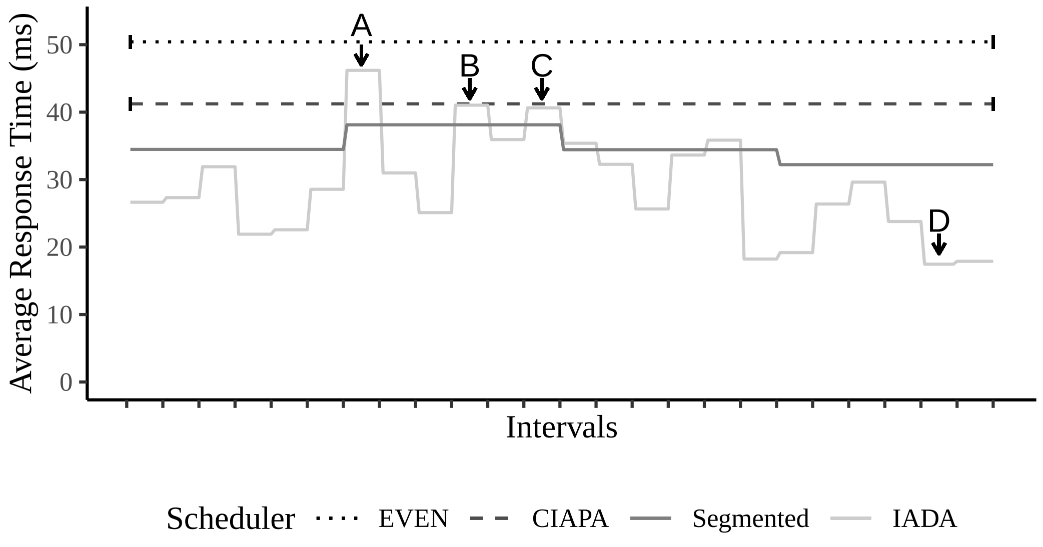

Figure 5.10 – Average Response Time in each scheduled interval from the 24-nodes experiment.

In general, IADA achieved the lowest response time rates (best results). However, there was an interval that presented worse results than CIAPA's scheduler, depicted in point A. This happened because IADA relies on a reactive approach, using the most recent data and not a general view to make scheduling decisions, so that in interval A the workload had an abrupt behavior change, not presenting the best scheduling arrangement, but still is considered as an acceptable result.

There were intervals that IADA was close to CIAPA's outcomes, which were the B and C intervals' cases. Also, the major response time reduction can be seen in the interval D, reaching an average improvement of 57% in relation to EVEN, CIAPA, and Segmented scheduling approaches.

It is important to highlight that the proposed scheduling architecture adjusts applications over the hardware based on the workload oscillation, in real-time, and in a reactive manner. So far, the outcomes found in this work support our idea that an interference-aware dynamic scheduling architecture designed to observe workloads tends to reduce the overhead generated by cross-application interference over the system, and consequently utilizing the available hardware resources more efficiently.

With these results, the RQ3 is answered, affirming that to move from static to dynamic interference-aware scheduling some architectural changes must be made. We discovered that a common approach used in the state-of-the-art is to profile applications, somehow classify them, and make scheduling actions, which was what we did in the previous chapter. To build a dynamic interference-aware scheduling architecture, we had to include in our Node Manager, a module that finds the right moments to analyze profiled data and coordinate with the entire system. This component guarantees that our architecture has the capability of performing scheduling decisions automatically, without user intervention. Also, we had to develop some modifications in the scheduling algorithms to reduce the number of migrations as much as possible, since we are performing scheduling actions at runtime and this type of operation can generate more overhead than we are trying to prevent, if not controlled. It turns out that our strategy tackled this issue, ensuring improvement in hardware utilization and reduction in applications' degradation.

5.2.5 Overhead Evaluation

Considering the dynamic nature of the problem, it was necessary to run some preliminary steps in order to make the scheduling decisions. As mentioned before, those steps are profiling applications, analyzing time-series data, and performing interference classification with ML techniques. After that, with a heuristic-oriented algorithm, it was possible to execute scheduling actions.

All these techniques put some overhead pressure on the cluster system, as well as on the Node Manager. To examine these aspects, in the next sections, we performed some analysis to find out if the overhead generated by the proposed architecture could make it infeasible.

Overall Migration Cost

When running experiments with the practical scenario, we performed many container migrations across the cluster. In terms of hardware resource usage, the overhead rate created by a single LXC/LXD migration can be considered low over the entire computational environment. To present how much this operation affected the system, we executed one container migration and profiled LXC/LXD processes with IntP. Figure 5.11 illustrates the hardware utilization while performing a single container migration across the cluster.

In our experiments the migration time took about 18 seconds to be performed, on average. However, when the number of container migrations increased, the resulting overhead considerably increased as well. So, such operations should be minimized as much as possible. As mentioned in section 5.1.4, IADA uses a heuristic to find the best applications' set to schedule applications over the cluster. We developed an optimized version of CIAPA [LXK$^+$19] scheduler algorithm, taking the number of migrations into account when deciding the best scheduling actions. The IADA scheduling algorithm was developed to dynamically deal with workload behavior changes, adjusting its decisions considering the most recent data and its behavior. It is important to keep the number of migration operations at the minimum, and therefore, the amount of migrations operations

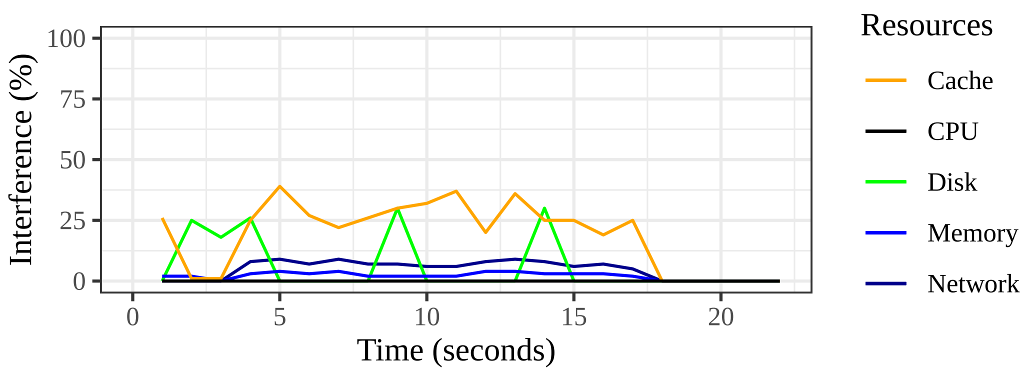

Figure 5.11 – Resources' behavior while running a container migration across cluster nodes.

is contemplated as a quality measure to decide if the new solution created is better than the actual one.

Considering that IADA automatically finds the best moments to classify data intervals, and it runs scheduling decisions based on this, in all experiments we also observed the number of intervals found by the Data Analyzer component and how many migration actions were performed. These metrics are presented in Table 5.4 for each host arrangement.

Table 5.4 – Number of intervals found by Data Analyzer component and how many migrations were performed per host arrangement.

Hosts	Applications	Intervals	Migrations	Migrations/Interval
6	24	18	115	6
12	48	23	245	11
24	96	24	712	30
48	192	27	1323	49

Observing this table, it is evident that the more applications IADA is controlling, the more intervals will be found. The more data (more applications with distinct workload patterns) the proposed architecture is analyzing, the greater the amount of information to be processed, consequently increasing the dynamism in the environment and generating greater optimization opportunity. Of course, this is also highly dependent on the variation of workloads, but since we are monitoring dynamic applications, these are not unexpected outcomes. Looking at the number of migrations performed, the more applications running in the cluster, the greater the number of migrations as well, practically following a linear trend with applications' number. At the first sight, the number of migrations performed with 192 applications seems to be exaggerated, but when dividing the number of migrations by the interval (Migrations/Interval), it is possible to notice that the number of migrations makes sense, being proportional in relation to the number of hosts, almost one migration per host per interval, demonstrating a reasonable outcome.

Machine Learning Utilization Boundaries

According to section 5.1.4, when P is defined, then Δ_T is established. After, this data interval is sent to the Classifier component so that depending on the data quantity (interval length), this process time can vary. On average, in our experiments, each application took about 1 second to be classified. To improve the performance of this proceeding, we used the *doParallel* library [CW20] from R packages. This library can be adopted to send tasks (encoded as function calls) to each of the processing cores on a machine in parallel. This is done by using a function that distributes the tasks to multiple processors. After this function gathers the responses up from each process call and it returns a list of responses, which is the same length as the list or vector of input data (one return per input item). To speed the classification process up, we performed the ML training phase previously, generating *RDA* files. These files are the results from the R programming language within the training dataset phase so that it is not necessary to execute the model training phase each time the classification process is performed.

