PUCRS

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

MIGUEL GOMES XAVIER

DATA PROCESSING WITH CROSS-APPLICATION INTERFERENCE CONTROL VIA SYSTEM-LEVEL INSTRUMENTATION

Porto Alegre
2019

PÓS-GRADUAÇÃO - *STRICTO SENSU*

Pontifícia Universidade Católica
do Rio Grande do Sul

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

DATA PROCESSING WITH CROSS-APPLICATION INTERFERENCE CONTROL VIA SYSTEM-LEVEL INSTRUMENTATION

MIGUEL GOMES XAVIER

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 F. De Rose

Porto Alegre
2019

Ficha Catalográfica

X3d Xavier, Miguel Gomes

Data Processing with Cross-application Interference Control via System-level Instrumentation / Miguel Gomes Xavier. – 2019.
96.
Tese (Doutorado) – Programa de Pós-Graduação em Ciência da Computação, PUCRS.

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

1. Big Data. 2. Resource Management. 3. Virtualization. 4. Operating System. 5. High Performance 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: Clarissa Jesinska Selbach CRB-10/2051

MIGUEL GOMES XAVIER

DATA PROCESSING WITH CROSS-APPLICATION INTERFERENCE CONTROL VIA SYSTEM-LEVEL INSTRUMENTATION

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 Janeiro 08, 2019.

COMMITTEE MEMBERS:

Prof. Dr. Avaliador Antonio Tadeu A. Gomes (LNCC)

Prof. Dr. Avaliador Marco Aurelio S. Netto (IBM Research Brazil Lab/IBM)

Prof. Dr. Avaliador Avelino F. Zorzo (PPGCC/PUCRS)

Prof. César Augusto F. De Rose (PPGCC/PUCRS - Advisor)

Dedico este trabalho aos meus pais.

PROCESSAMENTO DE DADOS COM CONTROLE DE INTERFERÊNCIA ENTRE APLICATIVOS POR MEIO DE INSTRUMENTAÇÃO NO NÍVEL DO SISTEMA OPERACIONAL

RESUMO

O volume de dados na rede global está atingindo uma escala sem precedentes exigindo mudanças tecnológicas em diferentes espectros da computação para lidar com a crescente necessidade de desempenho. Embora as complexidades dos dados tenham aumentado, o impacto real depende da capacidade de extração e transformação desses conjuntos maciços de dados brutos e variados para extrair informações valiosas. Obter informações sobre esses dados derivou um amplo espectro para análise de Big Data. A análise de dados representou um grande desafio ao projetar sistemas de gerenciamento de recursos altamente escaláveis para integrar, extrair e transformar dados brutos em informações, mantendo a experiência dos usuários e as expectativas dos negócios. Os sistemas de gerenciamento de recursos para Big Data geralmente consolidam aplicativos e usam virtualização em nível de sistema operacional (contêineres) para permitir o compartilhamento de recursos e melhorar a eficiência. No entanto, o desempenho ainda varia imprevisivelmente devido à competição no acesso a recursos compartilhados como CPU, memória, disco e rede.

A intuição inicial que motiva o desenvolvimento desse trabalho é capacidade dos processadores modernos de disponibilizar informações que possam ser usadas para classificar a interferência emanada de aplicativos em contêiner. Portanto, conjeturamos que os clusters que interpresetam esses dados podem acelerar as aplicações no processo de análise de Big Data e melhorar a eficiência de recursos. Para confirmar nossa tese, primeiro estudamos as necessidades de desempenho de Big Data e os pontos fracos existentes no isolamento de desempenho de contêineres. Obtendo informações desses estudos para propor uma colocação de contêiner com reconhecimento de interferência, reunimos tudo

isso em um protótipo de planejador com reconhecimento de interferência, que resultou em ganhos de até 35% no desempenho da programação e 42% na eficiência dos recursos, portanto, confirmando a tese.

Palavras-Chave: Big Data, gerenciamento de recursos, virtualização, sistemas operacionais, computação de alto desempenho.

DATA PROCESSING WITH CROSS-APPLICATION INTERFERENCE CONTROL VIA SYSTEM-LEVEL INSTRUMENTATION

ABSTRACT

World's gigantic data collection is reaching a crucial point for significant technological changes to deal with the immense variety and performance needs. While the complexities of data have been increasing, the real impact depends on the ability of extraction and transformation of these massive and varied raw data sets to uncover valuable information. Gaining insights into this information has led to the area of Big Data analytics. Data analysis has represented a major challenge in designing highly scalable resource management systems to integrate, extract and transform data into information, while maintaining users' experience and business' expectation. Resource management systems for Big Data generally consolidate applications and use system-level virtualization (containers) to enable resource sharing and improve efficiency, but performance still vary unpredictably due to the competition in access to shared resources like CPU, memory, disk and network.

The initial intuition motivating our work is that the system-level information availability could be used to classify the interference emanate from containerized applications. We therefore conjecture that interference-aware clusters may speed up applications to accelerate Big Data analytics and improve resource-efficiency, while maintaining users' experiences and business' expectations. To confirm our thesis, we first studied Big Data performance needs and existing container system performance isolation weaknesses. Gaining insight from these studies to propose an container placement with interference recognition, we put all these together in an interference-aware scheduler prototype, which resulted in gains of up to 35% in scheduling performance and 42% in resource efficiency, thus, confirming the thesis.

Keywords: Big Data, resource management, virtualization, operating systems, high performance computing.

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

HPC – High Performance Computing

RMS – Resource Management System

CMS – Container Management System

OS – Operating System

QOS – Quality of Service

LLC – Last Level Cache

DC – Large Data Center

PM – Physical Machine

RRD – Round Robin Decreasing

SA – Simulated Annealing

SHC – Stochastic Hill Climb

CONTENTS

1. INTRODUCTION

Recent advances in cloud computing, social networks, and devices with sensory capabilities have shaped the so-called "Internet of Things" (IoT). This abundance of information has attracted a great deal of attention over the past few years and has led organizations to find out ways to handle this explosion of data sets in an efficient, reliable, and cost-effective fashion. MapReduce has become a prevalent programming model for large-scale data analysis. Diverse frameworks, such as Hadoop [106], have been proposed to simplify the concepts involved in large-scale distributed computing and make programming, scalability and fault tolerance easier. Hadoop-MapReduce is the most popular and widely deployed of the open-source MapReduce implementations. It was first used by Yahoo [106] and since then has hardened in production by many Internet companies including Facebook [13], Twitter [27, 99], and IBM [123].

As the popularity of large-scale data analysis increases, the emergence of purpose-built frameworks and programming models beyond MapReduce continues to grow. Tez [88], Spark [116], Storm [99], Flink [16], and Samza [105] complement the "Big Data" ecosystem and have been designed to allow data analysis that did not perform well with MapReduce, which is limited to acyclical executions [19]. Throughout the data analysis process, nothing prevents the same or different data from being simultaneously accessed by different frameworks. One organization puts all of its valuable data into the computing cluster, and there is a real need to process that insightful information with different frameworks in multiple ways such as human-interactive SQL queries by Spark, real-time event processing by Storm, batch processing by Hadoop-MapReduce, machine learning by TensorFlow [1], and so on. This requires a framework-agnostic resource sharing platform to enable data analysis not only for Hadoop-driven jobs, but for the entire Big Data ecosystem comprised of various frameworks. To cope with this, YARN [102], Mesos [42], Kubernetes [9], and so on, were developed, which are the next-generation Container Management Systems (CMS) that enable a fine-grained resource sharing for executing and managing the "Big Data" ecosystem in large data centers (DCs).

DCs have been striving for increasing resource utilization, while maintaining infrastructure and energy costs. Sharing cluster nodes among multiple applications (collocation) has become a de-facto strategy to achieve resource- and energy-efficiency in large computing infrastructures. However, this placement could lead applications to dispute access to compute resources, such as CPU, memory, disk, and network, making performance vary unpredictably, i.e., co-located applications suffer mutual performance interference. Avoiding unpredictable performance variation is a major responsibility for heterogeneous data centers (housing mixed-workload patterns) to give users the expected Quality of Service (QoS). When a resource-allocation mechanism assigns multiple tasks to CPU cores, it could lead the overall system to cache or/and memory contentious, simply because both the LLC (Last

Level Cache) and cache interfaces are shared on-chip resources [38, 59]. Or even when diverse tasks compete for a limited I/O bandwidth to carry many reading/writing requests between application buffers and the disk, they contend with each other and experiment ongoing I/O contention with increasing disk latency [74]. Fortunately, the processor industry has been dealing with contentious resources for quite some time when designing low-level, high-speed cross-over paths and exclusive data communication lines [73]. Even though these advances have opened new horizons for more controlled and isolated hardware architectures, resource contention still exists since the devices' capacity to handle instructions is generally not greater than the efficiency of the communication buses carrying data [97], leading to bottlenecks ranging from hardware to operating system (OS)'s subsystems and application buffers. Because workloads' patterns are for the most part unknown by OSs, improvements in resource-allocation mechanisms become increasingly necessary.

In the context of Big Data applications, the CMSs by nature, as they are designed, employ a fine-grained model, where cluster nodes are subdivided into "slots" and jobs are composed of "tasks" that are assigned to slots. Multiple co-slotted I/O-sensitive tasks randomly accessing a hard disk makes its cylinder rotate asynchronously, reducing the volume of requests it can handle per unit of time (throughput) while increasing the requests' completion times (latency). Likewise, many co-slotted cache-sensitive tasks writing to/reading from LLC, make their cached pages dirty most of the time and impossible to be evicted and re-used; they pollute cache pages with each other. These cross-task interference have been extensively explored in recent years, especially along the cache subsystem. StatCC [34] and other similar works, for example, uses statistical models to predict and model cache contention through performance estimation. However, it requires the cache entries to be referenced with a certain "task-label" to infer per-task cache occupancy, which is intrusive and challenging design, making data center administrators most often rethinking about best practices to increase the efficiency of resource allocation mechanisms to proactively assign a task to the node that least interferes on performance.

Our motivation arises from the confluence of data centers' heterogeneity and the inability of existing CMSs in assigning tasks to nodes with interference-awareness. With the actual explosion of frameworks in consonance with the variety of workload patterns, we believe that when all of these are materialized along a wise placement strategy, it may open up opportunities for increasing performance and resource-efficiency in the today's Big Data ecosystem. Understanding performance of data analysis applications becomes a non-return path in the actual scenario of growing scale of Big Data centers. Finally, now, we propose a method capable of profiling applications' sensitivity to contention, and use it to assist an interference-aware placement technique. It is evaluated in a data center infrastructure using popular benchmarks and real-world applications that are representative of significant big data processing use cases (e.g., data transformation, web search indexing, and machine learning). The direct value of this Ph.D. research is in the performance optimization brought

to big data job scheduler through an appropriate performance interference control, which may result in faster Big Data analytics, higher resource-efficiency, and therefore reduced time-to-insight. To this end, this Ph.D. work tackles the research challenges related to performance interference in big data clusters for efficient data analysis. The next sections will describe the scope, hypothesis, and research questions.

1.1 Hypothesis and Research Questions

This Ph.D. research aims to investigate the hypothesis that *an interference-aware container management system would improve containerized data processing applications' performance when compared to the state-of-the-art's interference-unaware container management system*. Some key research questions associated with the hypothesis have been stated to guide this research, as follows:

1. *What is the real need for performance optimization in containerized data processing applications?* This research question's primary objective is to study data processing applications in detail and identify common causes that make performance vary in container-based infrastructures. This study is essential to understand the opportunities for performance optimization in container management systems.

2. *What is state of the art in container performance isolation?* The objective of this research question is to verify the capability of the existing container technologies, scheduling policies, and placement strategies of isolating the temporal behavior (or limiting the temporal interference) among multiple containers. Answering this research question will allow us to understand the approaches that have already been tested, identify their limitations, and point out opportunities for performance optimization.

3. *How to characterize the sensitivity of applications to resource contention?* The motivation for this research question arises from our insight that contentiousness could be observed and measured at the operating system level. Therefore, we are interested in investigating how to discreetly extract operating system's information to characterize applications' sensitivity to contention.

4. *Is it possible to mitigate performance isolation weaknesses in interference-aware container management systems?* Classifying the sensitivity of applications could assist an interference-aware container placement technique to improve application and scheduling performance.

2. BIG DATA AND ITS PERFORMANCE NEEDS

Recently, a growing number of organizations of various sizes and segments have been using the Big Data Analytics philosophy as a strategic support tool. They hope to improve work processes and gain valuable insight into market trends, consumer behavior, and expectations. All these improvements allow corporations the possibility of making decisions that are more precise and anticipated than their competition. In the current extremely competitive reality, data analysis can be the difference between a business' success or stagnation.

Big Data analytics is the process of transforming a set of data so that it can be more better verified with the main goal of discovering information insights. It analyzes large data sets of a specific problem and solves it through different approaches. It is particularly important in areas like sciences, social studies, and business, due to the diversity of possible models [65]. While the data analysis process attempts to derive a trend or understand meaning, data processing is the process of extraction and transformation that occurs earlier in a preliminary stage. Traditional relational data processing software, such as MySQL [39] and Postgres [95], do not provide either resource efficiency or cost-effectiveness, since moving large data sets from one system to another requires many hardware and software resources. This has drawn the attention of both academia and industry to find highly scalable data computing software, where data stream comes from multiple distributed sources, transformed, and then loaded into the system based on the user's needs. A variety of applications have been created by data transformation frameworks that allow distributed, decentralized data processing across clusters of computers, and provide a standard way to build and run applications.

This chapter describes the most popular frameworks that facilitate application creation for the sole purpose of data analysis, and also the most popular CMSs that enable resource sharing for data applications. The convergence of frameworks and CMSs is of paramount importance to understanding the need for performance optimization.

2.1 Big Data Processing Frameworks

The industry has been flooded with a tremendous quantity of valuable information in recent years, with the popularity of distributed computing platforms (e.g., cloud computing) as well as advances in hardware portability, such as mobile, tablets, etc. This has required data to be analyzed with new programming engines capable of processing many petabytes of data securely, while simultaneously and making it available on a large scale. In the case of Big Data technologies, all of these are summarized with a kind of framework that provides

underlying facilities (e.g., data processing engine, data communication, resource management, etc.) with a unified view for end-users working on large projects with unstructured data, security assurance and scalability promises. A handful of frameworks have emerged to meet the ever-growing data analysis' demand. Hence, we outline the most popular open-source implementations, focusing on the aspects that have made them the most comprehensive large-scale big data processing systems.

- Hadoop [106] is a distributed programming framework and an execution environment for the MapReduce engine. The execution environment also includes a job scheduling system that coordinates the execution of multiple MapReduce workloads, which are submitted as batch jobs. A MapReduce job consists of numerous maps and reduces tasks that are scheduled to run in cluster nodes. There are two types of nodes that control the job execution process: the *JobTracker* and a number of *TaskTrackers* nodes. Users submit a job to the *JobTracker*, which in turns coordinates the execution across the cluster. *JobTracker* schedules the job and splits its MapReduce tasks between *TaskTrackers*, which have a fixed number of slots to run the map and reduce tasks. Finally, while the tasks are running, *TaskTrackers* report the execution progress back to the *JobTracker*, which keeps a per-job record of the overall progress. The *JobTracker* always tries to assign tasks to the *TaskTrackers* that are the closest to the input data.

- Apache Tez [88] generalizes the process of executing a complex directed acyclic graph (DAG) of tasks using separate stages, allowing these tasks spread across stages to be run as a single, all-encompassing job. For example, a reduce task of a traditional MapReduce job can feed directly into another reduce task without an intermediate map task, resulting in faster processing jobs. Tez has replaced the MapReduce in projects such as Apache Hive [5] and has in continuous and long progress.

- Giraph [63] is a scalable and fault-tolerant framework inspired by Google's Pregel [58]. Its programming engine was designed primarily to calculate page ranks at Google and later incorporated by many Internet companies like Facebook, Twitter, and LinkedIn [76] to create social graphs across users. The previous versions were designed on top of MapReduce but proved to be inefficient and unnatural completely for several reasons. In Giraph, a worker node or a slave node is a host (either a physical or even a virtualized server) that performs the computation and stores data into the disk. Such workers load the graph and keep the full graph or just a part of it (in case of distributed graph analysis) in memory. Hence, this allows graphs to be partitioned and distributed across many workers, making Giraph a highly scalable graph-processing system.

- Storm [99] is a distributed real-time processing framework for processing large volumes of streaming data. Traditional MapReduce applications are expected to start and finish, but this behavior differs slightly in Storm. Storm continuously processes messages

until they are stopped or interrupted such as audio and video streams. There are two types of nodes in a Storm cluster: the master node and a number of workers' nodes. The master node distributes code across the cluster, assigns tasks to nodes, and monitors the execution process. The workers start or stop tasks as they arrive. A single worker runs a subset of a topology, where a running topology consists of many workers that can be scattered over many nodes.

- Spark [116] has become one of today's most commonly used framework. It was initially developed for applications where keeping data in memory optimize performance, such as iterative algorithms. Spark is a general-purpose distributed data processing engine that is suitable for use in a wide range of circumstances. On top of the Spark core data processing engine, there are libraries for SQL, machine learning, graph computation, and stream processing, which can be used together in an application [123]. There are two core concepts in a Spark cluster: *Distribute data*, occurs is a preliminary stage when a data file is uploaded into the cluster, then it is split into chunks, called data blocks, and distributed amongst the data nodes and replicated across the cluster; *Distribute computation*, where users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs and a reduce function that merges all intermediate values associated with the same intermediate key. Applications written using Spark may be automatically parallelized and executed on a large cluster of commodity machines.

It is worth noting that these frameworks facilitate the creation process by building and deploying jobs across a cluster of computers. In their standalone (or bare-metal) forms, they perform the job scheduling function, as well as coordinate a resource allocation mechanism for resource sharing platforms. Yet multiple frameworks can not coexist in a single cluster, because both the job scheduler and resource management are framework-specific facilities (i.e., one framework is unaware of the resources coordinated by another). Given that these frameworks are independently developed, there is no way to perform fine-grained sharing across their jobs. Essential requirements have been established to address this, which are the basis for an expansion of the current CMS concept to meet the requirements of Big Data. Essentially, it should scale horizontally to thousands of nodes, be decoupled entirely from users' applications, enable high utilization of the underlying physical resources, have very reliable user interaction, enable dynamic resource configurations, and finally enable diverse programming engines and frameworks to coexist in the same cluster. These were some principles defined in the YARN's project—the Hadoop's implementation—that were raised from users' experiences and organizational demands [72].

2.2 Big Data Container Management Systems

Data-driven processing frameworks clearly will continue to emerge, and none will deliver programming engines for all use cases. To arbitrate and deliver on-demand resources to frameworks and allow them to access the same data set, the design of a new resource managers' concept was necessary. Big Data Container Management System is a class of CMS that enables multiple diverse frameworks to coexist in the same cluster. Figure 2.1 depicts a high-level view of a CMS's architecture.

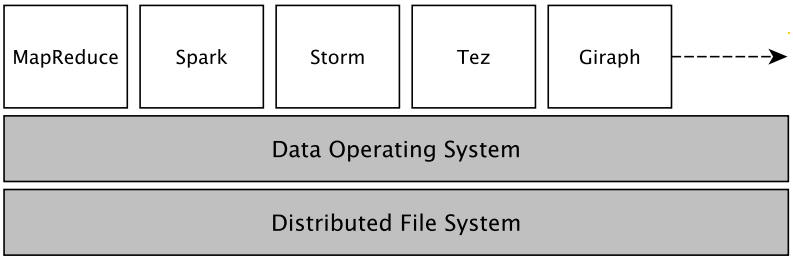

Figure 2.1 – The abstraction layer of CMS under diverse data analysis frameworks

A *job* is a unit of execution (e.g., an application created by Spark) in CMS and can be created by big data frameworks of any nature, and *task* is a unique job's work unit. Thus, a job is typically a set of many tasks that need to be performed to complete a job. A CMS employs a fine-grained resource sharing model, where cluster nodes are subdivided into slots and jobs are composed of many tasks that are assigned to slots. Under CMS, the data storage is implemented in a decentralized way through a Distributed File System (DFS) for greater scalability and data communication control. To store large amounts of terabytes and petabytes, the files are partitioned into smaller data blocks and scattered across the cluster nodes. DFS therefore communicates with the nodes indicating where the blocks reside, merges the blocks, and makes them available to the frameworks. The short duration of tasks and the ability to run multiple tasks per node allow jobs to achieve high data locality, as each job will quickly get a chance to run on nodes storing its input data. Short-lived tasks also allow frameworks to achieve high utilization, as jobs can rapidly scale when new nodes become available. In short, whereas CMS arbitrates and admits access to the underlying physical resources, DFS delivers scalable, fault-tolerant, cost-efficient storage for data sets.

In large-scale High Performance Computing (HPC) clusters nothing prevents multiple diverse applications from being placed/consolidated and executed side by side in the same node; often known as collocation. It is undoubtedly a determining factor for data centers to increase resource utilization while reducing infrastructure costs, but collocation requires a way to isolate applications from the rest of the system mainly for security reasons. We would like to illustrate two scenarios in which the isolation layer plays an important role:

- **Fine-grained resource sharing:** HPC clusters usually have their resources controlled by an RMS (e.g., PBS/TORQUE [25]), or more recently with a CMS (e.g., Mesos), which enable sharing of physical resources among multiple applications. With the proliferation of the multicore technology, current cluster nodes are composed of dozens of processing units. Since one of the primary goals of a resource manager is trying to maximize the overall utilization of the system, a single multicore node can be shared by many different applications at the same time. However, without a cross-applications isolation layer, there are no performance guarantees that the applications will work together in the same node. In this scenario, a performance isolation layer is needed to improve resource sharing by allowing for multiple isolated user-space instances.
- **Configuration management:** HPC clusters are also typically shared among many users' or institutes' applications, which may have different requirements regarding software packages and configurations. Even when applications share certain software packages, it is hard to update them without disturbing others. In practice, software packages in production clusters are often installed and then not updated for a very long time except in order to fix a bug, enhance security, or some type of small upgrades [20]. This makes it difficult to deploy newly developed or experimental technologies in traditional cluster environments. In fact, a user-customized sandbox for configuration management is required to facilitate the creation and maintenance of diverse environments customized according to the users' needs.

In both scenarios, there is a real need for an isolation layer to enhance performance isolation and rapidly meet software requirements' changes. Virtualization based on containers (also known as containerization) has become the key enabling technology through which applications may entirely run within containers with mutual isolation. Therefore, current CMS implementations have incorporated support for a variety of container technologies to enable faster deployment and performance isolation among applications, and also because of its promising advantages such as high scalability with minimal overhead in performance.

2.2.1 Hadoop YARN

YARN is the architectural center of Hadoop's latest generation which enables multiple frameworks with a variety of data processing engines, such as interactive SQL, real-time streaming, and batch processing to compute data on a single platform. Multiplexing a cluster between frameworks improves resource utilization and allows them to share data sets that may be too costly to replicate across the cluster [42]. Applications in YARN are wrapped in the form of container on top of a node, through which they are isolated from each other.

The system is typically coupled with a DFS, such as HDFS [14], which is responsible for storing huge data files and coordinating large data-streamed volume access. All data in YARN is stored as HDFS files, which are composed of many fixed-size data-decoupled blocks (64MB each, by default) distributed across the cluster nodes. There are two types of nodes in an HDFS cluster: a *NameNode* and a number of *DataNodes*. The *NameNode* maintains the file system metadata, which includes information about the files and directories tree as well as where each data block is physically stored. *DataNodes* store the data blocks themselves. Every time a client needs to read a file from HDFS, it first contacts the *NameNode* to determine the *DataNodes* where all the blocks for that file are located. Then, the client starts reading the data blocks directly from the *DataNodes*. Each data block is independently replicated (typically three replicas per block) and stored within multiple *DataNodes*. The replicas' placement follows a well-defined rack-aware algorithm that uses the information of where each *DataNode* is located in the network topology to decide where data replicas should be placed in the cluster. Basically, for every data block, the default placement strategy is to place two replicas on two different nodes on the same rack and the last on a node on a different rack.

2.2.2 Mesos

Mesos [42] has been developed using the same principles as the Linux kernel but at a different level of abstraction. The Mesos kernel is installed on cluster nodes and provides APIs for resource management and job scheduling controlling. The architecture of Mesos consists of the *Master* process that manages agents that span the nodes. The *Master* process enables fine-grained sharing of resources (e.g., CPU, memory, etc.) across multiple diverse frameworks by making them resource offers. Furthermore, it decides how many resources to offer according to organizational policies, such as *Fair Sharing* or *Strict Priority*. To support a diverse set of resource-allocation mechanisms, a plugin interface is enabled to make the development of alternative user-designed resource-allocation mechanism easier. Mesos also consists of two other central components: the *Scheduler* component that registers with *Master* to be offered resources, and an *Executor* process that is launched on agent nodes to run tasks created by different frameworks (e.g., Hadoop, Spark, Kafka, Elasticsearch [37]).

2.3 Understanding Performance in Big Data Clusters

As stated before, container-based virtualization has become the foundation for the development of CMSs. Containers have met CMSs' requirements incorporating an abstrac-

tion layer for fine-grained sharing of resources and promises good task deployment speed, and configuration management capability. The use of the traditional virtualization—allowed by a hypervisor (Virtual Machine Monitor)—has essentially been prohibited in HPC environments for a long time due to its inherent performance overheads [87, 104]. However, container technologies, such as Linux-VServer [118], OpenVZ [36] and LXC [52] have proven to be lightweight alternatives to the traditional hypervisor-based systems, delivering better manageability with near-native performance. There is a trend toward using containers under DCs since it has brought new opportunities for sharing environments where users' requirements differ or frequently change. One has developed a diverse array of multi-framework applications, such as MPI [93], Hadoop and Spark, in order to simplify programming and to better fit parallel applications, then a fine-grained resource-allocation system is needed to prevent mismatches between existing frameworks. Hence, Mesos and YARN take containers for sharing commodity clusters between multiple diverse cluster-computing frameworks. However, as of yet, there have been no studies found that evaluate actual performance overheads in containerized big data applications and their ability to provide isolation from the outside. In recent years, we have demonstrated through a variety of experiments that containers under traditional HPC clusters may offer several benefits for MPI- and OpenMP-driven applications [109, 111]. Here we focus exclusively on Big Data use cases. We conducted a number of experiments evaluating the overall performance of an actual cluster under different container technologies. In particular, we evaluate crucial internal components of CMS and the ability to perform while throttling under stressful high-load conditions.

2.3.1 Experimental Setup

We deployed the container technologies into separate software workspaces, but we worried about using the same hardware configuration (i.e., physical hardware) for a fair comparison. The chosen systems were: LXC, Linux-VServer, and OpenVZ. LXC is the latest and cutting-edge implementation, since its internals were supported on the Kernel's main line, and currently underpins Docker [67]—the today's most popular container manager tool. Our testbed comprises four identical nodes with two 2.27GHz processors (with eight cores each), 8M of L3 cache per core, 16GB of RAM, one 146GB disk and one-gigabit network adapter. Once each system has kernel requirements that differ between each other, we have taken care of compiling different kernels for each system and also in using the same release version. This ensures that the results are not influenced by increases and losses of performance introduced from distinct kernel releases. The kernel 2.6.32-28 was chosen, as it has support to all systems' patches and configurations. For OpenVZ, an additional patch (2.6.32-feoktistov) is needed to allow us to use namespaces and containers. Likewise, to put up the Linux-VServer, the patch (2.3.0.36.29.4) developed by the Linux-VServer team was

installed into the kernel. Unlike OpenVZ and Linux-VServer, LXC is now available with the kernel mainline (since 2.6.24), and no modification was required. We just installed the toolkit (lxc-tool) needed to manage containers and ensured that all requirements extracted from *lxc-checkconfig* utility are met. Finally, we set up a standalone Hadoop cluster and HDFS. HDFS was configured by using 2 Namenodes and 4 Datanodes distributed across the cluster. The HDFS's replication factor was set to 3, and the *Java Heap* size was configured to be 1024MB. The number of Map and Reduce tasks per node was set to 6 and two respectively, in an attempt to balance the overall utilization across the eight-core CPU. As the experiments are high-pressure stress tests, 30 minutes of task timeout was configured to avoid any disruption in samples from the environment. Given a confidence interval by an error margin greater than 95% with less than or equal to 10 sample sizes, any other samples outside the margin were throwing away.

2.3.2 Evaluation of Distributed File System Throughput

We selected the application TestDFSIO (comes with Haddop) to identify the best performance results for DFS. The benchmark runs many disk operations (reading/writing) until it reaches the disk's maximum capacity. Hadoop's users have traditionally used TestDFSIO to pinpoint performance bottlenecks in OSs, HDFS's configurations, and over the network substrate via I/O-efficiency and throughput's performance metrics. For a job using N Map tasks, the throughput is defined considering an index that goes from 1 to N where N denotes the number of tasks. Hence, the throughput metric is indicated as follows:

$$Throughput(N) = \frac{\sum_{i=0}^{N} filesize_i}{\sum_{i=0}^{N} time_i} \tag{2.1}$$

Many TestDFSIO tasks work by running a number of HDFS operations, while individually and the metric *time* comes from the operations' elapsed time, then the *filesize* strongly infer on the results. To achieve optimal accuracy, we ran tests writing ten files per MapReduce cycle, varying the *filesize* from 100MB to 3GB. Also, to produce significant results when setting the replication factor for 2 and 3, we evaluated with only writing operations, since the results have reached a confidence interval superior of 95%. Figure 2.2 depicts the results. The Throughput metric, which denotes the maximum rate at which the data blocks can be processed, is measured in Mbps (million bits per second).

As can be observed, the throughput with HDFS set up with 2 replication factors dropped a little until it reaches 2GB file size, and then remained level. It is because many copies of HDFS data blocks were transferred over the network (across the *DataNodes*), to meet the replication factor. While with a replication factor of 3, it was a bit worse to write 10 files than 2GB in size. Given that the blocks have 3GB in size, then for each

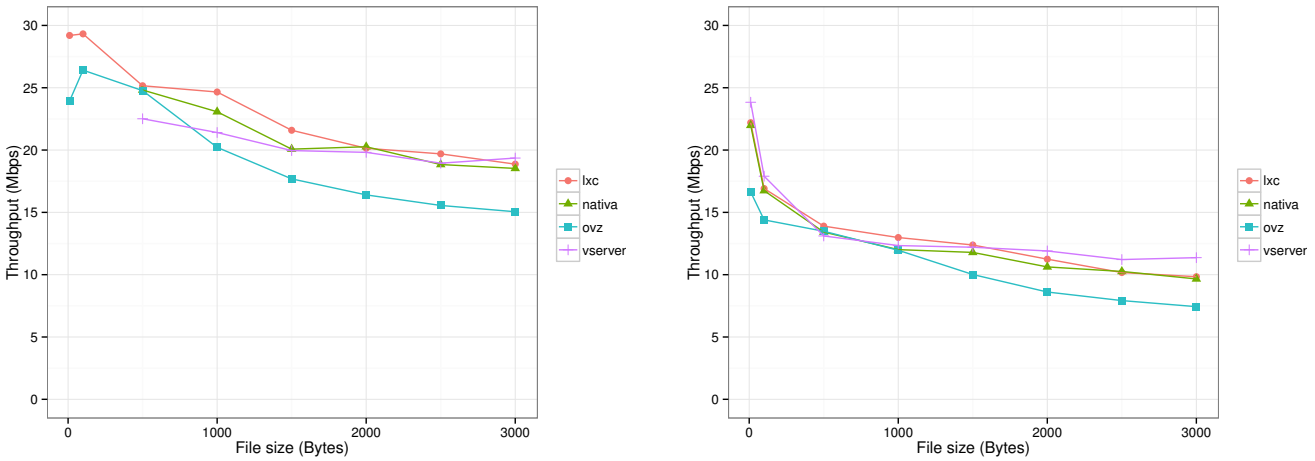

Figure 2.2 – HDFS performance using TestDFSIO with a replication factor of 2 (left side) and 3 (right side).

writing operation, at least 60GB are likely to be transferred over the network. We believe that the network affected the results because the network substrate was unable to reach its maximum capacity until the transferred data reaches 60GB. The conclusions obtained from Linux-VServer are similar to a previous work [107] when the network subsystem was stressed with the NetPipe benchmark [92]. The implementation of the network subsystem differs substantially from-system-to-system. While OpenVZ and LXC take a bridge-based network to increase scalability, Linux-VServer uses a physical network substrate that, by its very nature, reveals a latency very close to the native, but with lower scalability. That Linux-VServer revealed no significant difference when fewer data were transferred over the network. OpenVZ reached a 6.7Mbps average for 3GB files in size, reducing performance by about 3Mbps. The I/O default scheduler also differs among systems. While LXC and Linux-VServer use *deadline* for operations that aggressively reorder requests to increase performance, Openvz takes *CFQ* to classify instructions by priorities, making the resources better distributed among tasks. Albeit there are benefits in using *CFQ* for fair reasons, the inherent overhead needs to be taken into consideration in container systems. Our testbed comprises a single disk per node; then performance could be even better if the number of disks is increased.

2.3.3 Evaluation of Data Block Indexing Mechanism Performance

We evaluated the data block indexing mechanism (NameNode) using the NNBench benchmark (comes with Hadoop) to make the mechanism handles a large number of data requests. NNBench is a simulation-based I/O-oriented load test to create, read, rename, and delete files in HDFS. We ran two sets of operations: (1) creating/writing and (2) opening/reading. The first means that the files are first created and then written, while the second

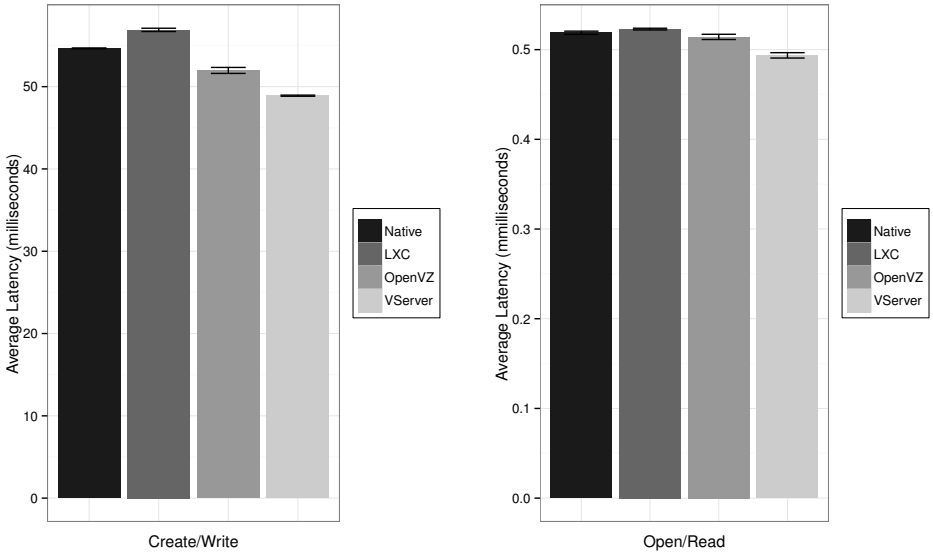

Figure 2.3 – Data block indexing mechanism latency using NNBench

means that the files are first opened and then read. NNBench was set up to flood operations on 1000 files over the HDFS nodes; then it produced exactly 8000 file operations. The Average Latency metric, which denotes the delay between the operations, is measured in ms (milliseconds). The ability of *NameNode* to handle many different I/O operations under high-load conditions can be observed in Figure 2.3. On the left are creating/writing operations (1). The set of opening/reading file operations are on the right. The results become slightly significant when we analyze the differences between the systems. While Linux-VServer achieved an average latency of 48ms for (1) operations, LXC got the worst case at an average of 56ms. The differences are not insightful in a system-wide manner for (2) operations, but the strengths are that no exception was observed during the high load and that all systems reported almost native performance.