Depending on the number of applications running inside the cluster and the length of data sent to the Classifier component, the ML analysis can take longer to be performed. In our experiments, the largest classified interval took less than 25 seconds, which is very reasonable, since each scheduling decision was not performed within less than a 20-seconds interval, that is the meantime to migrate containers across the cluster nodes. To mitigate how much time the classification process takes, in an isolated manner, we performed a partial experiment only with this component. It was applied a set of application workloads quantity with varied interval lengths. We used fixed intervals, between 30 and 600 seconds, with a 30-seconds variation. For each selected interval, we ran 24, 48, 96, and 192 application workloads, from the simulated experiment phase. Figure 5.12 presents theses results.

When looking at this image, it is possible to observe the classification follows a linear trend, which is already expected since the classification is not a distributed process, and at certain times we are allocating more tasks than the number of cores our Node Manager owns. It is interesting to notice that it takes less than 30 seconds to accomplish the classification of 192 application workloads with a 10-minutes interval length, meaning the biggest application quantity with the largest period in this experiment. This result can be considered acceptable if the target applications do not have workloads with extreme behavior patterns variability.

In conclusion, if there are an expressive number of applications or the length of the monitored period will be increased, it could be necessary to adopt a different machine

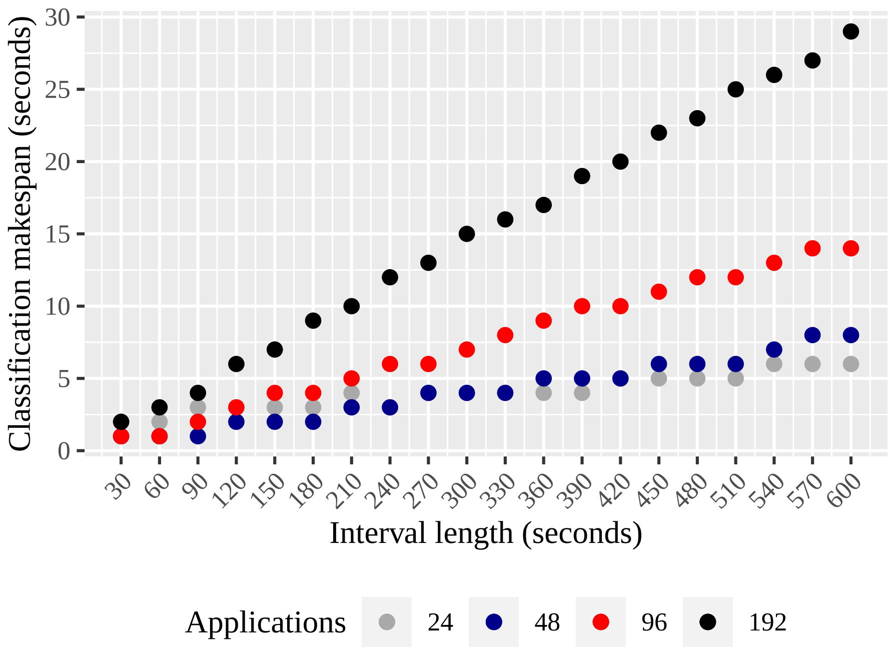

Figure 5.12 – Classification makespan from a set of applications with different interval lengths.

with more computational power (more CPUs) than our Node Manager, in order to ensure the architecture works correctly.

A Non-intrusive Profiler

To ensure that IntP does not input a considerable overhead over the hardware, we ran an experiment with NAS Parallel Benchmarks (NPB)[9], which are a small set of programs designed to help evaluate the performance of parallel supercomputers. First, we ran BT.d, CG.c, DC.b, EP.d, LU.c, MG.d, and UA.c algorithms without any profiler. Then, we ran each one again with IntP profiling them. Each execution was performed 10 times and the resulting meantimes are presented in Figure 5.13.

Since IntP core works with low-level kernel events instrumentation, in our experiments, when it was enabled, its execution practically did not generate overhead. Meaning that IntP plays a non-intrusive role over the entire system.

General Findings

When a layer of control is included in a system, it is expected to incur a certain level of overhead. The larger the environment the system will control, the greater the amount of expected overhead. In this section, we analyzed the three major overhead sources of our proposed architecture: Migration movements, Machine Learning tech-

[9]https://www.nas.nasa.gov/publications/npb.html

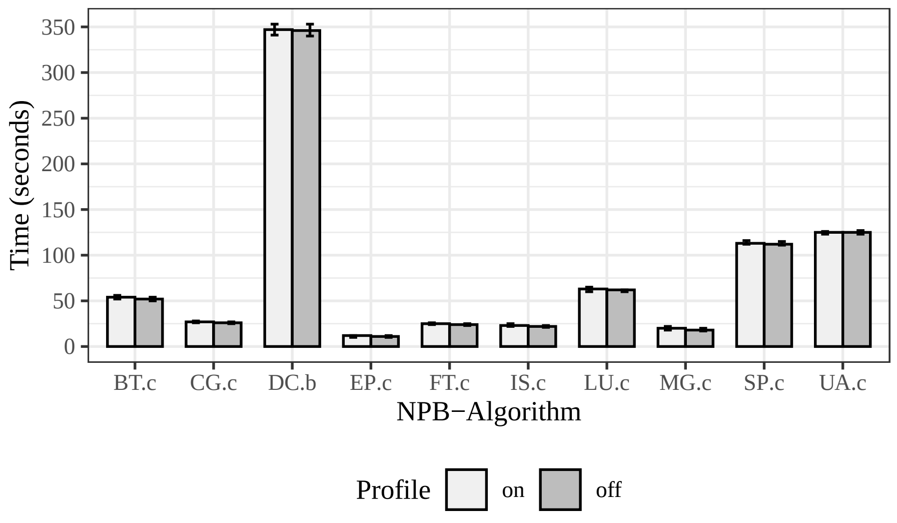

Figure 5.13 – NAS Parallel Benchmark (NPB) algorithms' executions with and without IntP profiling them.

niques, and the profiling strategy. The modifications carried out in the scheduling algorithms presented satisfactory results in terms of the number of migrations, which could be scaled out without an excessive increase in overhead. To classify the running applications, our approach uses machine learning techniques, and these techniques add a layer of overhead. However, after analyzing the results, we found that even with a large number of applications, the classification overhead is still in an acceptable ratio. Finally, we ensured that our profiling technique did not add a substantial layer of overhead to the system. After processing all of the outcomes, RQ4 was answered, confirming that there are ways to implement the changes in the proposed architecture in a manner that controls the overhead in an acceptable status, not invalidating the gains of the improved scheduling.

6. RELATED WORK

Many studies have been previously conducted on building interference-aware scheduling strategies. The main challenge is to find fast and scalable tools for addressing real-world applications. Furthermore, with virtualization technology, it has become possible to easily consolidate and quickly adapt resource allocation. Consequently, many recent efforts have studied performance interference issues in light of these new capabilities. In this chapter, we focus solely on those works that are concerned with interference-aware scheduling issues.

Kansal and Ghaffarkhah [NKG10] claim that cloud providers should transparently provide additional resources as necessary to achieve customer performance requirements if they are running their applications in an isolated manner. Taking this statement into account, the authors developed Q-Clouds, a QoS-aware control framework that alters resource allocations to mitigate the effects of performance interference. Q-Clouds uses online feedback to build a multi-input multi-output (MIMO) model that captures performance interference interactions and uses it to perform closed-loop resource management. Q-Clouds dynamically provisions underutilized resources to enable elevated QoS levels, thereby improving system efficiency. To make its scheduling decisions, Q-Clouds must first determine applications' requirements. Therefore, to begin with, VMs are profiled on a staging server to measure the amount of resources needed to attain a desired level of QoS in an interference-free environment. The resource capacity determined in the staging phase dictates what the VM owner will pay, regardless of whether additional resources need to be provisioned in real-time due to performance interference. Then, Q-Clouds is able to run its hardware rearrangements based on Q-States, previously defined. Experimental evaluations using several workload mixes from well-know benchmark applications show that performance interference is mitigated completely when feasible, and system utilization is improved by up to 35% using the authors' solution.

Aiming to reduce the runtime and improve the I/O throughput for data-intensive applications in a virtualized environment, Chiang and Huang [CH11] created TRACON, a Task and Resource Allocation CONtrol framework that deals with the interference effects from data-intensive applications, trying to improve the system performance, and consequently minimizing the overall overhead. TRACON utilizes machine learning-based techniques to perform an interference model prediction that infers application performance from resource consumption observed from different VMs and an interference-aware scheduler that is designed to take advantage of the model for effective resource management. Data-intensive applications normally consume a significant amount of I/O bandwidth and CPY cycles, therefore the authors decided to assign each application four key characteristics: (i) the read throughput, (ii) the write throughput, (iii) the local CPU utilization in the current guest VM domain (DomU), and (iv) the global CPU utilization in the virtual machine monitor (Dom0). TRACON explores three different scheduling strategies:

- MIOS - an online scheduler that reduces the queuing time for each incoming task by quickly dispatching them to various virtual machines. This technique was designed based on the concept of the minimum completion time (MCT) heuristic [BSB+99]. With the goal of minimizing the sum of execution times of all tasks, MCT maps each incoming task to the machine that completes the task in the shortest time;
- MIBS - a batch scheduler that pairs the incoming tasks based on predicted interference. It is based on the concept of the Min-Min heuristic [IK77]. In a batch scheduling scenario, the scheduling process takes place when the queue that holds the incoming tasks is full. Firstly, the Min-Min heuristic finds a machine with the minimum score for each task on the queue. Secondly, among all task-machine pairs, Min-Min finds the pair with the minimum score and assigns the selected task to its corresponding machine. This procedure repeats until the queue is empty.
- MIX - a mixed scheduler that aims to balance between both batch and online scheduling. It gives every job a chance to be the first job in the queue when executing MIBS, and hopes that future assignments can possibly offer new opportunities for better scheduling decisions. The obvious drawback here is that for each task, the delay may be increased, although the overall performance could potentially be improved.

MIOS has the lowest scheduling overhead, MIX has the potential to achieve the best performance while incurring the highest possible overheads, and MIBS is in between which can lead to a good balance between the scheduling performance and overhead. Evaluation results show that TRACON can achieve up to 50% improvement in application at runtime, and up to 80% for I/O throughput for data-intensive applications in virtualized data centers.

Zhu and Tung [ZT12] developed an interference model which predicts the application QoS metric. The authors built an influence matrix that estimates the additional resources required by an application to overcome the interference of consolidated applications and achieve a desired level of QoS. The most important feature is the consideration of time-variant inter-dependency among different levels of resource interference. The application resource utilization depends on the workload that could be dynamic throughout the application's execution, so the authors apply the *Kalman filter* to predict the application workload with the assumption that applications' future behavior is related to execution history. To prove the effectiveness of the proposed model, the authors tested several applications from a test suite and SPECWeb2005 and achieved an average prediction error of less than 8%. In addition, they demonstrated that using the proposed interference model to optimize the cloud provider's metric (number of successfully executed applications) to realize better workload placement decisions and thereby maintaining the user's application QoS.

Delimitrou and Kozyrakis [DK13] propose Paragon, an online and scalable data center scheduler that is heterogeneity and interference-aware. Paragon is an improvement of robust analytical methods and instead of profiling each application in detail, it leverages information the system already has about applications it has previously seen. It applies collaborative filtering techniques to accurately and quickly classify an unknown, incoming workload with respect to heterogeneity and interference in multiple shared resources, by identifying similarities to previously scheduled applications. In an offline setup, the authors choose applications and profile them on all different server arrangements. Then, they normalize and save the performance results. This only needs to happen once. If a new configuration is added in the DC, these applications must be profiled on it and the results added to the past results. In the online mode, when a new application arrives, Paragon profiles it for a period of 1 minute on any two server configurations, inserts it as a new performance result and uses this information to perform its system decisions. Co-scheduled applications may contend for a large number of shared resources. Therefore, the authors identified ten such sources of interference (called SoI) and designed a tunable microbenchmark for each. SoIs span resources such as memory (bandwidth and capacity), cache hierarchy (L1/L2/L3 and TLBs) and network and storage bandwidth. Paragon also evaluates the accuracy of interference classification addressing single- and multi-threaded workloads and the same systems for heterogeneity classification. The average error achieved by the classifier is 5.3% in estimating interference across all SoIs. Results show Paragon keeps QoS guarantees for 52% of the applications and bounds degradation to less than 10% for an additional 33% out of 8,500 applications.

Bu et al. [BRX13] introduce a task scheduling approach to alleviate interference and simultaneously preserving task data locality for applications with MapReduce characteristics (Hadoop). The authors' strategy includes an interference-aware scheduling policy, based on a task performance prediction model, and an adaptive delay scheduling algorithm for data locality improvement. The interference and locality-aware (ILA) scheduling strategy have been developed in a Hadoop virtual cluster. ILA's engine works with four major components: (i) the Interference-Aware Scheduling Module (IASM) to analyze the interference between tasks running on co-allocated virtual machines supported by an interference prediction model; (ii) the Locality-Aware Scheduling Module (LASM) keeps good data locality for map tasks by applying an Adaptive Delay Scheduling algorithm; (iii) the Task Profiler (TP) estimates the task's requirements of each job and feeds task information to IASM and LASM modules; (iV) the ILA scheduler instructs IASM and LASM modules to execute interference scheduling actions. The scheduler module executes the interference and locality-aware scheduling movements based on fair scheduling. At each interval, ILA selects a task from a queue, according to the job's fairness. However, occasionally the goal of alleviating interference and maintaining data locality may conflict. In this case, the ILA scheduler always considers the interference mitigation first, since virtual machine interference causes much more performance degradation than remote data access and no interference is a precondition for achieving good data locality. Effectiveness and efficiency evaluations on a 72-node Xen-based virtual cluster showed that ILA was able to achieve a speedup of 1.5 to 6.5 times for individual jobs and improvement system throughput up to 1.9 times compared to the four other MapReduce schedulers.

Zang et al. [ZRWZ14] propose two schedulers: one in the virtualization layer designed to minimize interference on high priority interactive services, and one in the Hadoop framework that helps batch processing jobs meet their own performance deadlines. The authors' approach uses performance models to match Hadoop tasks to the servers that will benefit them the most, and deadline-aware scheduling to effectively order incoming jobs. The combination of these schedulers allows data center administrators to safely mix resource-intensive Hadoop jobs with latency-sensitive web applications, and still achieve predictable performance for both. Together, these schedulers form the Minimal Interference Maximal Productivity (MIMP) system, which enhances both the Hypervisor's scheduler and the Hadoop job scheduler to better manage their performance. MIMP monitors resource usage information on each node to support task scheduling and prevent overload. MIMP runs a monitoring agent on each dedicated and shared node, and sends periodic resource measurements to the centralized Job Scheduler component. MIMP tracks the CPU utilization and disk read and write rates of each virtual machine on each host. These resource measurements are then passed on to the modeling and task scheduling components. The evaluation shows that both schedulers allow a mixed cluster to reduce web response times by more than ten fold while meeting more Hadoop deadlines and lowering total task execution times by 6.5%.

Chen et al. [CRO$^+$15] present CloudScope, a system for diagnosing interference for multi-tenant cloud systems. It employs a discrete-time Markov Chain model for the online prediction of performance interference of co-resident VMs. It uses the results to optimally (re)assign VMs to physical machines and to optimize the Hypervisor configuration, e.g. the CPU share it can use, for different workloads. CloudScope has tree main components: (i) The Monitoring Component collects application and virtual machine metrics at runtime. A daemon script reads the resource usage for Dom0 and every virtuak machine within Dom0 via xentop. The resource metrics include CPU utilization, memory consumption, disk I/O request statistics, network I/O, number of virtual CPUs (vCPUs), and number of virtual network interfaces. External monitoring tools are used to keep track of application SLOs in terms of application metrics such as response time, disk/network throughput, or job completion time. The resource and SLO profiling metrics are fed to the Interference Handling Manager; (ii) The Interference Handling Manager analyses the monitoring data from each VM and obtains the application metrics, and provides migration-based interference-aware scheduling and adaptive Dom0 reconfiguration; (iii) The Dom0 controller calls the corresponding APIs to trigger virtual machines migration or Dom0 reconfiguration based on the prediction results and the SLO targets. Authors have implemented CloudScope on top of the Xen Hypervisor and conducted experiments using a set of CPU, disk, and network-intensive workloads and a real system (MapReduce). The interference-aware scheduler improves virtual machine performance by up to 10% compared to the default scheduler, achieving an average error of 9%. Authors claim that the Hypervisor reconfiguration can improve network throughput by up to 30%.

Alves et al. [MTFD18] have developed an interference-aware virtual machine placement strategy for HPC applications in cloud computing. The authors' approach implements a method which predicts interference level in order to minimize the number of used physical machines. To solve this problem, a mathematical formulation and a strategy based on the Iterated Local Search framework were proposed. This framework defines the total interference by applying an average slowdown produced in all applications allocated to the same physical machine. The slowdown is defined as the percentage of additional time spent by the applications when they are executed concurrently in the same machine. To evaluate the proposed approach, the authors used two metrics: (i) first, they determined the percentage of test cases where the solution achieved a strictly smaller sum of interference levels than the one reached by the heuristic. Concerning the minimization of interference, this metric allows to quantify the number of cases where the solution outperformed the tested heuristic; (ii) second, it was calculated the percentage of cases where the solution used a smaller or equal number of physical machines than the one used by the heuristic. Results show the method reduced interference in more than 40%, using the same number of physical machines as the most widely employed heuristics.