2.3.4 Evaluation of Job Scheduling

	Native	LXC	OpenVZ	VServer
Turnaround time (ms)	14251	13577	14304	13614

Table 2.1 – Big Data Job Scheduler scalability using MRBench

Job schedulers essentially attempt to handle one or multiple job queues as efficiently as possible. MRBench is a benchmark that put stress on the Hadoop's job scheduler (known as *JobTracker*). The benchmark measures the efficiency of a job scheduler, while it strives to handle several arriving jobs. The stress test works by dispatching floods of jobs to the scheduling system and measuring the average of all jobs' turnaround times. The MR-

Bench's jobs create of a "dummy" file containing generated data in the order of 1 million lines; then the file is read, proceeded, and written back to the HDFS. There was some similarity when the results are pair-wise compared, as shown in Table 2.1. The Job Turnaround Time metric, which means the total job's completion times from the instant it arrived on the queue to the instant it was served and dropped out from the queue, is measured in ms (milliseconds) and presented by the sum of all job times. As can be observed, the systems properly handled a sequence of 50 jobs that were dispatched from MRBench. Also, it was not identified significant system failure or timeout during the test. It is worth mentioning that MRBench did not put any load on HDFS, since writing operations were not performed on any node. Finally, the containers proved to be a highly scalable platform for scheduling large big data jobs.

2.3.5 Evaluation with Real Big Data Application

We selected real-world big data applications to bring together all of the CMS's components discussed above and evaluate overall system performance. This type of analysis generalizes the evaluation so that we can assertively identify any performance overload based on the bottlenecks learned from the past. These selected applications are part of the HiBench Benchmark Suite [44], which includes a wordcount- and sort-like algorithm, as described below:

- WordCount [40] is a simple application that counts the number of occurrences of each word in a given input data set. It is widely used as a way of comparing the performance among different bi-data clusters. This experiment does not make any stress test; instead it only demonstrates a performance comparison when a real-world application is running on. We created an input file with a dummy text loop up to the size of 30GB, which we believe is large enough to figure out the best performance results, taking into account the characteristics of the experimental cluster.
- TeraSort [79] has been extensively used in the whole Big Data industry for performance comparison using gigantic records among diverse different clusters. Terasort is a MapReduce-driven implementation that aims to sort a volume of data as fast as possible. The application first generates a gigantic input data set and then run the sort algorithm over the data across the cluster. We produced 30GB of random data to be conveniently used as input. We configured a HDFS block size of 64MB to ensure that the application's starting and ending times are not larger than the sorting times; we considered the cluster's characteristics.

As can be observed in Figure 2.4, both applications achieved a near-native performance while running deployed on top of a container. We assumed that performance degra-

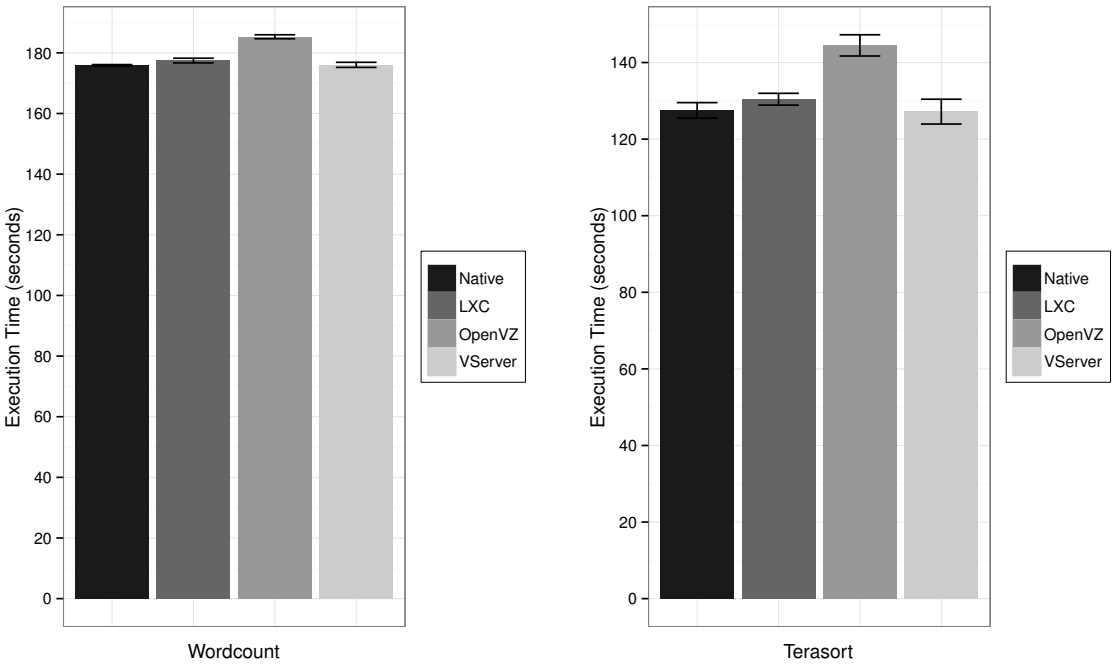

Figure 2.4 – Real Big Data application performance analysis using WordCount and TeraSort

dation observed in OpenVZ could be explained by the results learned from the past. This subtle degradation in performance is a consequence from the configuration of OpenVZ's I/O scheduler, such as shown in Section 2.3.2. In addition, both applications are differentiated by algorithms that are workload-specific and yet presented similar results, which may also be indicative of bottlenecks intrinsic to HDFS.

2.3.6 Experiment of Performance Isolation

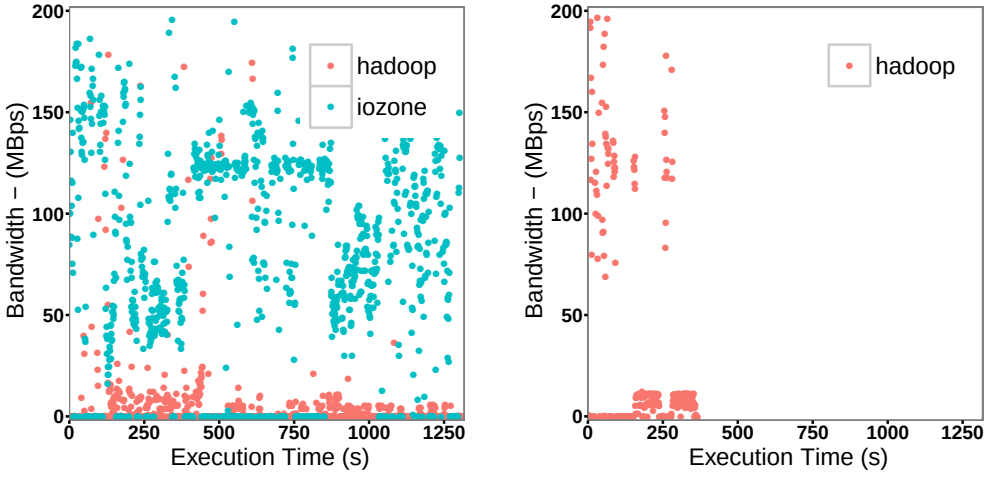

Figure 2.5 – Pairwise applications experimenting mutual performance interference.

We now evaluated the ability of containers to isolate performance between pairwise applications in collocation (i.e., placed on the same node). We selected the TeraSort application from the HiBench benchmark suite and Iozone [77]—a well-known filesystem benchmark that writes a variety of file operations to the disk. We modified the testbed to restrict a node's half capacity to each application. This configuration assigned each application a half of CPU, memory, disk, and network. Both applications started at the same instant time, and while the TeraSort's job runs, Iozone floods many requests to the disk. The graph in Figure 2.5 shows a comparison between the jobs' completion times in function of the disk bandwidth. We can see that the performance adversely affected the Hadoop-driven application. Although the disk I/O scheduler attempts to be as fair as possible, by distributing the disk capacity between applications, the applications are narrowed to write data into DFS and experiment mutual performance interference. It illustrates disturbed, misbehaving data processing in which two disk-intensive tasks are simultaneously writing to/reading from a single disk when the I/O bandwidth is not sufficient to carry all data segments over the channel.

2.4 The Need for Performance Optimization

Big Data applications tend to run on resource sharing platforms, such as YARN, IBM Spectrum Symphony [84], or Mesos. As we have seen, these platforms take container systems, such as Docker, LXC, and so on, to enable fine-grained sharing. Unlike traditional hypervisors, compute resources in container-based systems, such as CPU scheduler, interrupt handler, memory controller, etc., are governed by a unique, shared operating system's kernel, turning itself into a single point of failure. Over the past few years, containerization was characterized as a virtualization architecture that does not correctly isolate performance [111]. Performance isolation refers to the capability of isolating the temporal behavior (or limiting the temporal interference) of multiple containers among each other. Nowadays, with the advent of novel container-based technologies such as LXC, and with the improvements in OS feature to better control the limits of the system resources like *cgroups* [47], and also breakthroughs, the results of isolation are uncertain and deserve to be further studied.

Regarding interference measurements, stress-injection techniques became a usual way to quantify performance isolation by pinpointing cross-workloads' performance interference. These strategies work by running two co-located workloads, where they are reserved to consume equally the same node's resource capacity and can not extrapolate their constraints, otherwise they are poorly isolated. Performance isolation-detection techniques based on stress-injection consist of a two-step evaluation, as shown in Figure 2.6; applied to the containers and data-intensive workloads' viewpoint.

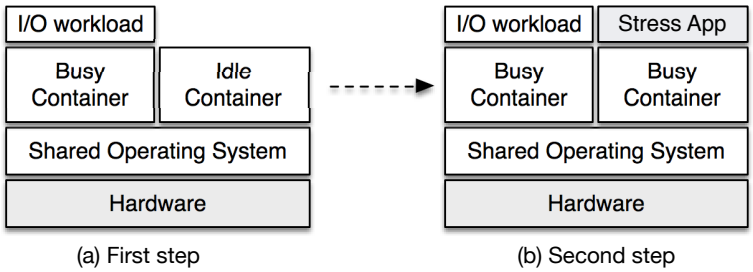

Figure 2.6 – Common injection-based performance isolation analysis approach

In step one (a), a workload runs solo in isolation, so that the system has not been disturbed. During this period a certain performance metric (e.g., execution, response time, IOPS, etc.) for a given baseline workload is collected and stored. In step two (b), the metric is then collected while an on-node co-located disturbing-guest program runs. Finally, the performance isolation metric is measured through the delta-time with the first-step stored metric, collected before the disturbing injection. In hypervisor-based systems, such as Xen, KVM, etc., this or any other similar technique is needed because multiple virtualized applications run on their own guest OS and are governed by a hypervisor, which is responsible for translating instructions from the upper (guest) to lower layers (physical hardware). Hence, looking at the guest OS to quantify performance isolation becomes toilsome or even an impractical task, simply because resource sharing is a hypervisors' burden and the guests do not have a holistic view of the baremetal system (a.k.a dom0), what is expected for security reasons. By the way, the same technique can also be applied for quantifying performance isolation between containers. As such, used in experiments presented earlier in Chapter 2.

2.4.1 Isolation Analysis Between Big Data Applications

Systems	CPU	Memory	Disk	ForkBomb
LXC	0%	8.3%	5.5%	0%

Table 2.2 – Performance isolation between two containers using TeraSort as baseline.

We evaluated the ability of containers to isolate performance, in particular big data application performance, we modified the testbed so that the nodes are shared between two containers simultaneously to force resource sharing. The resources of the nodes were subdivided by two and assigned to each container in collocation, so that each container receives the same portion of CPU, memory, disk and network. The experiment was based on the IBS benchmark method [64] which consists of two stages: (1) Collecting the runtime of a given baseline containerized application; and (2) runs the application again in collocation (side-by-side) with a containerized misbehaving disruptive stress benchmark (e.g., CPU, memory,

I/O, and forkbomb). The performance degradation metric is calculated through the delta time between the execution times collected in the two stages. By normalizing with the delta time the performance isolation issues is quite revealed. We selected TeraSort application—in its implementation by Hadoop—from the HiBench suite as the baseline application. Hence, we primarily experimented LXC, as it is the most recent implementation and also because there has been a trend toward using it in BD-CMSs. Mesos, for instance, takes LXC to subdivide the node into slots for multiple diverse frameworks like MPI and Hadoop. Likewise, YARN has also incorporated LXC to control memory access per framework. The results are shown in Table 2.2. The numbers represent the normalized performance degradation. It was possible realizing that the OS's scheduling is adequately isolating the CPU time and there was no impact during the ForkBomb test. During the memory and I/O stress tests, however, a little performance degradation needs to be taken into account. The ForkBomb is a classic test that loops creating new child processes until the resources become unavailable or the system crashes. No performance issues was noticed during the ForkBomb test.

2.4.2 Isolation Analysis Beyond Big Data

Systems	LXC	OpenVZ	VServer	Xen
CPU	0	0	0	0
Memory	88.2%	89.3%	20.6%	0.9%
Disk	9%	39%	48.8%	0
Fork Bomb	0	0	0	0
Net Receiver	2.2%	4.5%	13.6%	0.9%
Net Sender	10.3%	35.4%	8.2%	0.3%

Table 2.3 – Performance isolation between two containers using LU as baseline.

We also explored other container systems and applications to pursue better performance isolation findings particularly in the memory subsystem. Thus, we now chose a pseudo-application called Lower-Upper Gauss-Seidel solver (LU) from the NPB benchmark [6]. LU is a benchmark that performs matrix factorization being a product of a lower triangular matrix and upper triangular matrix, and also performs complex interprocess communication. The LU application fits this experiment since the algorithm implements a high degree of parallelism that performs continuous memory operations. We ran LU as the baseline and injected the IBS to put pressure on the overall system in a 1-by-1 manner. The results are shown in Table 2.3. As can be observed, the systems had no impact during the CPU-intensive tests, as also observed earlier with TeraSort earlier. However, other resources when stressed had some impact on LU. Containers co-existing on the same computer share the OS's functions, in particular, the kernel, and while the OS handles instructions from the misbehaving container, it becomes unable to execute functions that the well-

behaved container needs to operate efficiently. This disruptive scenario have influenced the performance during the tests of memory, disk, and network. Yet, OS-level I/O schedulers, such as CFQ, Deadline, and Noop, are able to detect resource bottlenecks and divide block devices by fairly reordering/prioritizing tasks. Since the overhead is distributed equally across the tasks, the performance still fluctuates because the schedulers are unable to predict and make decisions based on workloads' pattern. Hence, performance overheads may also arise due to resource isolation issues in the virtualization system, which occurs when a virtual domain exceeds its constraints and extrapolates resource consumption. Since the resource reservation facilities in its more common form are capacity-driven (e.g. GB, VCores, etc.) and not throughput-driven (e.g. bandwidth, IPC, etc.), even though a container receives a limited amount of resources, there is nonetheless leakage due to contentious.

2.5 Summary

In this chapter, we provided the background on the Big Data ecosystems, including an overview of purpose-built data-processing frameworks, data center operating systems, and finally an overall performance analysis. We could conclude from the preliminary experiments that although the containers offer a near-native performance lightweight alternative and configuration management capability to data-processing clusters, their performance isolation still needs to be better controlled. We also have observed that multiple containerized Big Data's tasks under high-load conditions contend for the same compute resource, making performance fluctuate in unpredictable ways.

Current studies have shown that containers' orchestration tools, by their very nature, are unable to provide a good isolation layer, nonetheless they proved to have a near-native performance. It seems very good, at first glance, to deploy high performance applications, while in isolation (solo execution) it narrows the large-scale resource-efficiency data centers' spectrum, which is not the case of big data processing-centre ones. Finally, we realized that by reserving disk bandwidth fractions would be possible to improve performance isolation among containerized disk-intensive applications' tasks and also accelerate the OS's task scheduler. As discussed, this strategy is loss-making in many ways, but its results were instrumental in opening up performance optimization opportunities in the today's Big Data application ecosystem. These opportunities have encouraged us to more closely study the causes that lead resource sharing infrastructures to poor performance isolation from underlying hardware architectures to the highest OS's level.