To address latency-sensitive applications issues, such as QoS impact, and overcome limitations in existing offline approaches, Shekhar et al. [SAB$^+$18] 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. The performance of the entire system is monitored using a resource usage and performance interference statistics collection framework, called FECBench. The measurements include macro and micro resource metrics, such as CPU, memory, network I/O, disk I/O, context switches, page faults, cache, retired instructions per second (IPS), memory bandwidth, scheduler wait time and scheduler I/O wait time. These system performance metrics are calculated together with application workload and latency data, and passed on to the model predictor. The performance of the latency-sensitive application is predicted and forwards the information to a decision engine. The decision component then makes makes scheduled actions, which can be add/remove cores to the application and, remove/add cores, or checkpoint/restore for batch applications. A comparison with a representative latency-sensitive application reveals up to 39.46% lower tail latency than reactive approaches.

Wang et al. [WKNG19] developed data-driven analytical models to estimate the effect of interference among multiple Apache Spark jobs on job execution time in virtualized cloud environments. After, they present the design of an interference aware job scheduling algorithm leveraging the developed analytical framework. As different stages of a job are expected to have different characteristics in terms of resource utilization (e.g., CPU, I/O, memory), different stages of multiple jobs running con currently on a system are expected to result in different interference patterns, affecting the execution time differently. That is the reason each stage is represented as a vector consisting of execution time, CPU usage, disk I/O rate, and network I/O rate. The evaluation of model accuracy was measured using real-life applications on a 6 node cluster while running up to four jobs concurrently. Experiment results show that the scheduling algorithm reduces the average execution time of individual jobs and the total execution time significantly, and ranges between 47 and 26% for individual jobs and 2 to 13% for total execution time respectively.

6.1 Differences to our Proposal

When addressing interference-aware scheduling issues, most of the above mentioned related studies apply prediction models [CRO+15, NKG10, SAB+18, DK13, CH11, ZT12, BRX13, MTFD18, WKNG19], since they perform scheduling actions based on previously measured tasks. Also, many of them employ online scheduling approaches [CRO+15, NKG10, SAB+18, DK13, CH11, ZT12, BRX13, ZRWZ14], since they keep profiling their job metrics at runtime to improve scheduling decisions. Moreover, some of them [NKG10, SAB+18, DK13, ZT12] are concerned about meeting QoS requirements while improving the system efficiency.

In contrast to the related work cited above, we have utilized a dynamic and reactive system that analyzes the applications in real-time. Based on the individual interference they generate, we then run scheduling actions aimed at accomplishing a substantial improvement in hardware utilization.

Our work differs from related studies due to the evolution of technology and the update of operating systems and Kernels versions. Such evolution makes it possible to extract information from hardware in a manner that was not possible in the recent past. For example, the advanced feature developed by Intel, called Intel Cache Monitoring Technology, makes it now entirely viable to collect information about the usage of the cache by applications running inside any piece of equipment. This is essential, because multi-thread architectures are exponentially growing within the computer market. This technology allows us to use an ID denominated Resource Monitoring ID to metrify the number of threads scheduled among the operation system. For each thread, there is one ID associated with it. Therefore, these metrics can be collected within an MSR interface. This was not possible before the creation of this technology.

Q-Clouds [NKG10] requires at least one application execution to understand its behavior in order for the system to be able to manage this application. In our work, it is not necessary to learn how the applications behave, we only train our classification method with a mix of workloads once, then the new incoming applications are handled automatically. Another important aspect is that Q-Clouds focuses only on CPU bound to make VM placement decisions, while ours uses five main resources. TRACON [CH11] monitors only I/O and CPU hardware counters and tackles static and dynamic workloads. However, the dynamic workloads used follow the Poisson distribution process, unlike our approach that deals with real dynamic workload datasets. Zhu and Thung [ZT12] do not consider cache in their approach, while ours technique does.

Paragon [DK13] is an interference-aware scheduling approach which uses collaborative filtering to identify how well an incoming application will run on the different hardware platforms available. This approach analyzes ten different resources, focusing on heterogeneous data centers, while our architecture handles homogeneous clusters. Bu et al.'s [BRX13] study is limited to analyzing only CPU from Hypervisor through *xentop* counters and disk metrics through linux *iostat* from Hadoop workloads. Similar to [BRX13]'s work, Zhang et al. [ZRWZ14] propose ILA, where only CPU and disk counters are monitored from Hadoop applications.

Similar to our work, Chen et al. [CRO+15] includes in their Cloudscope strategy some different components to distribute systems' responsibilities. Also, the authors' approach profiles specific virtual machine characteristics, such as VCPUs and VNICs. The main difference from our architecture is that we run applications over containers instead of traditional virtual machines. As mentioned before, this kind of virtualization presents many benefits over the traditional method, such as low management overhead and portability [ZTL+19]. Alves et al.'s [MTFD18] work utilizes a slowdown factor that measures the applications' time and how much (percentage) each one has increased regarding isolated execution. Also, an average period is calculated for each host to compare them with each other. Our work on the other hand, applies interference levels instead of the raw percent of performance degradation. Moreover, we also use automatic techniques that produce the interference levels, with no user intervention.

Similar to our work, Shekhar et al. [SAB+18] investigate the interference generated by container-based instances with workload variations. However, our work is distinctive in the following ways: (i) instead of focusing on a single server, our proposed approach is performed over a distributed architecture (cluster). Furthermore, our proposed architecture focuses exclusively on applications that have dynamic workloads, such as latency-sensitive applications, while the others mix them with batch-jobs. Wang et al. [WKNG19] are interested in improving Apache Spark job makespans. Since this type of application is workflow-oriented, they present multiple job stages, and the authors analyze each stage separately, stating that each one has different (specific) resource behaviors. The authors consider only execution time, CPU usage, disk I/O rate, and network I/O rate, not observing cache and memory metrics, which our approach does.

In addition, it was possible to scale the problem out through the simulation. We developed an extension of a well-know simulator tool, namely ContainerCloudSim [PDCB17], to analyze performance interference aspects from applications. This strategy makes it possible to test the proposed architecture in an environment closer to a real world scenario, which had not been done in any related work we found.

7. CONCLUSIONS

Cloud computing has been attracting a great deal of attention from the IT community due to its promise of unlimited computing resource provisioning and the pay-per-use model. It can offer these benefits through the virtualization technique, allowing large data centers to dynamically take advantage of their infrastructure while reducing energy consumption, and the costs related to it. Cloud computing providers also apply resource scheduling policies to manage their hardware efficiently and improve applications' performance. Therefore, improving user satisfaction. However, multiple cloud-services contending for shared resources generate cross-application interference and this can lead to severe performance degradation. Evidence shows that interference is related to application performance penalty and it may occur depending on the application's workload and variation.

After searching for related-studies in the literature, this work evaluated how interference-aware resource scheduling has been covered recently and discovered that there are still research opportunities to explore regarding dynamic strategies and workload variation patterns. This dissertation's main goal is to determine whether dynamic interference-aware scheduling algorithms, which analyze workload variations of latency-sensitive applications over time, can further improve resource utilization.

Four research questions were defined to conduct this study and they were answered through the work. To begin solving RQ1, which aimed to discover if dynamic workloads modify their interference levels over time, chapter 3 presented how applications with those characteristics act under different circumstances. We introduced an interference analysis, highlighting how each hardware resource behaves when the workload varies, and its interference impact on response time. We concluded that each application demonstrates specific resource utilization characteristics, generating distinct interference rates. When observing specific hardware resources, applications may or may not generate interference indexes according to workload variability, enforcing the need for an in-depth interference profile analysis.

After better understanding the effects of workload variations on hardware resources, an interference classification technique [LXK+19] was adopted to validate if interference levels truly vary. This classification analyzes the interference levels over the entire application life process and assigns just one label per profiled resource. Since this technique applies a static approach, we created a segmented version of this classification in chapter 4 and ran some experiments with different workloads. These experiments answered RQ1, confirming that applications with dynamic workloads are susceptible to change their interference levels over time. Even though this strategy leads to better resource utilization, it still relies on a static interference classification technique. Therefore, moving into a dynamic scheduling environment, we developed an interference-aware application classifier by combining two machine learning techniques. This tool automatically classifies applications without user intervention or previous knowledge of application behavior. We then explored dynamic resource scheduling issues based on the interference profile through the proposal of an ML-driven classification scheme. Preliminary results revealed a 27% improvement in resource utilization efficiency, on average, when this classification approach was applied in cloud scenarios. We could therefore answer RQ2, confirming that a dynamic interference classification system, which represents better the workload variability over time, can lead to better resource scheduling.

Moving in the direction of dynamic scheduling, in chapter 5 we introduced IADA, a full-fledged interference-aware scheduling architecture for dynamic workloads in clouds. IADA combines and improves different techniques studied in previous work, including machine learning, bayesian algorithms, and heuristics to find abrupt changes in applications' workload behavior, classifying and placing them in a way that minimizes overall resource contention. We compared our solution with closely-related studies in this field using real workloads from NASA, Wikimedia, and Alibaba Open Cluster Trace datasets and results showed that IADA reduced the resulting performance degradation by 26% when compared to EVEN, CIAPA, and Segmented scheduling approaches in practical experiments, and by 24% in simulated experiments. These conclusions made it possible for us to answer RQ3, that it is necessary to develop some architectural modifications to move from static to dynamic interference-aware scheduling. To build IADA, we had to create a module that finds the right moments to analyze profiled data and coordinates with the entire system. Also, we had to develop some modifications in the scheduling algorithms to reduce the number of the migrations as much as possible. Since we are performing scheduling actions at run-time, such operations can generate more overhead than we are trying to prevent, if not controlled.

Moreover, an overhead analysis was also performed and presented under the Migration, Machine Learning, and Profiler techniques used by IADA. We concluded that: the scheduling algorithm was developed and optimized to reduce the number of migrations as much as possible. Our solution presented a reasonable numbers of scheduling actions per interval, keeping the general overhead at an acceptable rate; the chosen machine learning techniques generated a layer of overhead, but in our experiments, these indexes are considered acceptable. However depending on the number of nodes and applications the architecture controls, the resulting overhead could be bigger than ours. Consequently, the Node Manager might need to be resized. Yet, the profiler chosen (IntP) practically does not put any overhead pressure over the system, since this tool was built to analyze hardware events at the kernel layer. After highlighting all of these outcomes, RQ4 was answered. We confirmed that there are ways to implement modifications in the proposed architecture in a way that keeps the overhead to a controlled status, not invalidating the gains of the improved scheduling.

7.1 Contributions

The main contributions of this dissertation are as follows:

- Applications with dynamic workloads were shown to be susceptible to change their interference levels over time.
- An interference-aware application classifier based on machine learning techniques which sets interference levels ranges automatically, considering dynamic workloads, was proposed.
- Our classifier was found to have the potential to significantly improve application placement in consolidated environments.
- A resource scheduling architecture observing cross-application interference aspects for dynamic workloads was proposed. Unlike previous work, which has tackled partial issues, in this study, we present a full scheduling architecture solution targeting real production systems.
- An online bayesian changepoint detection (OBCD) algorithm that finds time-points automatically to perform classification and scheduling decisions. This specific topic was considered to be a gap in previous work [MKDD21b]. Since we did not know how to find the best moments to run classification and scheduling actions, we used a static-defined interval scheme to analyze our strategy. Therefore, we included an OBCD algorithm as a new feature in the proposed architecture to overcome this issue.
- An optimized version of a Simulated Annealing heuristic to tackle dynamic scheduling aspects was proposed. The original version, presented by [LXK^{+}19], was built upon static interference models, and to apply it in a dynamic scenario, we had to perform some modifications.
- An extension for the CloudSim toolkit was developed to execute interference-aware scheduling, using real case provisioning requirements and constraints, making it available in a GitHub repository to allow reproducibility.

7.2 Publications

The work presented in this dissertation has been partially or completely derived from a set of papers published/submitted during the doctoral period, listed as follows:

- **MEYER, V.** ; LUDWIG, U. L.; XAVIER, M. G. ; KIRCHOFF, D. F.; DE ROSE, C. A. F.. "Towards Interference-aware Dynamic Scheduling in Virtualized Environments", IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). Proceedings of the 23nd International Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP). New Orleans, USA, 2020, pp. 1-24. [MLX$^+$20].
- **MEYER, V.**; KIRCHOFF, D. F.; SILVA, M. L.; DE ROSE, C. A. F.. An "Interference-aware Application Classifier Based on Machine Learning to Improve Scheduling in Clouds", Proceedings of the 28th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP), Västerås, Sweden, 2020, pp. 80-87. [MKdD20].
- **MEYER, V.**; KIRCHOFF, D. F.; SILVA, M. L.; DE ROSE, C. A. F.. ML-driven "Classification Scheme for Dynamic Interference-aware Resource Scheduling in Cloud Infrastructures", Journal of Systems Architecture (JSA), v.116, p.102064, 2021. [MKDD21b].
- **MEYER, V.**; SILVA, M. L.; KIRCHOFF, D. F.; DE ROSE, C. A. F.. "IADA: A Dynamic Interference-aware Cloud Scheduling Architecture for Latency-sensitive Workloads", submitted for publication - Under Review.

In addition to the papers included in this work, the following publications arose from work conducted by the author during his doctoral studies:

- **MEYER, V.**; KRINDGES, R.; FERRETO, T. C.; DE ROSE, CESAR A.F.; HESSEL, F.. "Simulators Usage Analysis to Estimate Power Consumption in Cloud Computing Environments", Proceedings of the Symposium on High Performance Computing Systems (WSCAD), 2018. p. 70. São Paulo, Brazil. [MKF$^+$18].
- THAMSEN, L.; VERBITSKIY, I; NEDELKOSKI, S; TRAN, V. T.; **MEYER, V.**; XAVIER, M. G.; KAO, O.; DE ROSE, C. A. F.. "Hugo: A Cluster Scheduler that Efficiently Learns to Select Complementary Data-Parallel Jobs", Euro-Par 2019: European Conference on Parallel Processing Workshops (EUROPARW), 2019. v. 1. p. 519-530. Goettingen, Germany. [TVN$^+$20].
- **MEYER, V.**; XAVIER, M.; KIRCHOFF, D.; RIGHI, R.; DE ROSE, C. A. F.. "Performance and Cost Analysis between Elasticity Strategies over Pipeline-structured Applications", Proceedings of the 9th International Conference on Cloud Computing and Services Science (CLOSER), 2019. v. 1. p. 404-411. Heraklion, Greece. [MXK$^+$19].
- KRZYWDA, J.; **MEYER V.**; XAVIER, M.; ALI-ELDIN, A.; ÖSTBERG, P.; DE ROSE, C. A. F.; ELMROTH, E.. "Modeling and Simulation of QoS-Aware Power Budgeting in Cloud Data Centers", Proceedings of the 28th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP), Västerås, Sweden, 2020, pp. 88-93. [KMX$^+$20].

- SILVA, M. L.; **MEYER, V.**; KIRCHOFF, D. F.; SANTOS, J. ; VIEIRA, R.; DE ROSE, C. A. F.. "Evaluating the performance and improving the usability of parallel and distributed Word Embeddings tools", roceedings of the 28th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP), Västerås, Sweden, 2020, pp. 201-206. [DSMK$^+$20].
- SILVA, M. L.; **MEYER, V.**; DE ROSE, C. A. F.. "Edge Computing and its Boundaries to IoT and Industry 4.0: A Systematic Mapping Study", International Journal of Grid and Utility Computing (IJGUC), vol. xx, pp. xxx–xxx.
- KIRCHOFF, D. F.; **MEYER, V.**; CALHEIROS, Rodrigo N.; DE ROSE, C. A. F.. "Evaluating machine learning prediction techniques and their impact on proactive resource provisioning for cloud environments", submitted for publication - Under Review.

7.3 Future Directions

There are several possible directions for future research based on this work. We suggest some possibilities for future research as follows.

7.3.1 Proactive Scheduling Techniques

We presented a solution to perform dynamic scheduling decision based on a reactive approach. However, there are several works presenting the benefits of applying proactive techniques in order to improve resource rearrangement actions. As a possibility, we expect to evaluate proactive scheduling approaches by applying machine learning prediction algorithms and comparing them with the current work. The goal is to analyze how much the performance degradation can be reduced by forecasting the workload variability and anticipating the arrangement of hardware resources within a dynamic scheduling architecture.

7.3.2 Interference Classification

In the interference classification strategy, we adopted four interference levels (absent, low, med, high) to execute scheduling actions. The decision of using this interference subdivision was based on [LXK$^+$19] work. However, are four interference levels enough to improve resource scheduling? Why not to use more levels? How many levels should be used? To solve these answers, we intend to deeply explore the classification process by creating more ranges of interference levels to analyze how they can affect scheduling decisions and consequently applications performance.

7.3.3 Cloud Elasticity

Cloud elasticity or scalability in cloud computing refers to the ability to increase or decrease computational resources as needed to meet changing demands. Scalability is one of the hallmarks of the cloud and the primary driver of its exploding popularity with businesses. Since our approach did not handle scaling issues yet, as an interesting future direction, we plan to explore horizontal and vertical cloud elasticity techniques to improve the use of our architecture in real scenarios even more.