In general, while specific resource allocation mechanisms may selectively restrict resource usage of tasks in collocation, such mechanisms often do not effectively isolate cross-task performance and accompanying mutual performance degradation. Furthermore, such mechanisms ineffectively balance a trade-off between system resource usage and

task mutual performance degradation simply because they can not differentiate workloads' patterns. Accordingly, container-based resource management systems lack efficient mechanisms for improving performance isolation, especially when the resources include internal processor components that share various caches, pipelines, and other hierarchical on-chip low-level resources. Containers inherit all these shortcomings as they are purely implemented at the OS level, and existing OSs often do not observe lower level resources that are traditionally not fine-grained at the resource-allocation mechanism level. This preliminary study was essential to understand the opportunities for performance optimization in big data processing clusters. As we have shown [108–111], it is clear that performance isolation still needs to be better understood in container systems and that it is the critical factor to enable containerization at an even more significant pace in the Big Data's industry. These opportunities answer the first defined research question and strengthen the hypothesis that interference-aware data centers may speed up containerized big data applications.

3. THE STATE OF THE ART IN CONTAINER PERFORMANCE ISOLATION

As the complexity of distributed, decoupled computing environments are growing fast, problem-solving studies addressing topics such as low-usage, overloading, low-level performance, resource restriction/reservation weakness and even power consumption/energy inefficiency are in fact increasing largely in the whole industry and academia. All of these topics are exceptionally critical to HPC environments, such as those built for the purpose of Big Data. However, we narrow this broad spectrum and survey the concepts intrinsic to the "Performance Isolation" problem. This chapter presents a review of the most likely sources of performance interference, and also the interference-detection and -prevention strategies. This study was organized through a bottom-up approach, covering works that range from hardware architectures to scheduling algorithms. First, of course, answering a preliminary research question, which is the studies' starting point.

3.1 What is Performance Isolation?

Uncontrolled access to shared compute resources can cause contentiousness that leads applications to fail or not run steadily. This performance variation emanating from contention in RAM, disk, cache, or internal memory buses is called performance interference. Disk contention, for instance, may occur when multiple applications contend for access to the bottlenecked disk, and it can not handle requests at the same rate as they arrive. Many efforts have been made to alleviate resource contention at the operating system level, ranging from better scheduling techniques in multi-core architectures [122] to dynamically addressing mapping to minimize memory contention [81]. The steady growth in container adoption has raised a concern about resource contention, and the impact it might cause in resource sharing systems where QoS becomes crucial and can not be violated. Data center administrators attempt to exaggerate resource reservations to applications to sidestep contentious scenarios, making the data center underutilized and no longer resource efficient. In the context of this work, therefore performance interference may be understood as either a resource contention or performance isolation issue.

3.2 Resource Container

Resource virtualization consists of using an intermediate software layer on top of an underlying system to provide abstractions of multiple virtual resources. In general, the vir-

tualized resources are called virtual machines (VM) and can be seen as isolated execution contexts. There are a variety of virtualization techniques. Today, one of the most popular is the hypervisor-based virtualization, which has Xen, VMware, and KVM as its main representatives. The hypervisor-based virtualization, in its most common form (hosted virtualization), consists of a virtual machine monitor (VMM) on top of a host OS that provides a full abstraction of VM. In this case, each VM has its operating system that executes completely isolated from the others. This allows, for instance, the execution of multiple different operating systems on a single host.

A lightweight alternative to the hypervisors is the container, also known as Operating System Level virtualization or System-level virtualization. This kind of virtualization partitions the physical machines resources, creating multiple isolated user-space instances. Figure 3.1 shows the difference between container- and hypervisor-based virtualization. As can be seen, while hypervisors provide an abstraction for full guest OS's (one per virtual machine), containers work at the operating system level, providing abstractions directly for the guest processes. In practice, hypervisors work at the hardware abstraction level and containers at the system call layer.

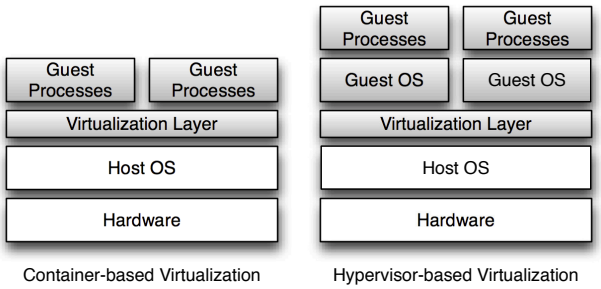

Figure 3.1 – Comparison of container-based and hypervisor-based virtualization.

Since the container works at the operating system level, all virtual instances share a single operating system kernel. For this reason, the container is supposed to have weaker isolation when compared to hypervisor-based virtualization. However, from the point of view of the users, each container looks and executes exactly like a stand-alone OS [36]. For example, a container can be rebooted independently and has root access, users, IP addresses, memory, processes, files, applications, system libraries and configuration files [36]. The isolation in a container is normally done by kernel namespaces [10]. It is a feature of the Linux kernel that allows different processes to have a different view on the system. Since containers should not be able to interact with things outside, many global resources are wrapped in a layer of the namespace that provides the illusion that the container is its system. As examples of resources that can be isolated through namespaces, consider filesystem, process IDs (PID), inter-process communication (IPC) and network [10]. On the other hand, the resources management in container systems is usually done by Control

Groups (cgroup) [47], which restricts the resource usage per process groups. For example, using cgroups it is possible to limit/prioritize CPU, memory and I/O usage for different containers. In some cases, some systems use their implementations to perform the resource management due to the incompatibility with cgroups.

3.3 Getting Beyond Isolation Weakness

A resource is a hardware device or a piece of information that is manipulated by a unique task at a specific time. However, two or more tasks running together at one time may need to access the same resource. This is a large burden for the operating system, as it has the special ability to temporarily grant exclusive access to specific resources [97]. This conflict for resources is called resource contention and can be observed at different levels of the system. These subsystems are explained below.

3.3.1 CPU Contention

Multiple CPU-intensive applications running at the same time contend for a CPU when the tasks require a large number of cycles per unit of time to execute, and the OS is barely able to allocate it for every instruction that is needed. CPU contention is essentially observed in virtualization systems in which multiple virtual domains share CPU cycles through many virtual CPUs (vCPU) pinned over a single real one. CPU contention makes virtual domains to wait in a ready-to-run state while the hypervisor is servicing another vCPU [55]. From virtualization's point of view, the performance metric "steal time" denotes the percentage of time a vCPU waits.

- **Front-side Bus (FSB):** FSB is a communication mesh created for transferring data between the CPU cores and the north-bridge, also known as memory controller hub (MHC) in the computer architecture. Within symmetric multi-processing architectures, the memory accesses are processed by a unique shared memory bus and this mechanism performs quite satisfactorily on a small number of CPU cores. However, on a multi-processing server with a high scalability of CPU cores to satisfy the needs of hundreds of users at the same time, it consequently creates a significant bottleneck (figure 3.2). The NUMA architecture (Non-uniform memory access), which is usually the case in these systems, considerably reduces the amount of congestion by limiting the number of cores that share the same memory bus. Still, the memory access time only depends on where the memory is located in relation to the CPU cores [7].

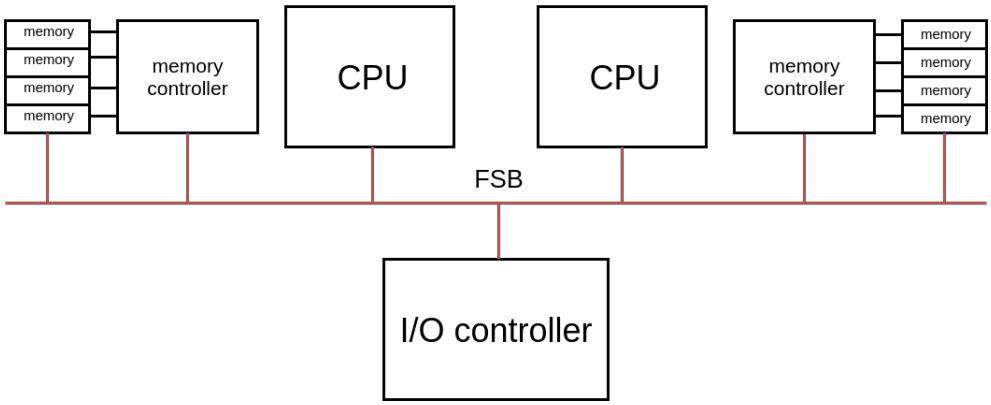

Figure 3.2 – Example of FSB contention within NUMA multi-core architectures

- **Quickpath Interconnect (QPI):** QPI is a technology that provides a full replacement for the FSB that was previously explained. In FSB, all traffic carries over one single shared bidirectional data bus, which can lead to resource bottlenecks. QPI is a point-to-point unidirectional high speed interface developed by Intel [45]. It was built to enhance the high-speed server environment bandwidth because it was becoming increasingly difficult to support higher speeds with FSB interconnection topology. With QPI, each CPU is connected to the I/O HUB or to another CPU using two unidirectional links, therefore, supporting two-way traffic simultaneously. To increase efficiency, the OS needs to take control and decide which CPU/memory will be allocated for that particular workload. Otherwise, CPU contention may occur as shown in Figure 3.3, especially in the NUMA architecture, if a block of memory is constantly accessed from multiple CPU units [73].

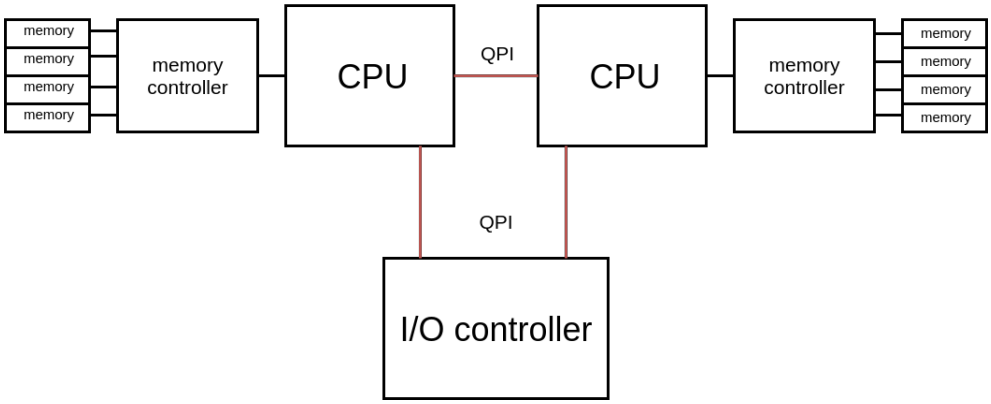

Figure 3.3 – Example of QPI contention within NUMA multi-core architectures

- **Context Switch Overload:** Is the OS's process that switches out a task of a single CPU while the task is waiting for I/O, system call, or any other system-specific interruption, freeing up CPU time for other tasks. When the OS performs a context-switch, it stores the state of the scheduled task, so that it can be restored and resumed at a later time. Context-switch is an important part of multi-threading systems to avoid busy-wait

tasks—scheduled tasks waiting for hardware interruptions (interrupt-driven archs) or system calls that need to be handled. However, the dispatcher which is responsible for saving and loading registers and memory maps requires a certain amount of time to be performed. The processor registers must be saved and restored, the OS' scheduler must execute, the TLB entries need to be reloaded, and the processor pipeline must be flushed [68]. If some process causes frequent heavy context switch loads, the dispatcher process will become computationally intensive, affecting the overall system performance.

- **On-core Cache:** Context switching occurs when a CPU switches from one process to another, allowing the creation of a multitasking environment. The context-switch process has a direct impact on overall system performance, as it can cause contention implicitly in on-core caches (L1, L2). On-cache contention may appear in different ways. Every time a context-switch occurs, the CPU's registers need to be saved and restored later [53]. After the task's context is saved, the L1 and L2 caches are reloaded and the pipeline is cleared for the next task. On-cache contention appears when all data from the previous task, which were stored in L1 and L2, are lost.

3.3.2 Memory Access Delay

Memory contention occurs when active tasks exceed the available physical memory. In a memory contention state, the system can not provide enough memory space for the tasks to run and eventually it starts to crash. Memory contention also prevents the CPU cores from achieving their peak performance. To address this problem, the OS starts to move fractions of active processes to the disk and tries to recover physical memory and reestablish stability. This management strategy is called system paging. Another alternative is to swap an entire process to the disk to reclaim memory, causing high disk overheads. This is an emergency technique used to combat extreme memory shortages, called system swapping. It is relatively difficult to avoid system paging and swapping, and in the end there are only two simplistic possibilities to optimize memory-performance: make more memory available for what the processes depend on most or decrease the extent of the competition for tasks. Unfortunately, if the users continue to spawn more tasks, the system will continue to induce performance overloads in memory, I/O and consequently CPU [55].

3.3.3 Cache Pollution

LLC memory is a hardware device created to accelerate the speed to access data content in RAM. It reduces the system bus and RAM traffic, and restores recent translations

from the virtual memory to the physical one. This procedure is also defined as the principle of locality. When two or more tasks are assigned to the same CPU node, tasks occasionally share on-chip memory space which may lead to contention. This occurs when a greedy task pollutes LLC pages with data that is never reused, forcing other co-located cache sensitivity tasks to fetch data from RAM most of the time [35]. This is the case of streaming applications. On the other hand, when co-located tasks are cache-sensitive, the level of occupancy (capacity) should be taken into account during scheduling to minimize the cache miss ratio—the number of cache misses in function of data loading.

3.3.4 Block Storage Latency

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, bus, and I/O controllers. OS level I/O schedulers, such as CFQ, deadline, and noop, detect resource utilization bottlenecks and attempt to divide block devices by reordering/prioritizing operations in a fairly-balanced manner. As a result, the overhead is distributed equally across applications, but performance still varies unpredictably since the schedulers are unable to predict and make decisions based on workload characteristics. Applications might suffer from interference when there are consecutive random operations arriving in the disk. Then, the head assembly rotates to the track of the disk where the data will be read or written. This scenario makes the disk become busy while the I/O bus is kept below its full capacity. Furthermore, when expressive short operations (under 4KB) arrive in the disk, it makes the disk to handle a bunch of operations without reaching its maximum throughput.

3.3.5 Network Stack Back-pressure

Network card vendors have often changed the way that packets are handled from the hardware buffers up to the networking data-path of the operating system. The faster the network devices become, the more processing time is necessary to handle hardware interrupts and process incoming packets at the same rate as they arrive. The time for processing a packet is strongly related to the multitude of protocol functions that it passes through after being fetched from NIC internal buffers and before reaching application sockets. In NUMA (Non-Uniform Memory Access) architectures, where there are different costs for accessing memory across CPU sockets, it becomes even worse. When data has to be traversed between the sockets it consumes CPU cycles resulting in less work per unit of time, since the tasks consume resources to deal with the cross-talk. A great deal of work has been done

with Linux kernel over the past few years, but the improvements sometimes depend on the workload type and are not always system-agnostic. Therefore, the system must be manually tuned to adjust depth queues, flow control, DMA delay, etc. With an understanding of the underlying factors that actually affect network packet processing and the need to do so, it is possible to minimize overheads and mitigate the network back-pressure.

3.3.6 Power Leakage

It is widely known and the community acknowledges that the main resources are CPU, memory, disk, and network. However, recent studies are showing that another aspect requires attention, power-level contention. As previously mentioned, this resource needs to be addressed as a first class shared resource, like a shared memory subsystem. Power contention takes place when multiple threads are competing for power, leading directly to low-level system performance. For example, if a program executes with high power consumption within chip multiprocessors, eventually the power management system throttles the processor at its maximum, leading to degradation. A solution released to manage CPU and DRAM power contention was a scheduling based approach, which concluding that the impact caused by power contention is clearly hard to predict within different applications, especially when memory/cache and power contention occurs concurrently. Therefore, they must both be addressed at the same time [89].

3.4 Experiments with Big Data Processing

Considering the fact that the disk bandwidth is to narrow to carry such a large volume of data, leading the system to a contentious scenario that affects tasks' performance (i.e., performance interference). Based on the study of these systems, we argue that performance isolation could work better in containers by simply identifying disk's capacity to limit data access flows for long-running disk-intensive tasks. While long-running host-limited (e.g., limited by disk I/O rate) tasks slow down, short-lived host-unlimited tasks speed up. To discuss our claims, we now show a disk reservation-based strategy that accounts for the disk scheduler's capacity handling large volumes of data flows from various supposedly isolated tasks.

Our testbed was comprised of two identical machines equipped with two Xeon Six-Core (24 cores with Hyper-Threading) E5645 2.4GHz processors, 24GB of RAM, and one 300GB 10K RPM SAS disk. The software stack was installed on the Ubuntu Server 16.04 LTS distribution, which was patched with a custom Linux's kernel to support buffered

I/O write restrictions using the *cgroups* toolkit.[1]. On top of Ubuntu, we installed YARN with Docker support enabled—its container orchestration tool—to enable cross-task performance isolation among tasks. We selected the TeraGen application from the HiBench suite [79]. We installed a modified version of IOtop [18] to measure per container disk activities.

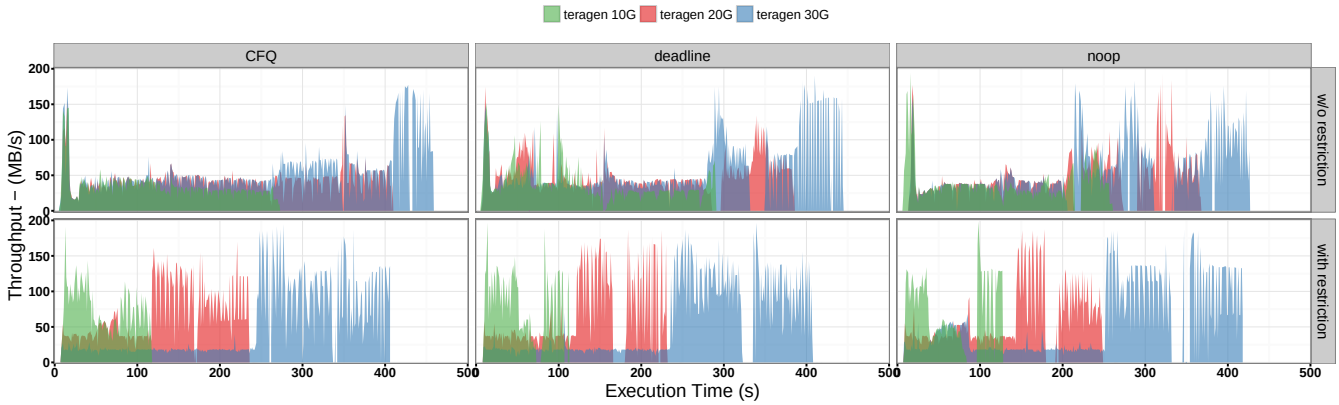

Figure 3.4 – Per scheduler performance isolation weaknesses with and without disk I/O capacity constraints.

The TeraGen was chose to flood the disk with 10GB, 20GB, and 30GB in volume size, and induce high disk I/O activity. Teragen-30GB is the application that writes the largest volume of data to disk. Consequently, it should also induce the highest disk I/O activity and take longer than others to compute. On the other hand, Teragen-10GB is supposed to take less time than others to compute. Finally Teragen-20GB, is the middle-term. We therefore statically limited the disk bandwidth giving Teragen-30GB and Teragen-20GB a fraction of 10% of the total disk I/O capacity, while Teragen-10GB received the other 90%. In addition, we compared the three Linux's default disk schedulers, which are CFQ, Noop, and Deadline. Figure 3.4 shows the jobs' turnaround times and the resulted schedulers' makespan.

The restriction-based strategy shows that a good cross-task performance isolation proved to minimize the overall application completion times by up to 50%, and also accelerate the schedulers' makespan by as much as 26%. When the workloads' patterns are revealed, it becomes undoubtedly easier to control performance interference and deliver resource efficiency to data centers, while also improving the quality of services to applications. Yet the problem emerges when an application's task ends, because the disk bandwidth needs to be redistributed among the others. This requires a system to lookup to scheduler's queues and calculate ready-to-run tasks' reservations to find a new optimal bandwidth distribution. Even so, if the system knows nothing about the tasks' workload, it would require a time-consuming profiling stage for workload recognition. In addition, big data tasks are mostly short-lived and compute tiny unique DFS's file blocks, which traditionally is 64MB in size. This also requires that the system to quickly reserve resources, since the tasks could run and end before even the I/O bandwidth constraints take place.

[1] Custom Kernel: git://git.kernel.org/pub/scm/linux/kernel/git/wfg/linux.git buffered-write-io-controller

Regarding Linux's disk schedulers, *noop* does not perform sorting or any other form of seek-prevention to minimize disk's latency. The I/O requests are simply inserted into a scheduling queue to be served later on the First-In-First-Out (FIFO) basis; it is best suited for solid-state drives (SSD). This is ideal for high-throughput unattended jobs that perform write-buffered (asynchronous) random-access operations, most often in Big Data applications, and also noted in our experiments about 35% improvement compared to the others. *Deadline*, on the other hand, is a latency- and starvation-aware scheduling algorithm that attempts to guarantee a start service time for a request. *CFQ* (Complete Fair Queueing) is the Linux's most common scheduler; it preserves fairer sharing of resources and organizes I/O requests in per-task queues at the cost of worse latency. It then allocates timeslices for each of the queues to access the disk, yet it is not ideal for write-buffered operations, such as those carried out by Big Data applications, and proven to be the worst-case experiment.

3.5 Synthesis

In this chapter, we presented several techniques that have been proposed to prevent a system from contentiousness and reduce co-location issues in CMPs, ranging from dynamic scheduling/placement strategies to predictive algorithms to automate workloads' resource control. However, prominent distributed resource sharing infrastructures, co-location technologies and a myriad of application classes have cropped up every year, prompting engineers to continually rethink all of the practices proposed so far. These include the popularity of cloud computing environments, the emergence of containerization as a lightweight alternative to traditional under-performing virtualization, and the growing Big Data processing applications' classes. Finally, all of these studies were classified on an ad-hoc basis, as shown in Table 3.1.

It is worth noting that most of the works have not addressed the performance isolation problem in systems based on containers. It is because container-based clouds have enticed much more attention recently [24, 80, 91], driven primarily by the Docker's popularity [67, 82]. Performance interference in hypervisor-based systems differs from one in a container-based system in a few different ways. First, multiple VMs on a hypervisor contain independent resource schedulers, each of which managing shared resources without visibility of the others [8]. Second, the guest (OS or virtualized applications) can not be fully informed about co-running workloads [51]. Thus, it becomes toilsome to identify and sidestep performance interference. Moreover, while a baremetal OS can easily access detailed information of applications, the hypervisor has restricted visibility into the guest OS. These are the main reasons why many cutting-edge optimization techniques are difficult for the hypervisor to implement.

Ref.	Hardware Resource	System Architecture	Computing Infrastructure	Application Class	Detection/Management level
[12]	Memory, cache/LLC	OS-level	Cloud/HPC	Multi-thread	Thread scheduler
[122]	Memory, cache/LLC, FSB	OS-level	Cloud/HPC	Multi-thread	Thread scheduler
[90]	CPU, memory, disk, network	OS-level	HPC	Multi-thread	Thread scheduler
[46]	Memory, LLC, disk, network	OS-level	Cloud	Multi-thread	Thread scheduler
[89]	CPU, memory, cache/LLC	OS-level	Cloud/HPC	Multi-thread	Thread scheduler
[86]	CPU, memory, cache/LLC, hypervisor	Virtualization	Cloud/HPC	Multi-thread	VM scheduler
[29]	Memory, cache/LLC, disk, network	Virtualization	Cloud	Single/Multi-thread	Data-center scheduler
[22]	CPU, memory, cache	Virtualization	Cloud/HPC	Multi-thread	Prediction model
[115]	CPU, memory, disk	Virtualization	HPC	Multi-thread/DAG	MapReduce scheduler
[15]	CPU, disk	Virtualization	Cloud/HPC	Multi-thread/DAG	MapReduce scheduler
[98]	CPU, memory, LLC, disk	OS-level	HPC	Multi-thread/DAG	Thread scheduler
[75]	Disk	OS-level	HPC	Multi-thread	I/O placement
[17]	CPU, disk, network	Virtualization	Cloud	Multi-thread	VM placement
[113]	CPU, memory	Virtualization	Cloud	Multi-thread	VM placement
[11]	Memory, Cache/LLC, disk, network	Containers	HPC	Multi-thread/DAG	Task placement
[66]	CPU, network	Virtualization	Cloud	Multi-thread	VM placement
[70]	CPU, memory	Virtualization	Cloud	Multi-thread	VM placement
[48]	Cache/LLC	Virtualization	Cloud/HPC	Multi-thread	VM placement
[83]	CPU, memory, disk, network, hypervisor	Virtualization	Cloud	Multi-thread	Measurement model
[49]	CPU, disk, network	Virtualization	Cloud	Multi-tier	Measurement model
[21]	CPU, disk, network	Virtualization	Cloud	Multi-tenant	Prediction/Scheduler
[3]	Memory, LLC, network	Virtualization	Cloud/HPC	Multi-thread	Prediction model
[22]	CPU, memory	Virtualization	Cloud/HPC	Multi-thread	Prediction model
[4]	-	Virtualization	Cloud	Multi-thread	Monitoring tool
[101]	CPU, memory, LLC, disk, network, hypervisor	Virtualization	Cloud	Multi-tenant	Monitoring tool
[120]	CPU, memory, cache/LLC	Virtualization	Cloud	Single/Multi-thread	Prediction model
[121]	CPU, memory, disk, network	Virtualization	Cloud	Multi-thread/Tier/DAG	Prediction model
[114]	Memory, cache/LLC	OS-level	Cloud	Multi-thread/DAG	Measurement tool
[62]	LLC	OS-level	Cloud	Single/Multi-thread	Measurement tool
[38]	Memory, cache/LLC	Virtualization	Cloud	Single-thread	Prediction tool
[61]	CPU, memory, cache/LLC	OS-level	Cloud/HPC	Multi-thread	Measurement tool
[32]	CPU, disk	Virtualization	Cloud	Multi-thread	Monitoring tool
[119]	CPU	OS-level	HPC	Multi-thread	Monitoring tool
[78]	CPU, cache, disk, network, hypervisor	Virtualization	Cloud/HPC	Multi-thread	Monitoring tool
[31]	CPU, memory, cache/LLC, disk, network	Virtualization	Cloud/HPC	Multi-thread/DAG	Monitoring tool
[64]	CPU, disk, network	Virtualization/Containers	HPC	Multi-thread	Monitoring tool
[28]	CPU, memory, cache/LLC, disk, network	Virtualization	Cloud/HPC	Multi-thread/DAG	Monitoring tool

Table 3.1 – Related work' ad-hoc classification.

In fact, containerization does not provide complete performance isolation, simply because co-located containers share a unique OS and still contend for non-reservable shared compute resources, such as on-chip/-core caches, CPU/IO scheduling, and off-chip/-core memory/IO bandwidth, as seen in Chapter 2, and proven from [109,111]. To the best of our knowledge, yet we have been realizing that current resource contention-detection studies have not been fully addressed performance isolation issues in container systems, and a more comprehensive understanding is lacking at the OS level. It is important to reiterate that CMSs, e.g., Mesos and YARN, have employed a underlying co-location enabling container orchestration system to bring along with it well-known virtualization features, such as scalability, security, and rapid management of users' sandbox configurations. One more note, the lifespan of Big Data's jobs can vary from short-lived applications of a few seconds (e.g., MapReduce-driven) to long-running applications (e.g., Slider-driven) that run for days or even months [102]. Even though a small group of them are running long, their worker tasks typically die as soon as their DFS's data-block processing is completed, moving the applications' pipelines toward new blocks' processing waves. The short duration of tasks and the ability to run multiple tasks per node are likely in need of a resource-prevention strategy.

Finally, when putting these all together it becomes highly challenging for the data centers housing tens of thousands of Big Data's jobs to increase resource efficiency, while maintaining users' QoS expectations. In order to address these shortcomings, the remainder of this dissertation will deal with: (1) the proposal of a system capable of indicating the contentiousness that a given containerized application puts on shared physical resources, (2) in line with the system's outcomes, we will present a more in-depth study of CMS's internal components and how it could be improved to become aware of performance interference.

4. CONTENTION-AWARENESS VIA PERFORMANCE-DRIVEN INSTRUMENTATION

In this chapter we present IntP—the system-level contentiousness monitoring module. By instrumenting the OS's subsystems, IntP gives users insights about task's sensitivity to contention. It is not about common resource usage counters' measurements, but OS subsystems' contentiousness measurements. Thanks to recent advances in processor architectures, the OS has now the ability to fetch architectural counters that were not available earlier. This led us to explore task contentiousness in more detail towards proposing a lightweight resource contention-driven instrumentation module for the Linux's kernel to pinpoint possible sources of performance interference. IntP attempts to (1) not be intrusive in task workloads; (2) to instrument the sensitivity of tasks to contention; and (3) work with parallel tasks. These requirements were raised considering the limitations found in the top studies.

4.1 System-level Resource Contentious Instrumentation

The IntP's sub-modules collect system counters from different hardware components and OS's subsystems. After IntP is loaded, its modules probe OS's internal functions and apply filters on every instruction that comes from tasks to the hardware. For the case of storage block and network stack, interference may come from scheduling queues and the dispatch rate is governed by the synchronism between the OS and an external timer clock. This synchronism is architecture-dependent and comes from an external hardware timer that fires interrupts (jiffies) in time intervals of 1/HZ, where HZ is a compile-time constant that varies from 100 to 1000 in modern OSs. Hence, the variables analyzed by IntP to assess interference in scheduling queues are defined as follows:

Variable	Description
υ	average service time
γ	arrival rate
t	elapsed time
HZ	timer interrupt rate

Table 4.1 – Queue Instrumentation Variables

The service time per unit of time is defined by:

$$f(t) = \frac{\upsilon * \gamma}{t} \tag{4.1}$$

Considering that the OS performs scheduling decisions at intervals denoted by HZ, we divided the service time by HZ and integrate it from the instant t_0 to t_1:

$$I_{queue} = \int_{t_0}^{t_1} f(t)/HZDt \tag{4.2}$$

It means that each time the OS looks at a scheduling queue, a task may or not be in progress. This assumption produces the level of stress that an application is putting on OS's queues at instant time t. The next sections describe the instrumentation points that collect above mentioned variables and other interference perspectives that IntP is capable of infer.

4.1.1 Probes in Block Storage

A good metric to assess performance is defined by the time the disk takes to handle a request (i.e. service time). In order to infer the service time, we measured the delta-time from the *block_rq_complete* to *block_rq_issue* kernel functions. Theses points are called whenever a block segment is added and removed from the scheduling queue after the optimizations have taken place. Based on this, we measured the average service time υ (in milliseconds) for I/O requests and the arrival rate γ to quantify interference in elevator queue. This interference metric is referred to as I_{disk} within IntP.

4.1.2 Probes in Network Stack

With advances in CPU architectures and operating system structures, the network performance has also been 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 arrives, causing suspension of 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. The effect of this is that the operating system can drain potentially multiple packets from the network device receive buffer, increasing throughput, and decreasing latency at the same time as reducing the interrupt overhead. In operating system based on Linux, the 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 its cpu-time is up (limited to 2 jiffies). The processing ends up when the

data is copied to application-specific socket buffer. It turns out that at this point 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 dequeue packets from socket buffer fast enough.

We focused on analyzing the network packet path from the network device (ring buffer) to the application buffer (socket's receive buffer) or vice-versa, so that an application can be classified by its level of pressure placed on hardware device (throughput) and operating system's network stack (latency). The latency is meant as the average service time υ. Since the OS's network stack controls two-ways communications (send/recv) using different queues, the IntP should instrument the scheduler functions in isolation. The average service time of the sending queue is obtained by the delta-time from the *net_dev_xmit* to *__dev_queue_xmit* functions. And the average service time of the receiving queue is obtained by the delta-time from the *napi_complete_done* to *__napi_schedule_irqoff* functions. The average service time υ is given by the sum of both metrics. The arrival rate γ is given by the total of send and receive packets per unit of time. This interference metric is referred to as $I_{netstack}$ in the IntP.

On the other hand, IntP aims to measure the interference that is sourced from contention in the network card, which occurs when the bandwidth is not enough for multiple tasks to carry all the data that is needed (i.e. capacity overflow). The bandwidth consumed per tasks is obtained using the probes as above, but accumulating the length of each packet dispatched and received per unit of time. Hence, the contention in the hardware device is given by:

$$I_{netcapacity} = \int_{t_0}^{t_1} \frac{SUM(length)}{bandwidth} \tag{4.3}$$

Where bandwidth is the nominal limit of the network card capacity.

4.1.3 Probes in CPU Scheduler

The context-switch metric is collected by the IntP's scheduling module. When a context-switch occurs, the module probes the OS's dispatcher process and accumulates the event-waiting time α for each application's thread in blocking state waiting for I/O or system call. IntP ignores the waiting time in preemptive operating systems when quantum expires. The waiting time of a thread in blocking state is given by the delta-time between the instant that it was preempted and resumed back to CPU. The waiting time is collected for all context-switch operations throughout the task runtime in intervals denoted by t_0 and t_1 as follows:

$$\alpha_{th} = \int_{t_0}^{t_1} CSW_{time} Dt \tag{4.4}$$

Given that a application may be multithreading, then we need a discrete equation to sum the waiting time α of a set of threads. Thus, let $S : S \subseteq E$ be a subset of threads running in the system E. The context-switch instrumentation metric of application's threads S is given by

$$I_{csw} = \sum \alpha_{th}, \forall_{th} \in S \tag{4.5}$$

4.1.4 Probes in Memory Controller

IntP aims to assess the level of interference an application causes during memory accesses. 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 integrated memory controller (IMC). The first approach was to use LLC_MISS (last level cache miss) * 64 Bytes (size of cache line). However, the problem with this approach is that he LLC_MISS counter would not include prefetch misses. This can be a huge issue when there are a lot of prefetching activities involved (for example, when there is streaming access involved in the program). Recent CPU architectures made available counters that can be fetched from the uncore IMC, allowing more precise observations. Hence, the level of interference an application puts on memory access is given by:

$$\gamma_{th} = \int_{t_0}^{t_1} (MRC + MWC) * CLDt \tag{4.6}$$

Where *MRC* and *MWC* denote the number of reads and memory writes, respectively. And *CL* is the size of cache line (commonly 64). Finally, the integration of application's threads is summed as follows:

$$I_{mem} = \sum \gamma_{th}, \forall_{th} \in S \tag{4.7}$$

By normalizing I_{mem}, IntP outputs a metric (0..1), which ranges from lowest to highest interference degree, of which is possible to infer the behavior of the application's threads while they are accessing the main memory.