7.3.4 Scheduling Heuristics

The proposed architecture operates at a level above certain decisions of the infrastructure, such as which containers and where to migrate them. This doctoral dissertation used a heuristic known as Simulated Annealing (SA), one of the heuristics presented by Ludwig et al. [LXK+19], which aims to optimize applications performance, observing interference issues. Other heuristics that decide in what ways container should be migrated can be tested along with our solution, and it can provide even better results of resource utilization, applications performance, and energy expenses reduction.

7.3.5 Power Optimization

Power optimization techniques are becoming increasingly important in cloud computing system design. Virtualization allows the deployment of co-existing computing environments over the same hardware infrastructure in cloud ecosystems. However, the co-existing environments often create scenarios that lead to performance degradation. This issue, known as Performance Interference, introduces a non-negligible overhead that affects both a data center’s Quality of Service and its energy efficiency. In future work, we expect to combine power-aware or energy-oriented scheduling algorithms with our solution to reduce the impact of performance interference on energy efficiency.

REFERENCES

[AdAD17] Alves, M. M.; de Assumpção Drummond, L. M. “A multivariate and quantitative model for predicting cross-application interference in virtual environments”, *Journal of Systems and Software*, vol. 128, Apr 2017, pp. 150–163.

[AETEK13] Ali-Eldin, A.; Tordsson, J.; Elmroth, E.; Kihl, M. “Workload classification for efficient auto-scaling of cloud resources”, Technical Report, Umeå University, Department of Computing Science, 2013, 36p.

[AHK17] Athmaja, S.; Hanumanthappa, M.; Kavitha, V. “A survey of machine learning algorithms for big data analytics”. In: International Conference on Innovations in Information, Embedded and Communication Systems, 2017, pp. 1–4.

[AKF15] Amannejad, Y.; Krishnamurthy, D.; Far, B. “Detecting performance interference in cloud-based web services”. In: IFIP/IEEE International Symposium on Integrated Network Management, 2015, pp. 423–431.

[AL17] Alipour, H.; Liu, Y. “Online machine learning for cloud resource provisioning of microservice backend systems”. In: IEEE International Conference on Big Data, 2017, pp. 2433–2441.

[ASZ20] Al-Sinayyid, A.; Zhu, M. “Job scheduler for streaming applications in heterogeneous distributed processing systems”, *The Journal of Supercomputing*, vol. 76, May 2020, pp. 9609–9628.

[AW10] Abdi, H.; Williams, L. J. “Principal component analysis”, *WIREs Computational Statistics*, vol. 2–4, Jul 2010, pp. 433–459.

[BB12] Beloglazov, A.; Buyya, R. “Optimal online deterministic algorithms and adaptive heuristics for energy and performance efficient dynamic consolidation of virtual machines in cloud data centers”, *Concurrency and Computation: Practice and Experience*, vol. 24–13, Sep 2012, pp. 1397–1420.

[Bro04] Broadwell, P. M. “Response time as a performability metric for online services”, Technical Report, University of California, 2004, 49p.

[BRX13] Bu, X.; Rao, J.; Xu, C.-z. “Interference and locality-aware task scheduling for mapreduce applications in virtual clusters”. In: 22nd International Symposium on High-performance Parallel and Distributed Computing, 2013, pp. 227–238.

[BSB+99] Braun, T. D.; Siegal, H. J.; Beck, N.; Boloni, L. L.; Maheswaran, M.; Reuther, A. I.; Robertson, J. P.; Theys, M. D.; Bin Yao; Hensgen, D.; Freund, R. F. “A comparison study of static mapping heuristics for a class of meta-tasks on heterogeneous computing systems”. In: 8th Heterogeneous Computing Workshop, 1999, pp. 15–29.

[Cas01] Casanova, H. “Simgrid: a toolkit for the simulation of application scheduling”. In: 1st IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, 2001, pp. 430–437.

[CGD+17] Chen, S.; GalOn, S.; Delimitrou, C.; Manne, S.; Martínez, J. F. “Workload characterization of interactive cloud services on big and small server platforms”. In: IEEE International Symposium on Workload Characterization (IISWC), 2017, pp. 125–134.

[CH11] Chiang, R. C.; Huang, H. H. “Tracon: Interference-aware scheduling for data-intensive applications in virtualized environments”. In: International Conference for High Performance Computing, Networking, Storage and Analysis, 2011, pp. 47:1–47:12.

[CNLC13] Castañé, G. G.; Núñez, A.; Llopis, P.; Carretero, J. “E-mc2: A formal framework for energy modelling in cloud computing”, *Simulation Modelling Practice and Theory*, vol. 39, Jun 2013, pp. 56–75, s.I.Energy efficiency in grids and clouds.

[CRB+11] Calheiros, R. N.; Ranjan, R.; Beloglazov, A.; De Rose, C. A. F.; Buyya, R. “Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms”, *Software: Practice and Experience*, vol. 41–1, Aug 2011, pp. 23–50, https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.995.

[CRO+15] Chen, X.; Rupprecht, L.; Osman, R.; Pietzuch, P.; Franciosi, F.; Knottenbelt, W. “Cloudscope: Diagnosing and managing performance interference in multi-tenant clouds”. In: 23rd IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 2015, pp. 164–173.

[CSG14] Caglar, F.; Shekhar, S.; Gokhale, A. S. “Towards a performance interference-aware virtual machine placement strategy for supporting soft real-time applications in the cloud”. In: 3rd IEEE International Workshop on Real-time and distributed computing in emerging applications, 2014, pp. 15–20.

[CSGK16] Caglar, F.; Shekhar, S.; Gokhale, A.; Koutsoukos, X. “Intelligent, performance interference-aware resource management for iot cloud backends”. In:

IEEE First International Conference on Internet-of-Things Design and Implementation, 2016, pp. 95–105.

[CSK20] Chhabra, A.; Singh, G.; Kahlon, K. S. “Multi-criteria hpc task scheduling on iaas cloud infrastructures using meta-heuristics”, *Cluster Computing*, vol. 24, Jun 2020, pp. 885–918.

[CW20] Corporation, M.; Weston, S. “doparallel: Foreach parallel adaptor for the ’parallel’ package”. R package version 1.0.16, Source: https://CRAN.R-project.org/package=doParallel, 01 dec 2021.

[DK13] Delimitrou, C.; Kozyrakis, C. “Paragon: Qos-aware scheduling for heterogeneous datacenters”, *SIGPLAN Notices*, vol. 48–4, Mar 2013, pp. 77–88.

[DKCS18] Devarajan, H.; Kougkas, A.; Challa, P.; Sun, X. “Vidya: Performing code-block i/o characterization for data access optimization”. In: 25th IEEE International Conference on High Performance Computing, 2018, pp. 255–264.

[DLCO19] Duc, T. L.; Leiva, R. G.; Casari, P.; Östberg, P.-O. “Machine learning methods for reliable resource provisioning in edge-cloud computing: A survey”, *ACM Computing Surveys*, vol. 52–5, Sep 2019, pp. 94:1–94:39.

[DSMK$^+$20] Da Silva, M. L.; Meyer, V.; Kirchoff, D. F.; Santos, J.; Vieira, R.; De Rose, C. A. F. “Evaluating the performance and improving the usability of parallel and distributed word embeddings tools”. In: 28th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, 2020, pp. 201–206.

[ENBH13] Eklov, D.; Nikoleris, N.; Black-Schaffer, D.; Hagersten, E. “Bandwidth bandit: Quantitative characterization of memory contention”. In: International Symposium on Code Generation and Optimization, 2013, pp. 1–10.

[FHOM09] Ferri, C.; Hernández-Orallo, J.; Modroiu, R. “An experimental comparison of performance measures for classification”, *Pattern Recognition Letters*, vol. 30–1, Sep 2009, pp. 27–38.

[GdSTd20] Guevara, J. C.; da S. Torres, R.; da Fonseca, N. L. “On the classification of fog computing applications: A machine learning perspective”, *Journal of Network and Computer Applications*, vol. 159, Mar 2020, pp. 102596.

[GGS$^+$19] Gill, S. S.; Garraghan, P.; Stankovski, V.; Casale, G.; Thulasiram, R. K.; Ghosh, S. K.; Ramamohanarao, K.; Buyya, R. “Holistic resource management for sustainable and reliable cloud computing: An innovative solution to global challenge", *Journal of Systems and Software*, vol. 155, May 2019, pp. 104–129.

[GMC$^+$13] Guérout, T.; Monteil, T.; Costa, G. D.; Calheiros, R. N.; Buyya, R.; Alexandru, M. "Energy-aware simulation with DVFS", *Simulation Modelling Practice and Theory*, vol. 39, Jun 2013, pp. 76–91.

[GTGB14] Garg, S. K.; Toosi, A. N.; Gopalaiyengar, S. K.; Buyya, R. "Sla-based virtual machine management for heterogeneous workloads in a cloud datacenter", *Journal of Network and Computer Applications*, vol. 45, Aug 2014, pp. 108–120.