The last level cache (LLC) is a key resource to manage, since multi-threaded architectures and multicore platforms are constantly arise. The chip industry has been introducing a new feature in the hardware that allows an OS to determine the usage of cache by applications running on the platform. This is the case of Intel Cache Monitoring Technology

(CMT) [41]. CMT provides mechanisms for an OS to indicate a software-defined ID for each of 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. The built-in cache module of IntP takes advantages of this feature and begins mapping application's threads to RMIDs during runtime to infer per-application cache usage, thus cache interference can be denoted by;

$$\theta_{th} = \int_{t_0}^{t_1} MSR(rmid_{th})Dt \tag{4.8}$$

Where *MSR* is the interface that read the thread-specific *rmid* from the CPU register during the instant time t. Finally, the total of cache occupancy of an application is given by:

$$I_{cache} = \sum \theta_{th}, \forall_{th} \in S \tag{4.9}$$

4.2 IntP: System-level Resource Contention Monitoring Module

The IntP's components have been developed at the OS level (kernel space). This allowed us to instrument different OS's components from drivers to the scheduler queues in a non-intrusion way. Figure 4.1 depicts IntP's components and the relationship between all them.

Every time a userland tasks is waiting for an I/O event or synchronization operation to be completed, the OS needs to execute privilege instructions to take it place. This operation requires the OS's dispatcher to perform a context switch, saving the current task state in PCB to be restored/resumed later. The **scheduling module** is responsible for measuring the number of context switches a task performs per unit of time, and provide the level of interference that task cause to other tasks during its lifetime. While in kernel mode, a given task can be waiting for storage block I/O interrupt, meaning that it invoked a disk-related system call (read, write, seek, etc.) and is waiting data to be retrieved from disk. All data requests that comes from userland tasks are queued, scheduled and dispatched as the I/O controller is able to handle new requests. Hence, many queued requests make the I/O controller overloaded and unable to handle requests at the same rate as they arrived, given that the disk speed is slower than CPU. The **block storage module** quantifies the level of pressure a task puts on I/O dispatch queue, classifying those that are disk-intensive from those that do not disturb the I/O queue. The **network module** works similarly, but it rely on transmitted and received queues of network buffers, assessing network back-pressure generated per task. All instruction requires memory to be mapped and switched from user to

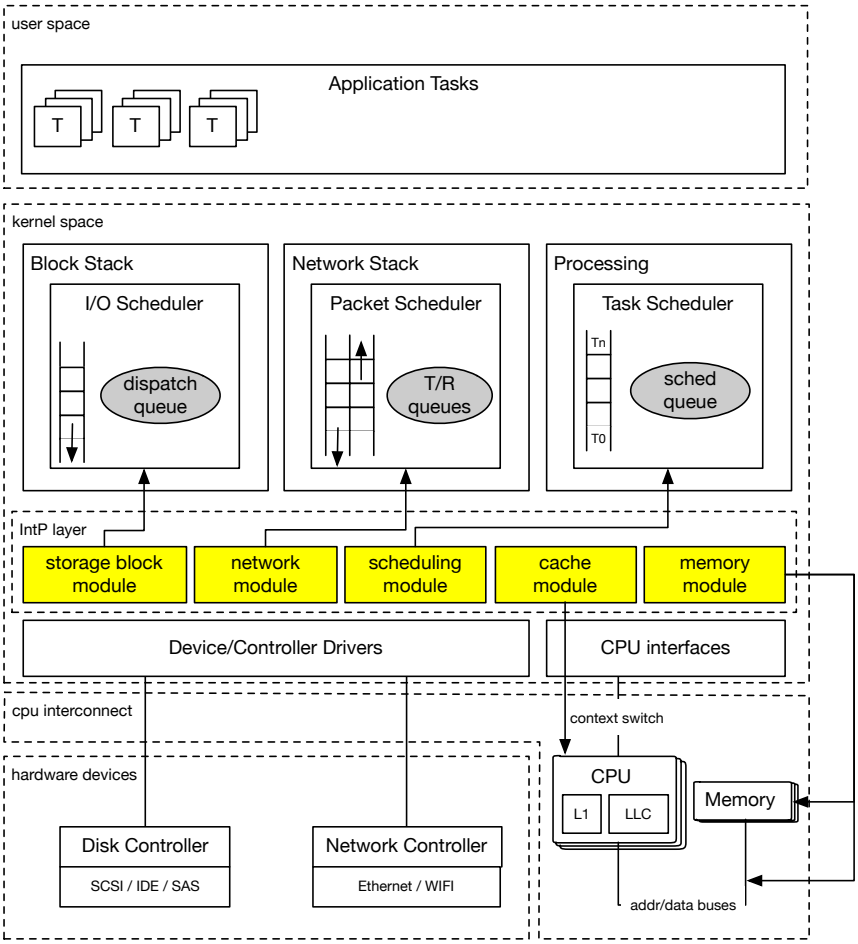

Figure 4.1 – Communication of IntP with the kernel's subsystems

kernel stack. However, there are tasks that requires even more memory to process their instruction. This is the case of memory-intensive tasks such as those that implement machine learning or data streaming programming models. Theses tasks not only use a lot of RAM memory to compute data, but also pollute CPU's caches while running on it. The **memory module** connects to CPU to collect per task cache occupation and derives with cache hits to generate cache sensitivity level. The level of memory bandwidth usage is also measured to classify memory-intensive tasks and differentiate them from cache-intensive tasks.

IntP monitors an application process that it expects as a parameter. It profiles the application during runtime, returning the interference the application generates on each subsystem. Moreover, IntP returns the interference metrics, in percentage, normalized, where the higher the metric is, the more interference the application being profiled generates. IntP differs from other resource usage tools since it inspects OS's internal components to infer contention-related performance interference due to bottlenecks in I/O queues, buffers, and uncore buses. From the memory's point of view, an application that allocates 80% of memory would not imply that it is stressing the memory, as it could just have it allocated and not doing further operations. On the other hand, an application that is using only 20% could

be doing a great amount of reading and writing operations to the memory; thus, it would generate a higher interference. These interference levels, though, are measured by IntP.

IntP outputs interference metrics for CPU, disk, memory, network, and cache. More specifically, it returns the following metrics:

- **netp** - percentage of physical network interference
- **nets** - percentage of network queue interference
- **blk** - percentage of disk interference
- **mbw** - percentage of memory bus interference
- **llcmr** - percentage of cache miss
- **llocc** - percentage of cache interference
- **cpu** - percentage of cpu interference

Figure 4.2 shows the interference levels generated by a given application while varying its workload. This application is disk-intensive and has a high network affinity with another application. It is noticeable that the interference levels tend to increase as the workload also grows. Moreover, the contention in disk storage has a unique behavior, which varies between every data collection. This behavior is due to the write operations that is being executed in an asynchronous manner.

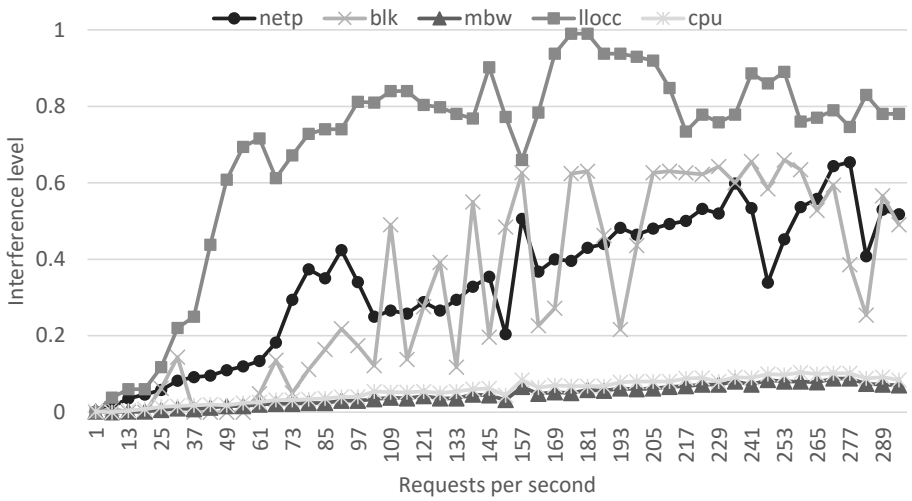

Figure 4.2 – Example of an IntP output for a disk-intensive application.

In addition, IntP may be also useful for verifying the network affinity between two applications. IntP gives the network interference that an application generates, but for this work, we are not considering it as network interference, but rather as network affinity. For example, if an application has network interference, it means it communicates with another

application. This means that the application has a network affinity with this other application. Moreover, the network affinity levels are defined by the interference level given by the network interference. Therefore, the higher the value is, the higher the affinity between two applications is.

4.3 Use Case on Big Data Application Characterization

This section takes advantages of IntP for the characterization of Big Data-driven applications in terms of their resource contention patterns. IntP was used to assess interference metrics of heterogenous applications that put stress on different hardware components and OS's subsystems. We selected popular benchmarks from HiBench Benchmark Suite [44], which are well-known representatives for the field of data analytics. In Table 4.3 is grouped a pool of fifteen applications which were chosen and classified by their resource consumption intensity levels, such as cache, compute, and disk-/network-intensive. This preliminary classification covers resource contention scenarios that IntP proposes to instrument.

App	Programming Engine	Sensitivity
App01	machine learning	LLC
App02	machine learning	LLC
App03	machine learning	LLC
App04	streaming	LLC/memory
App05	streaming	LLC/memory
App06	ordering	memory
App07	ordering	memory
App08	classification	CPU/memory
App09	classification	CPU/memory
App10	search engine	CPU
App11	sort	network
App12	sort	network
App13	query/scan	disk
App14	query/join	disk
App15	query/merge	disk

Table 4.2 – Workload classification

4.3.1 Instrumentation

Our hardware setup comprises 16 identical Dell PowerEdge R810 machines. Each of them equipped with two 3.46Ghz Intel Xeon C5690 processors with 8 cores each (with Hyper-Threading), totaling 32 virtual cores; 64Gb of RAM memory, and four Gigabit Ethernet

adapters. The communication between them is done via a Gigabit switch. We deployed the Linux distribution Ubuntu 16.04 onto the machine. The IntP was compiled and loaded into the kernel with all modules enabled.

The applications were scheduled on the experimental testbed in a 1-after-1 manner to collect the interference metrics for each application individually. Figure 4.3 presents the normalized interference ratios for each application instrumented by IntP. We could obtain some insights from the similarity among the applications. It is easy to see that by placing applications that least interfere with each other on different compute nodes, would be possible to minimize resource contention and maximize performance.

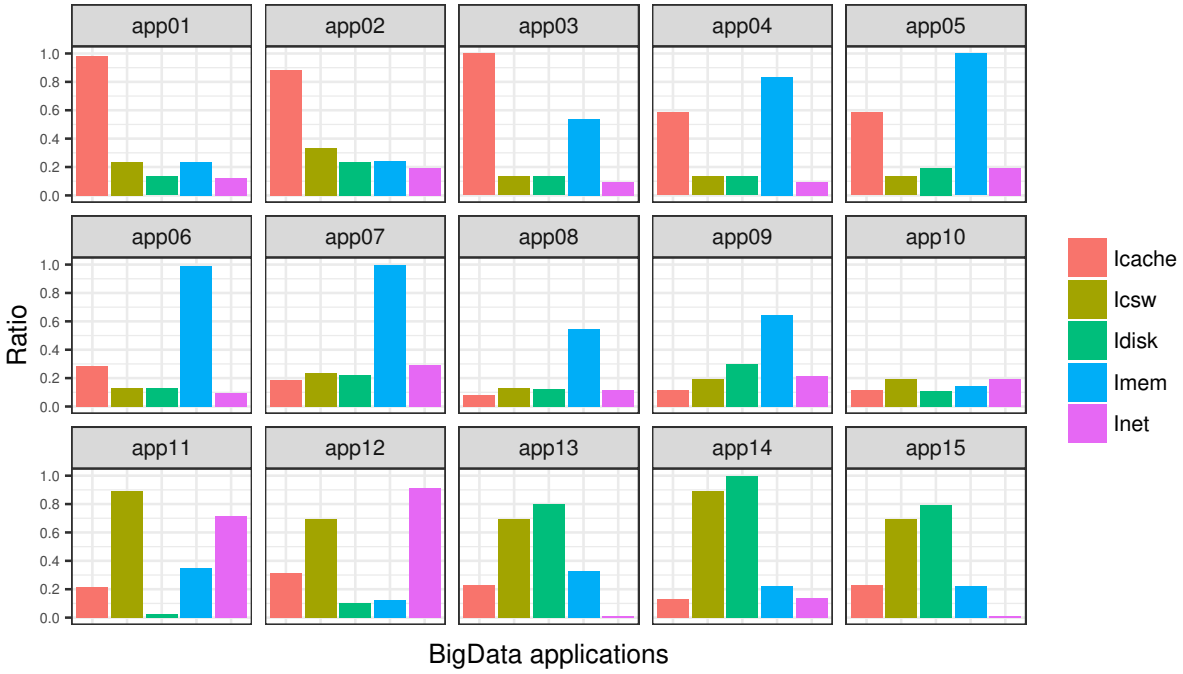

Figure 4.3 – IntP Instrumentation outcomes

4.3.2 Principal Component Analysis

To classify the level of similarity among the applications, it is necessary to reduce the dimensions from 5D to 2D. Hence, we used the statistical procedure called Principal Component Analysis (PCA) [69], that uses an orthogonal transformation to convert a set of observations or correlated variables into a set of values of linearly uncorrelated variables also called principal components. Generally, PCA results are less than or equal to the number of original variables. Finally, we derived to a two-dimension representation that depicts the interference-related proximity between the applications, as shown in Figure 4.4.

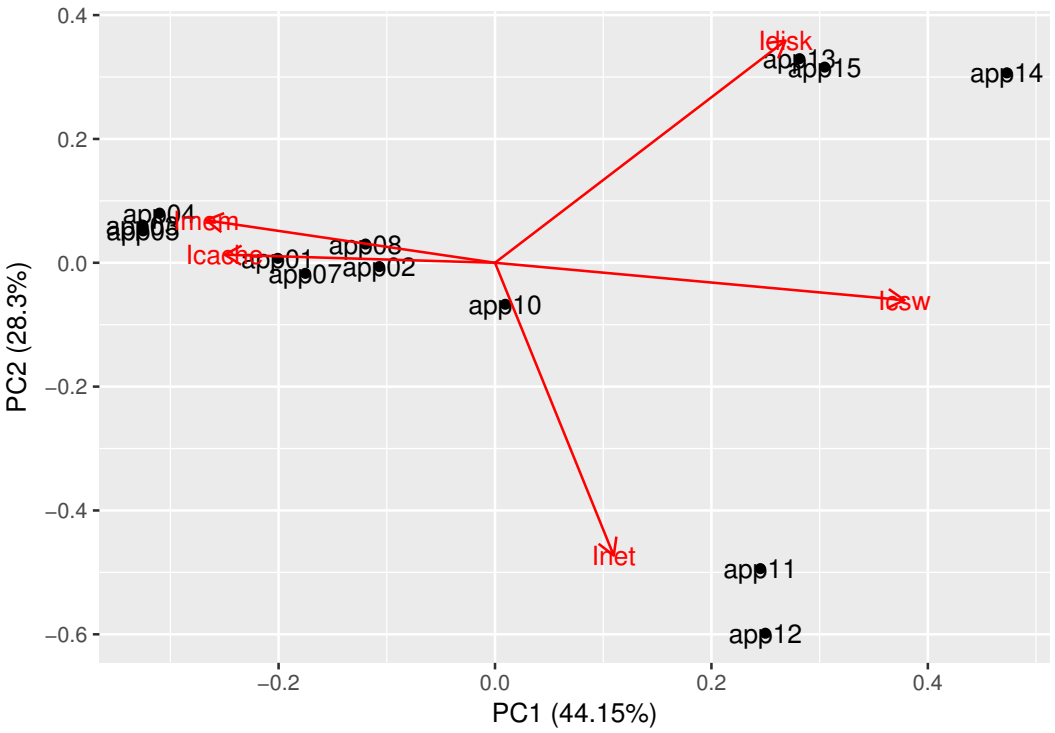

Figure 4.4 – Two-dimension correlation using the generated first (PC1) and second (PC2) components

4.3.3 Classification

PCA lead us to an optimization problem (clustering analysis) in such a way that we can find the K cluster centers and assign the objects to the nearest cluster center, where the squared distances from the cluster are minimized. We applied a centroid-based clustering analysis using the *K-means* method to group variables (i.e. applications) by their similarity. As we are interest in using IntP results for better scheduling and placement strategies in computer clusters, the value of K may be defined considering the number of cluster's nodes. The higher the number of nodes, the greater the granularity of K to accommodate applications with the least possible interference among them. The clustering analysis for $K = 4$, as well as the hierarchical clustering arrangement (tree diagram) are presented in Figure 4.5.

The classification could now be used to assist data center administrators during a scheduling decision or application placement process. One application can be consolidated with another that least interfere on performance, taking into account the level of contention that each application puts on compute resources and operating system levels.

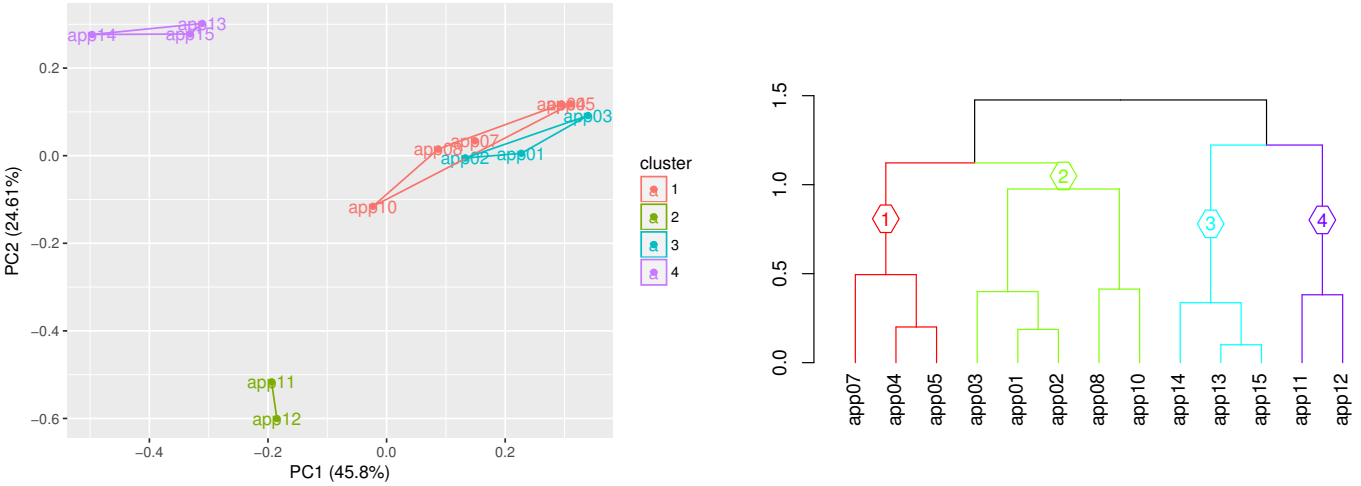

Figure 4.5 – K-means with K=4

4.4 Use Case on Multi-tier Data Processing Application Placement

For the performance analysis, we are using the node-tiers benchmark, considering three multi-tier applications with two tiers each, where both tiers stress the same resource. The first application was CPU-intensive, the second was disk-intensive, and the last did not use any resource intensively. Moreover, we generated an increasing workload, varying the request rate from 0 to 300 requests per second. This variation directly impacts the resource interference and network affinity levels since higher request rate leads to more resources used to answer the requests. Furthermore, we have considered two placement variations, where in the first both tiers were placed in the same PM (Physical Machine) and in the second each tier was placed in a different PM. We have also considered two variations of network affinity, where in the first, the application would have a low communication affinity, where the request size was set to 1KB. In the second variation, the application had a high communication affinity, and the request size was set to 512KB. For all experiments using node-tiers, we used the environment summarized in Table 4.3.

Resource	Description
Processor	Intel(R) Xeon(R) CPU X6550 @ 2.00GHz x2
Memory	64GB DDR3
Disk	146GB - Model: ST9146803SS
Network	Gigabit Ethernet
Number of PMs	2
Cores per tier	4
Memory per tier	1.76GB

Table 4.3 – Environment architecture and characteristics.

Figure 4.6 shows the performance of an application consisted of two CPU intensive tiers. The response time axis is shown in logarithmic scale for better visualization. It can

be noticed that the execution with higher request size (512KB) had a worse performance as compared with the lower request size (1KB). This is a natural behavior since the higher the request size is, the more pressure it puts on both operating system and physical network. Additionally, while the request rate was low, the performance for all executions remained stable. However, as the request rate increased, the execution with high network affinity running in different PMs suffered performance degradation. In this case, the network becomes flooded with many requests, and as the network bottleneck is reached, the response time increases exponentially. On the other hand, while running with same request size, but in the same PM, there is no impact on the performance. Therefore, we can conclude that, for this CPU intensive workload, network affinity is a more critical problem as compared to resource interference.

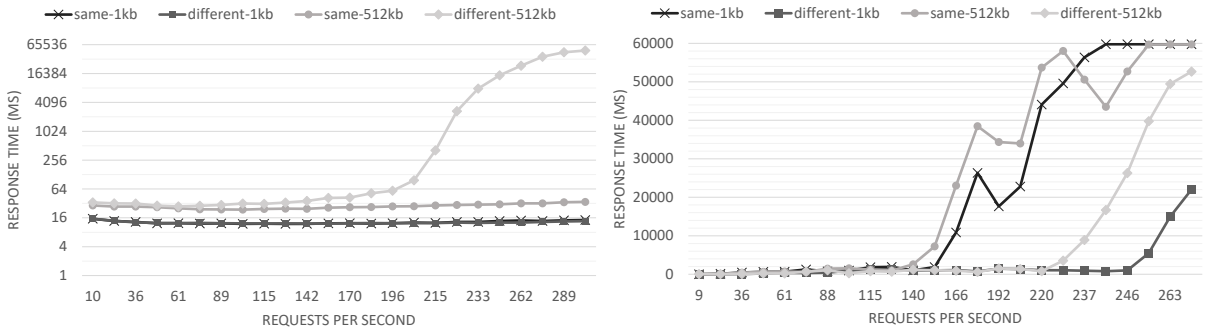

Figure 4.6 – Response time of the applications while varying the workload.

Figure 4.6 presents the response time of the application that had two disk I/O intensive tiers. The response time kept acceptable while the workload was low. However, as the workload increased, the application presents a different behavior from the one seen in the CPU intensive application. All four executions of this application have performance degradation, but this degradation comes earlier in the executions that run the tiers on the same PM. Furthermore, the network affinity also has an impact on the performance, and the higher the request size is, the higher the impact is. As a conclusion of this execution, disk I/O intensive applications tend to suffer more from the interference of co-hosted tiers, but there is still impact generated from different network affinity levels.

4.4.1 Policies

Taking the results of the experiments into consideration, we proceeded and created a set of placement rules. As seen in the experiments, the workload has a great impact on the performance of the application. For this reason, the following rules focus on maximizing the workload that the application is able to handle without having performance degradation.

Moreover, we detail these policies by describing forces that push the tiers closer or farther depending on the intensity of the interference and affinity.

R1. Tiers that interfere the same resource should be placed in different PMs. When there are few servers available, PMs might have to host tiers that interfere the same resource. However, it should be given preference for separating the tiers that generate the most performance degradation. In this case, disk I/O tiers have a strong repulsion, while CPU intensive tiers have a weak repulsion.

R2. Tiers that have network affinity should be placed in the same PM. When all the resources of a given PM are being used, it should be given preference to place in the same PM the tiers that have higher affinity, i.e. higher request size. Moreover, the higher the affinity is, the more attraction force it generates.

R3. Tiers that do not interfere nor have network affinity may be placed anywhere in the infrastructure. There is no force pushing the tiers.

R4. Tiers that interfere and have network affinity should follow a sub-set of rules. These sub-rules extract the best trade-off, which should generate a placement that leads to the best quality of service (QoS) possible.

R4a. CPU intensive tiers should be placed in the same PM. The attraction force generated by affinity is stronger than the repulsion force generated by interference.

R4b. Disk I/O intensive tiers should be placed in separate PMs. The performance degradation that comes from interference leads to a stronger repulsion than the attraction generated by network affinity.

Even though these policies are useful for optimizing the placement, they have two main weaknesses. Firstly, it would be important to expand these policies, considering other resources, such as memory and cache. However, the policies would become too complex and hard to be understood. Secondly, these policies do not consider the levels of resource interference and network affinity. This would lead to applications to not have the optimal placement.

4.5 Comparison to Related Work

Virtualization increases along with its co-location's ability in data centers' enterprise globally. Virtualized data centers have made cloud platforms to become more popular among a diverse set of users executing high performance computing applications, user-facing web services, machine learning algorithms, etc.. Co-location is barely the most beneficial virtualization's feature and has taken clouds' providers to strive for ever-increasing

resource utilization at the lowest infrastructure cost. This popularity has guided many studies aimed at the detection and remediation of contentious root-causes to alleviate introduced co-location issues at the hypervisor level, such as Xen, KVM, etc., and make them bearable to complete performance isolation. Nowadays, with the "era" of containerization and its adoption underneath many clouds and high performance data centers, the performance interference' research topic has brought additional attention closer to the OS-level virtualization perspective. Container has become more popular recently; however very few works have extensively approached the performance interference problem to improve isolation among containers at the operating system level. However, it is arguably a trending research topic.

Regardless of whether the co-location key enabling system is virtualization- or containerization-based, a reliable performance indicator is needed to systemically detect a disruption, anomalous or precisely contentious situation. It can be either a high-level application performance metric (e.g., execution time, transaction per second, response time, etc.) or a low-level architecture counter (e.g., IPC, IOPS, cache miss, etc.). Likewise, IntP collects low-level architecture counters, not to detect performance interference, but to give users intrusive task' information. Although most of the state-of-the-art techniques have addressed the interference problem from the virtualization designs' viewpoint, they have also been driven by performance indicator-assisted measurements that generally fit into one of both. IBS [64], for example, is an interference micro-benchmark (a.k.a. micro-bench) suite that spans CPU, memory, disk, and network. One injects a micro-bench into a misbehaving VM to induce interference on the co-located well-behaved VM, while a user-defined high-level application performance metric is used to infer degradation through a delta-time with a baseline performance metric. IBS relies on deprecated Linux's resource-stress apps, and they have to be executed manually and individually, leading to error-prone, inaccurate, and unreliable results. Delimitrou et al. [28] designed a stress-driven micro-bench suite referred to as IBench. It promises highly reliable results using a set of self-crafted functions to emit interference of increasing intensity to system-wide resources. However, neither the compiled IBench's program nor the source codes are available to the community. SmashBench [59] is a contentious micro-bench suite. The SmashBench's stress-driven micro-benches inject a tunable amount of "noise" in compute resources, and decouples the characterization of an application's sensitivity to contentious noise, in particular, to the memory subsystem, and the noise it emits to the subsystem. Likewise, the noisy program should be injected into the node to emit interference and quantify its effects on resources. In addition to these techniques, other remote works have explored interference detection in hypervisor-based systems by collecting hypervisor's performance counters from its scheduling components. In contrast, but designed differently, IntP works within the OS's kernel like a minimalist, but non-intrusive resource sensitiveness reporting module. Besides, it adds a functional module into the OS's

kernel to make it aware about tasks' workloads, and per-task noise counters accessible from userspace.

4.6 Summary

In this chapter, we presented a system-level instrumentation-based contentious profiler that gives user valuable information that we argued to be a good starting point to assist in a workloads' characterization process. IntP is a kernel-built-in module, which collects architecture counters from at the OS level to assist data centers' providers in the decision-making process, making conscious decisions about re-location and re-configuration applications' placement planning across clusters' nodes on a resources' sensitiveness basis. Further, IntP has the ability to give users information about how their applications are intrusive to the hardware and OS's subsystems. It not only profiles an application, but also provides insights about application's resource needs during application runtime.

While current works have focused on performance isolation issues in hypervisor-based systems due to their broadly adoption, container-based systems have contributed to increased data centers' scalability on an unprecedented scale, and deserve a lot of attention. The usual way to provide virtualization benefits while maintaining performance levels is through a lightweight and non-intrusive underlying layer. Containers underpin this layer by wrapping applications and giving them the illusion that they are running on its own operating system (i.e., they are spatially isolated).

Minimizing cross-task performance interference is a trend topic nowadays and it occurs most of time due to the inherent nature of data-processing frameworks that move large volumes of data to be processed on shared data centers. However, we are missing of a very accurate benchmark to quantify performance interference of virtualization systems that characterizes the workloads by their similarity regarding resource utilization. With IntP we will be able to accurately test the isolation layer of different virtualization technologies and the resource sensibility per application. This characterization allows pinpointing any possible resource contention generated by the applications, prevent them from crash or slowly run before the deployment in a production environment. Although still with a poor isolation layer as shown in Chapter 2.

We would like to illustrate two use cases where IntP could play an important role in performance optimization strategies:

- **Big Data job scheduling.** Specific big data applications are composed of many short-lived tasks that load a tiny DFS's data block and soon die. That would be challenging to indicate interference using an injection-based approach simply because that tasks could complete before the injected program' measurements take place. In this case,

IntP can assist the job scheduler with information about tasks' sensitivity to contention, which is extracted between the instant the tasks start and exit. With this information, the scheduler is capable of deciding which piece of hardware is more likely to be the bottleneck and look for a queued application that best interleaves with it. Or even if one application starts to affect others, it could be migrated to another node to minimize interference and increase performance.

- **Multi-tier application placement planning.** Multi(N)-tier architecture has been widely adopted in the development of web- and mobile-designed applications. These applications are essentially subdivided into physically separated "tiers," where each tier is responsible for a specific processing task, such as presentation, application processing, and data management functions. These tasks are application-specific, and their logical structuring mechanism may vary from-application-to-application. This logical segregation makes applications to forcibly distribute different tasks across a cluster, and they normally employ diverse workload patterns and require different compute resources, such as CPU, disk, network. Based on the fact that two co-located tiers could interfere with one another due to contentiousness, by separated placing them on different nodes would be an opportunity to minimize on-node contentious' effects. However, sometimes two or more tiers are strongly connected and have network affinity, meaning that they commonly exchange large data streams among them. Therefore, IntP could infer cross-tiers' contentiousness along a placement planning to suggest an optimal distribution that accounts for the trade-off between performance interference and network affinity.

In a broader sense, an IntP-assisted cross-tier interference analysis is fundamental to understand the impact the trade-off between interference and network-affinity has on overall application performance. When resources are better utilized, the performance of the co-located multi-tiers tends to improve as the network performance underlies application's performance expectation. In a more restricted view, Big Data short-lived tasks are instrumented for better workload patterns and this information when used proactively makes more balanced placement distribution increasing application completions time as well as scheduler performance.

5. INTP-ASSISTED CONTAINER SCHEDULING FOR BIG DATA JOBS

Despite offering high performance levels, containerized applications suffer from cross-interference, as described in Chapter 2. Cross-interference occurs when two co-running applications' tasks (i.e., provisioned in the same physical node) contend for the same compute resource (e.g., CPU, cache, memory, I/O buffer, etc.) and their performance is adversely affected. In a MapReduce cluster, for example, interference may cause performance overheads in Map tasks, impairing the functionality of Hadoop's internal components, such as the *Jobs Scheduler*, fault tolerance mechanism, and configuration strategy [117]. The problem may be even worse in CMS. Having many co-running tasks may cause unexpected performance variations due to the unpredictable tasks' workload pattern that can be framework-specific. Spark, in particular, implements an interactive query engine that performs in-memory computing. Stream on the other hand, computes unbounded streams of data in real time. These workloads' pattern mixes may compromise the efficiency and effectiveness of CMS's task scheduler simply because it is unaware of the full extent of the workload patterns and therefore has difficulty determining the best placement planning (i.e., those nodes which cause the least amount of interference). Unexpected interference-related task slowdown in CMS may render its traditional task schedulers ineffective, impacting on the scheduler' makespan and user's experience.

These shortcomings are the motivating factors that led us to study how to optimize application scheduling in CMS as a case study for IntP. Hence, here we present an alternative interference-aware scheduling policy for CMSs.

5.1 Big Data Common Workload Patterns

To discuss our claims that different frameworks may drive different workload patterns, we performed a system-level characterization of typical data-processing applications through real job executions. In particular, we investigated Hadoop and Spark using popular benchmarks and real-world applications that are representative of significant uses of Big Data processing engines (e.g., data transformation, web search indexing, and machine learning).

Our job execution results were obtained in a real cluster consisting of four identical Dell PowerEdge R810 machines. Each of them equipped with two 3.46 GHz Intel Xeon C5690 processors with eight cores each (with Hyper-Threading), totaling 32 virtual cores; 64 GB of RAM, 146 GB of local storage (SAS), and four Gigabit Ethernet adapters. The machines were interconnected by an Ethernet switch with 1 Gbps links. Regarding software

components, we deployed a 4-node Hadoop (version 2.7.1) cluster on top of Ubuntu Linux 14.04 LTS operating system. Finally, the resource consumption was collected using the *dstat* tool, which comes with Ubuntu.

5.1.1 On Framework-specific Applications

Firstly, we analyzed how data-processing applications differ in terms of workload patterns when they are created by the same framework and programming engine. To guide this investigation, we chose different MapReduce-driven algorithms that are part of Hadoop. They were K-means, Pagerank and Naive Bayes as below:

- In data mining, k-means is a Clustering method that aims to partition N observations among K groups where each observation belongs to the nearest group of the mean. The method is commonly applied to large data sets, particularly when using heuristics, and has been successfully used on a variety of topics including market segmentation, computer vision, geostatistics, astronomy, and agriculture [43].
- PageRank is an algorithm used by the Google's search tool to position websites among the results of the web searches. PageRank measures the importance of a page by counting the quantity and quality of links pointing to it. It is not the only algorithm used by Google to classify web pages, but it is the first used by the company and the best known.
- In probability theory and statistics, Bayes' theorem describes the probability of an event based on a prior knowledge that may be related to the event. The algorithm has become popular in the Machine Learning area to categorize texts based on word frequencies.

While the applications' jobs were being scheduled in the testbed in a 1-after-1 manner, the workload metrics were being collected from a single node in isolation. It means that no job was sharing resources during runtime. Our understanding comes from the portion of resources that the applications consume during their job sequences. Therefore, we traced each job promptly and divide the start and end times (timeslices) for better analysis, as shown in Figure 5.1.