[HVA$^+$16] Herdrich, A.; Verplanke, E.; Autee, P.; Illikkal, R.; Gianos, C.; Singhal, R.; Iyer, R. "Cache qos: From concept to reality in the intel® xeon® processor e5-2600 v3 product family". In: IEEE International Symposium on High Performance Computer Architecture, 2016, pp. 657–668.

[HZdLZ20] Hu, Y.; Zhou, H.; de Laat, C.; Zhao, Z. "Concurrent container scheduling on heterogeneous clusters with multi-resource constraints", *Future Generation Computer Systems*, vol. 102, Sep 2020, pp. 562–573.

[IAK$^+$18] Iorgulescu, C.; Azimi, R.; Kwon, Y.; Elnikety, S.; Syamala, M.; Narasayya, V.; Herodotou, H.; Tomita, P.; Chen, A.; Zhang, J.; Wang, J. "Perfiso: Performance isolation for commercial latency-sensitive services". In: USENIX Annual Technical Conference, 2018, pp. 519–532.

[IEM18] Iqbal, W.; Erradi, A.; Mahmood, A. "Dynamic workload patterns prediction for proactive auto-scaling of web applications", *Journal of Network and Computer Applications*, vol. 124, Oct 2018, pp. 94–107.

[IH12] Issariyakul, T.; Hossain, E. "Introduction to Network Simulator 2 (NS2)". Boston, MA: Springer US, 2012, chap. Introduction to Network Simulator 2 (NS2), pp. 21–40.

[IK77] Ibarra, O. H.; Kim, C. E. "Heuristic algorithms for scheduling independent tasks on nonidentical processors", *Journal of the ACM*, vol. 24–2, Apr 1977, pp. 280–289.

[JF16] Jersak, L. C.; Ferreto, T. "Performance-aware server consolidation with adjustable interference levels". In: 31st Annual ACM Symposium on Applied Computing, 2016, pp. 420–425.

[JG17] Javadi, S. A.; Gandhi, A. "Dial: Reducing tail latencies for cloud applications via dynamic interference-aware load balancing". In: IEEE International Conference on Autonomic Computing, 2017, pp. 135–144.

[KBK12] Kliazovich, D.; Bouvry, P.; Khan, S. U. “Greencloud: a packet-level simulator of energy-aware cloud computing data centers”, *The Journal of Supercomputing*, vol. 62, Dec 2012, pp. 1263–1283.

[KDSL18] Kougkas, A.; Devarajan, H.; Sun, X.; Lofstead, J. “Harmonia: An interference-aware dynamic i/o scheduler for shared non-volatile burst buffers”. In: IEEE International Conference on Cluster Computing, 2018, pp. 290–301.

[KGV83] Kirkpatrick, S.; Gelatt, C. D.; Vecchi, M. P. “Optimization by simulated annealing”, *Science*, vol. 220–4598, May 1983, pp. 671–680.

[KMX$^+$20] Krzywda, J.; Meyer, V.; Xavier, M.; Ali-eldin, A.; Östberg, P.; De Rose, C. A. F.; Elmroth, E. “Modeling and simulation of qos-aware power budgeting in cloud data centers”. In: 28th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, 2020, pp. 88–93.

[KS17] Kumar, R.; Setia, S. “Interface aware scheduling of tasks on cloud”. In: 4th International Conference on Signal Processing, Computing and Control, 2017, pp. 654–658.

[LKRH15] Landset, S.; Khoshgoftaar, T. M.; Richter, A. N.; Hasanin, T. “A survey of open source tools for machine learning with big data in the hadoop ecosystem”, *Journal of Big Data*, vol. 2–1, Nov 2015, pp. 24.

[LSN$^+$09] Lim, S.; Sharma, B.; Nam, G.; Kim, E. K.; Das, C. R. “Mdcsim: A multi-tier data center simulation, platform”. In: IEEE International Conference on Cluster Computing and Workshops, 2009, pp. 1–9.

[LW02] Liaw, A.; Wiener, M. “Classification and regression by randomforest”, *R News*, vol. 2–3, Jan 2002, pp. 18–22.

[LXK$^+$19] Ludwig, U. L.; Xavier, M. G.; Kirchoff, D. F.; Cezar, I. B.; De Rose, C. A. F. “Optimizing multi-tier application performance with interference and affinity-aware placement algorithms”, *Concurrency and Computation: Practice and Experience*, vol. 31–18, Sep 2019, pp. e5098, e5098 cpe.5098.

[MA19] Maniar, M. A.; Abhyankar, A. R. “Validity index based improvisation in reproducibility of load profiling outcome”, *IET Smart Grid*, vol. 2–1, Mar 2019, pp. 131–139.

[MDH$^+$19] Meyer, D.; Dimitriadou, E.; Hornik, K.; Weingessel, A.; Leisch, F. “e1071: Misc functions of the department of statistics, probability theory group”. R package version 1.7-2, Source: https://CRAN.R-project.org/package=e1071, 01 dec 2021.

[Mer14] Merkel, D. “Docker: Lightweight linux containers for consistent development and deployment”, *Linux Journal*, vol. 2014–239, Mar 2014, pp. 2.

[MGXD18] Matteussi, K. J.; Geyer, C. F. R.; Xavier, M. G.; De Rose, C. A. F. “Understanding and minimizing disk contention effects for data-intensive processing in virtualized systems”. In: International Conference on High Performance Computing Simulation, 2018, pp. 901–908.

[MK19] Manohar, S. S.; Kapoor, H. K. “Dynamic reconfiguration of embedded-dram caches employing zero data detection based refresh optimisation”, *Journal of Systems Architecture*, vol. 100, Oct 2019, pp. 101648.

[MKdD20] Meyer, V.; Kirchoff, D. F.; da Silva, M. L.; De Rose, C. A. F. “An interference-aware application classifier based on machine learning to improve scheduling in clouds”. In: 28th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, 2020, pp. 80–87.

[MKdD21a] Meyer, V.; Kirchoff, D. F.; da Silva, M. L.; De Rose, C. A. F. “Interference-aware application classifier for dynamic scheduling in cloud infrastructures”. Source: http://dx.doi.org/10.24433/CO.3183391.v1, 01 dec 2021.

[MKDD21b] Meyer, V.; Kirchoff, D. F.; Da Silva, M. L.; De Rose, C. A. F. “Ml-driven classification scheme for dynamic interference-aware resource scheduling in cloud infrastructures”, *Journal of Systems Architecture*, vol. 116, Feb 2021, pp. 102064.

[MKF$^+$18] Meyer, V.; Krindges, R.; Ferreto, T. C.; De Rose, C. A. F.; Hessel, F. “Simulators usage analysis to estimate power consumption in cloud computing environments”. In: Symposium on High Performance Computing Systems, 2018, pp. 70–76.

[MLX$^+$20] Meyer, V.; Ludwig, U. L.; Xavier, M. G.; Kirchoff, D. F.; De Rose, C. A. F. “Towards interference-aware dynamic scheduling in virtualized environments”. In: Job Scheduling Strategies for Parallel Processing, 2020, pp. 1–24.

[MMRB20] Morariu, C.; Morariu, O.; Răileanu, S.; Borangiu, T. “Machine learning for predictive scheduling and resource allocation in large scale manufacturing systems”, *Computers in Industry*, vol. 120, May 2020, pp. 103244.

[MTFD18] Melo Alves, M.; Teylo, L.; Frota, Y.; Drummond, L. M. A. “An interference-aware virtual machine placement strategy for high performance computing applications in clouds”. In: Symposium on High Performance Computing Systems, 2018, pp. 94–100.

[MXK$^+$19] Meyer, V.; Xavier, M. G.; Kirchoff, D. F.; da R. Righi, R.; De Rose, C. A. F. “Performance and cost analysis between elasticity strategies over pipeline-structured applications”. In: International Conference on Cloud Computing and Services Science, 2019, pp. 404–411.

[MYXW13] Moreno, I. S.; Yang, R.; Xu, J.; Wo, T. “Improved energy-efficiency in cloud datacenters with interference-aware virtual machine placement”. In: IEEE Eleventh International Symposium on Autonomous Decentralized Systems (ISADS), 2013, pp. 1–8.

[NKG10] Nathuji, R.; Kansal, A.; Ghaffarkhah, A. “Q-clouds: Managing performance interference effects for qos-aware clouds”. In: 5th European Conference on Computer Systems, 2010, pp. 237–250.

[NLC$^+$12] Núñez, A.; L., V.-P. J.; Caminero, A. C.; Castañé, G. G.; Carretero, J.; Llorente, I. M. “icancloud: A flexible and scalable cloud infrastructure simulator”, *The Journal of Supercomputing*, vol. 10, Mar 2012, pp. 185–209.

[Pag19] Pagotto, A. “ocp: Bayesian online changepoint detection”. R package version 0.1.1, Source: https://CRAN.R-project.org/package=ocp, 01 dec 2021.

[PBSJ19] Pahl, C.; Brogi, A.; Soldani, J.; Jamshidi, P. “Cloud container technologies: A state-of-the-art review”, *IEEE Transactions on Cloud Computing*, vol. 7–3, Jul 2019, pp. 677–692.

[PDCB17] Piraghaj, S. F.; Dastjerdi, A. V.; Calheiros, R. N.; Buyya, R. “Containercloudsim: An environment for modeling and simulation of containers in cloud data centers”, *Software: Practice and Experience*, vol. 47–4, Jun 2017, pp. 505–521.

[PKK19] Priya, V.; Kumar, C. S.; Kannan, R. “Resource scheduling algorithm with load balancing for cloud service provisioning”, *Applied Soft Computing*, vol. 76, Dec 2019, pp. 416–424.

[PL15] Pahl, C.; Lee, B. “Containers and clusters for edge cloud architectures – a technology review”. In: 3rd International Conference on Future Internet of Things and Cloud, 2015, pp. 379–386.

[Pot03] Potter, K. H. “Dynamic addressing mapping to eliminate memory resource contention in a symmetric multiprocessor system”. US Patent 6,505,269, Source: https://patents.google.com/patent/US6505269, 01 dec 2021.

[R C19] R Core Team. “R: A language and environment for statistical computing”. Source: https://www.R-project.org/, 01 dec 2021.

[Ros14] Rosen, R. “Linux containers and the future cloud”. Source: https://www.linuxjournal.com/content/linux-containers-and-future-cloud, 01 dec 2021.

[SAB$^+$18] Shekhar, S.; Abdel-Aziz, H.; Bhattacharjee, A.; Gokhale, A.; Koutsoukos, X. “Performance interference-aware vertical elasticity for cloud-hosted latency-sensitive applications”. In: 11th IEEE International Conference on Cloud Computing, 2018, pp. 82–89.

[Sav19] Savoie, L. “Inter-job optimization in high performance computing”, Ph.D. Thesis, The University of Arizona., 2019, 110p.

[SBB18] Sotiriadis, S.; Bessis, N.; Buyya, R. “Self managed virtual machine scheduling in cloud systems”, *Information Sciences*, vol. 433-434, Jul 2018, pp. 381–400.

[SBP15] Sampaio, A. M.; Barbosa, J. G.; Prodan, R. “Piasa: A power and interference aware resource management strategy for heterogeneous workloads in cloud data centers”, *Simulation Modelling Practice and Theory*, vol. 57, Jul 2015, pp. 142–160.

[Sch14] Scheepers, M. J. “Virtualization and containerization of application infrastructure : A comparison”. In: 21st Twente Student Conference on IT, 2014, pp. 1–7.

[SCSCC19] Sant’ana, L.; Carastan-Santos, D.; Cordeiro, D.; Camargo, R. D. “Real-time scheduling policy selection from queue and machine states”. In: 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, 2019, pp. 381–390.

[SWZV13] Shah, A.; Wolf, F.; Zhumatiy, S.; Voevodin, V. “Capturing inter-application interference on clusters”. In: IEEE International Conference on Cluster Computing, 2013, pp. 1–5.

[TQdAB17] Toosi, A. N.; Qu, C.; de Assunção, M. D.; Buyya, R. “Renewable-aware geographical load balancing of web applications for sustainable data centers”, *Journal of Network and Computer Applications*, vol. 83, Feb 2017, pp. 155–168.

[TRA15] Tosatto, A.; Ruiu, P.; Attanasio, A. “Container-based orchestration in cloud: State of the art and challenges”. In: 9th International Conference on Complex, Intelligent, and Software Intensive Systems, 2015, pp. 70–75.

[TVN$^+$20] Thamsen, L.; Verbitskiy, I.; Nedelkoski, S.; Tran, V. T.; Meyer, V.; Xavier, M. G.; Kao, O.; De Rose, C. A. F. “Hugo: A cluster scheduler that efficiently learns to select complementary data-parallel jobs”. In: Euro-Par: Parallel Processing Workshops, 2020, pp. 519–530.

[USR03] Urgaonkar, B.; Shenoy, P.; Roscoe, T. “Resource overbooking and application profiling in shared hosting platforms”, *ACM SIGOPS Operating Systems Review*, vol. 36–SI, Dec 2003, pp. 239–254.

[WKNG19] Wang, K.; Khan, M. M. H.; Nguyen, N.; Gokhale, S. “Design and implementation of an analytical framework for interference aware job scheduling on apache spark platform”, *Cluster Computing*, vol. 22–1, Jan 2019, pp. 2223–2237.

[Xav19] Xavier, M. G. “Data processing with cross-application interference control via system-level instrumentation”, Ph.D. Thesis, Pontifical Catholic University of Rio Grande do Sul, 2019, 100p.

[XMLDR16] Xavier, M. G.; Matteussi, K. J.; Lorenzo, F.; De Rose, C. A. F. “Understanding performance interference in multi-tenant cloud databases and web applications”. In: IEEE International Conference on Big Data, 2016, pp. 2847–2852.

[XNR$^+$13] Xavier, M. G.; Neves, M. V.; Rossi, F. D.; Ferreto, T. C.; Lange, T.; De Rose, C. A. F. “Performance evaluation of container-based virtualization for high performance computing environments”. In: 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, 2013, pp. 233–240.

[XRR$^+$17] Xavier, M. G.; Rossi, F. D.; Rose, C. A. F. D.; Calheiros., R. N.; Gomes, D. G. “Modeling and simulation of global and sleep states in ACPI-compliant energy-efficient cloud environments”, *Concurrency and Computation: Practice and Experience*, vol. 29–4, May 2017, pp. e3839, e3839 cpe.3839.

[XWHW19] Xu, M.; Wu, C. Q.; Hou, A.; Wang, Y. “Intelligent scheduling for parallel jobs in big data processing systems”. In: International Conference on Computing, Networking and Communications, 2019, pp. 22–28.

[Yag02] Yaghmour, K. “Linux trace toolkit project page.” Source: https://www.opersys.com/LTT/, 01 dec 2021.

[ZAL19] Zhou, Z.; Abawajy, J. H.; Li, F. “Analysis of Energy Consumption Model in Cloud Computing Environments”. Cham: Springer International Publishing, 2019, chap. 1, pp. 195–215.

[ZBF10] Zhuravlev, S.; Blagodurov, S.; Fedorova, A. “Addressing shared resource contention in multicore processors via scheduling”. In: ACM SIGARCH Computer Architecture News, 2010, pp. 129–142.

[ZCB10] Zhang, Q.; Cheng, L.; Boutaba, R. “Cloud computing: state-of-the-art and research challenges”, *Journal of Internet Services and Applications*, vol. 1–1, May 2010, pp. 7–18.

[ZCG$^+$18] Zhao, Y.; Calheiros, R.; Gange, G.; Bailey, J.; Sinnott, R. “Sla-based profit optimization resource scheduling for big data analytics-as-a-service platforms in cloud computing environments”, *IEEE Transactions on Cloud Computing*, vol. 9–3, Dec 2018, pp. 1236–1253.

[ZCV$^+$19] Zhao, Y.; Calheiros, R. N.; Vasilakos, A. V.; Bailey, J.; Sinnott, R. O. “Profit maximization and time minimization admission control and resource scheduling for cloud-based big data analytics-as-a-service platforms”. In: Web Services, 2019, pp. 26–47.

[ZLT$^+$16] Zhou, H.; Li, Q.; Tong, W.; Kausar, S.; Zhu, H. “P-aware: a proportional multi-resource scheduling strategy in cloud data center”, *Cluster Computing*, vol. 19, Sep 2016, pp. 1089–1103.

[ZRWZ14] Zhang, W.; Rajasekaran, S.; Wood, T.; Zhu, M. “Mimp: Deadline and interference aware scheduling of hadoop virtual machines”. In: 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, 2014, pp. 394–403.

[ZT12] Zhu, Q.; Tung, T. “A performance interference model for managing consolidated workloads in qos-aware clouds”. In: 15th IEEE International Conference on Cloud Computing, 2012, pp. 170–179.

[ZTL$^+$19] Zhang, F.; Tang, X.; Li, X.; Khan, S. U.; Li, Z. “Quantifying cloud elasticity with container-based autoscaling”, *Future Generation Computer Systems*, vol. 98, Nov 2019, pp. 672–681.

[ZWWZ11] Zhang, W.; Wang, S.; Wang, W.; Zhong, H. “Bench4q: A qos-oriented e-commerce benchmark”. In: 35th IEEE Annual Computer Software and Applications Conference, 2011, pp. 38–47.