Although the applications are Hadoop's MapReduce-driven, they differ significantly in resource usage, and there seems to be a data engine-specific pattern during job sequences. In MapReduce-based workflows, three high-level abstraction phases drive the operations: (1) data is read from disk; (2) Map or Reduce (or shuffle) operations are applied; and (3) the computed result is written back to disk. The peak of disk consumption

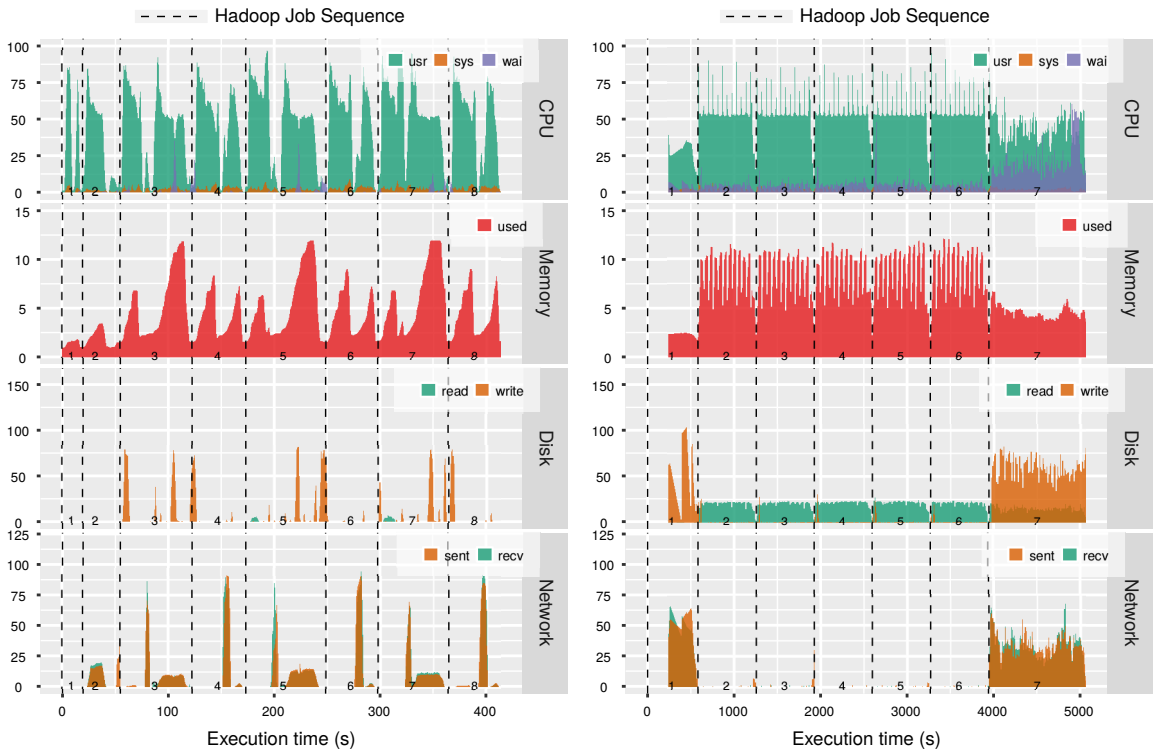

Figure 5.1 – Resource-use patterns of different MapReduce-driven algorithms

in K-means's first task is noticed when data is read from disk. The data processing is performed by the jobs that come next until the resulting data is written back to disk by the latest scheduled job. While K-means has taken advantage of memory footprint for intermediate data processing, the Naive Bayes does not seem to impair memory most of the time. The PageRank's job sequences, however, follow a pattern clearly noted for each two-job iteration. The distribution is evenly divided among all documents in the collection at the beginning of the PageRank's computational process. Our result does it to make sense, given the uniformity very similar to the learning algorithms.

5.1.2 On Framework-agnostic Applications

In contrast, we now analyzed how a specific data computation algorithm differ in resource usage when it is created by different frameworks. To guide this characterization, we selected the TeraSort application in its Hadoop and Spark's implementation. TeraSort is a popular sort-like application that measures the amount of time to sort one terabyte of randomly distributed data in a cluster. Most data in TeraSort pipelines start with a large amount of raw data, but as the pipeline progresses, the amount of data is reduced due to filtering out irrelevant data or more compact representation of intermediate data. Sorting is one of the most challenging because there is no reduction of data along the pipeline and

requires more one-node resources to compute. This led us to choose a sort application, to evaluate the same algorithm with the least intrusiveness of the frameworks.

To calculate system resources required for a Terasort job, we took memory and CPU characteristics of the nodes into account. The optimum number of Map and Reduce slots needed were calculated based on node's CPU and memory availability. We also took disk I/O characteristics into account when computing the overall system resources across nodes. These ensured disk bottlenecks are correctly identified during the resource allocation process making Hadoop and Spark jobs perform better. Results from both frameworks are compared in Figure 5.2.

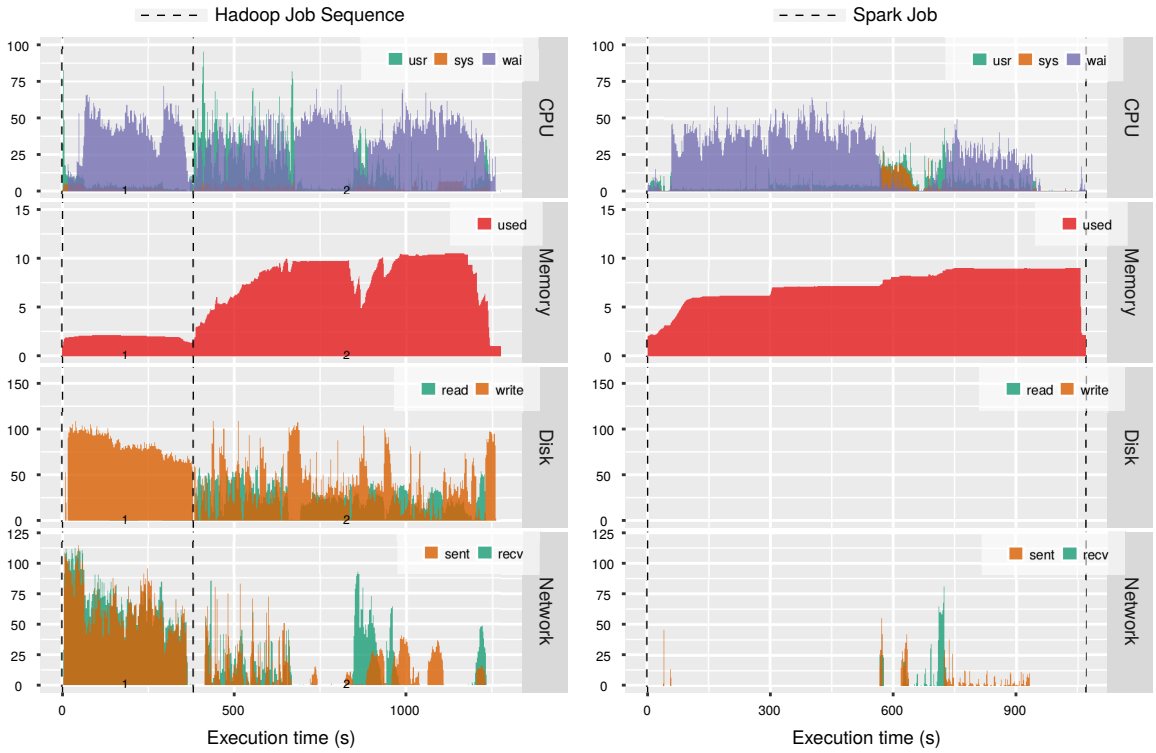

Figure 5.2 – Resource consumption patterns of a data processing algorithm created using different framework

As stated before, Spark achieves a significant speedup from keeping intermediate data in cache memories and becomes more efficient than MapReduce for data that fits in memory. Although both experiments have been run with the same data set and complexity size, it is worth noting that the resource usage varies considerably among them. This is due to how Spark and Hadoop differ in implementation and class of problems they seek to solve.

We can understand from our preliminary experiments that resource usage are framework-independent and algorithm-agnostic. It means that a single application may require diverse hardware resources when they are created using different data frameworks. Moreover, multiple applications may require also different hardware resources when they are created using the data framework. Our most valuable finding relies on the workload patterns that were observed in the application's job sequences.

5.2 Interference-aware Container Scheduling

We propose an interference-aware scheduling policy for CMS applications that run over a container-based resource orchestration system. The idea behind our scheduling consists of **profiling** queued applications based on their resource needs; **classifying** their resource consumption pattern; and **scheduling** applications's tasks on the best-suited node—the node that causes the lowest amount performance interference. The policy main goal is to increase application performance and minimize the makespan.

5.2.1 Application Profiling

Consists of characterizing an application's resource needs by analyzing its resource usage pattern along the runtime. Many techniques have been proposed ranging from kernel-based [100] to run-time application-level profiling [2]. Kernel-based techniques are the most beneficial, because they do not require any changes to the application at the source or binary levels. Furthermore, collecting resource usage in terms of the percentage of CPU usage, I/O throughput, and/or memory bandwidth requires a thorough analysis of the hardware counters at a fine time scale. Kernel-based toolkits, such as the Linux Trace Toolkit (LLT) [33], are quite capable of performing.

In general, data processing applications run over the same cluster [103], so that the resource usage may be tracked just once for each application. In addition, these applications consist of many parallel tasks computing the same data so that the resource usage pattern of all tasks is supposed to be the same [15]. Huang et al. [44] show that a MapReduce application flows over resource consumption patterns over its runtime, which makes profiling data processing applications an easier task.

5.2.2 Task Placement

Pulling a task based on a scheduling policy, and assigning it to an available node's slot. To do this with interference awareness, after pulling the queued task the algorithm should profile the task, predict the interference for each node, and schedule it on the node with the least performance interference. Profiling runtime applications is not a straightforward task, given that different tasks may burst arbitrary resources, causing variations in resource consumption. In addition, an intrusive profiler can induce the performance of applications and therefore compromise their reliability. To overcome this, we profiled applications at the end of each task's execution and stored the results in a repository to be used in the

next iteration. A similar approach was used by Delimitrou et al. [29] to profile cloud computing applications.

5.3 Prototype-driven Learning

The efficiency and effectiveness of the scheduling policy was evaluated in a YARN-based cluster. The next subsections present the YARN's architecture, the changes necessaries for the implementation, and how the policy interacts with the original YARN's built-in components.

5.3.1 Design in YARN

The architecture of YARN consists of three principal components: the *ResourceManager* process that tracks usage of resources, arbitrates users' right access and monitors the states of the nodes; the per-node *ApplicationMaster* process, which is responsible for monitoring the logical execution of a single application. *ApplicationMaster* also requests computing resources from the *ResourceManager*, generating a plan of its work and coordinating the execution of the application using the resources it receives. Finally, the per-node *NodeManager* process that monitors the resource usage (e.g., process, memory) of individual containers. *NodeManager* is also responsible for configuration the environment for applications. These three components are organized in a two-layer architecture: the YARN platform and the framework-specific layer. The platform layer is the first-level scheduling where the resource management takes place. The framework layer is the second-level scheduling where the execution of applications is coordinated. The *ResourceManager* is the component that implement the first-level scheduling, while the second-level is framework-specific takes place in *ApplicationMaster*. Figure 5.3 illustrates two different frameworks running applications in YARN.

The outer boxes are the YARN components *ResourceManager* and *NodeManager*, while the inner boxes are the containers where applications run in isolation. Basically, the job submission workflow in YARN is as follows: (1) a client submits a job to the *ResourceManager*; (2) the *ResourceManager* in turn deploys the *ApplicationMaster* container for the job on an available node's slot; (3) the *ApplicationMaster* organizes a plan for its work and requests computing resources for the application's tasks start, (4) the *ResourceManager* admits the resource request from *ApplicationMaster* and returns back the resources wrapped in a container form with information about which node it should be started up; and (5) the *ApplicationMaster* contact the node's *NodeManager* to start up the container and place the task into it.

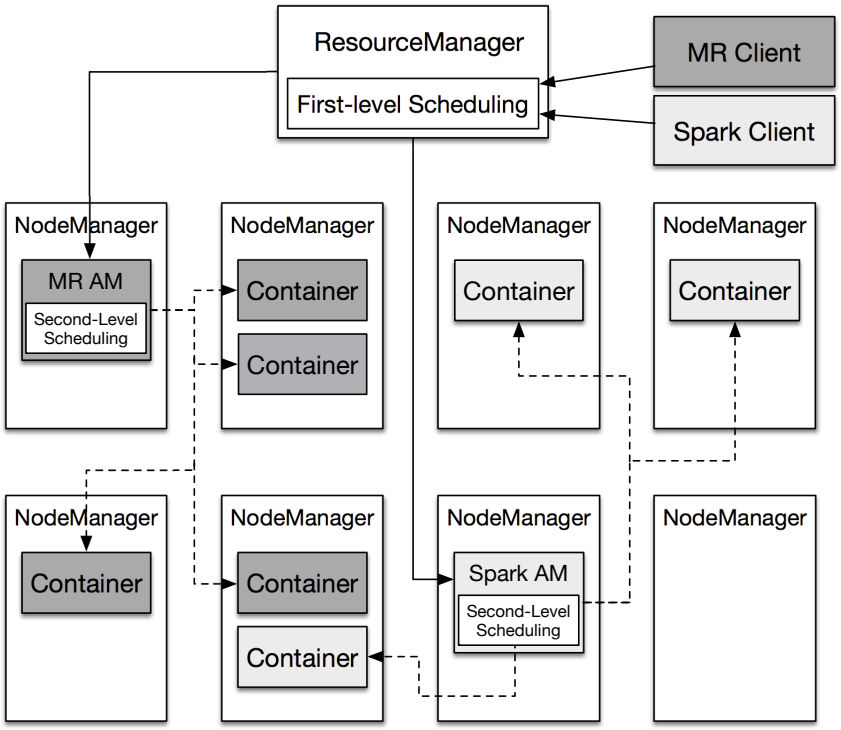

Figure 5.3 – YARN control elements

In step (4) the *ResourceManager* makes a decision about which node to place the container based on the following scheduling policies:

- FIFO (First-in, First-out) policy: for each iteration, a task which fits the residual capacity of the node is pulled from a descending job submission queue and scheduled onto the respective node;
- Fair policy: it considers only the resources usage of each job and attempts to share equal portions of resources with all jobs;
- Capacity policy: it is similar to the Fair policy, but the difference is that it relies on multiple per-organization queues that guarantee the capacity for each job. Within each queue, the tasks are scheduled using a FIFO approach. Based on data obtained from *NodeManager*, the Capacity scheduler can then place containers on the best-suited nodes.

5.3.2 Implementation

We rethink the YARN's components and design the changes in the architecture to integrate with the proposed interference-aware scheduling policy. It is depicted in Figure 5.4. The outer dotted boxes are the YARN's built-in components, while the Interference-aware

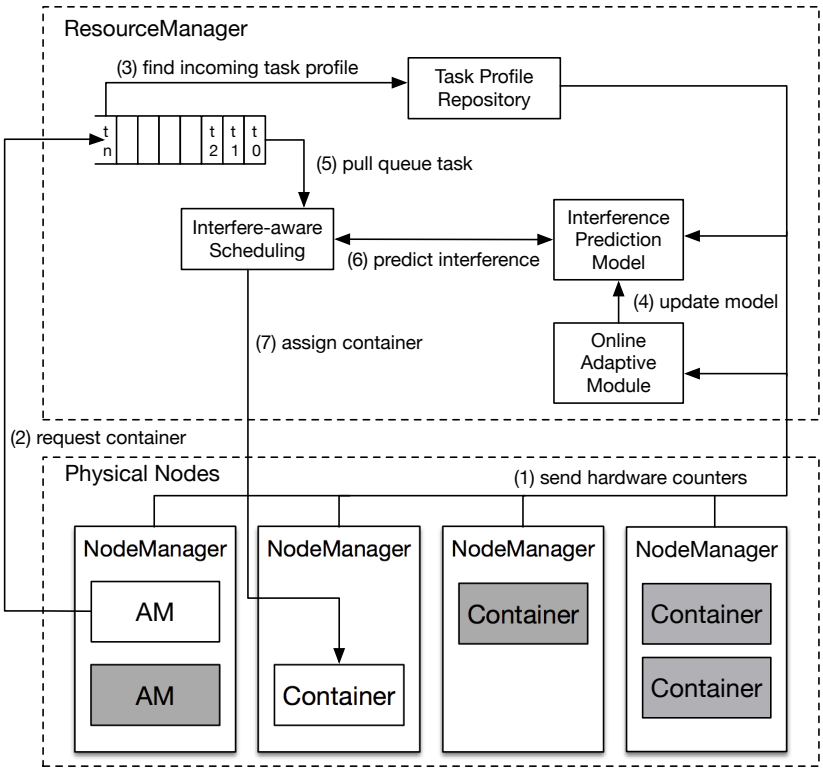

Figure 5.4 – Interference-aware scheduling architecture in YARN

Scheduling, the Task Profiler, the Interference Prediction Model, and the Online Adaptive Module boxes are the new interference scheduling purpose-built components introduced into the *ResourceManager* code. The job submission workflow in YARN with the interference-aware scheduling policy consists of seven steps. First of all, after *NodeManager* starts, it registers itself in *ResourceManager* and periodically sends heartbeat messages delivering the node's status and resource usage, such as CPU and memory, as seen in Figure 5.5. We modified the heartbeat messages to include per-container performance interference counters that is provided by IntP, not just the number of CPUs and amount of memory.

The framework-specific AM organizes its workflow plan and requests resources to the *ResourceManager*; the task is then queued. the task-related profile is fetched from the Task Profile Repository. If it is not found, then the task never ran in the cluster, then the interference prediction will be skipped. Otherwise, the profile is sent as input to the Prediction Model. The Online Adaptive Module checks the model's accuracy based on the received task profile and hardware counters. If the model returns inaccurate results, the model is retrained, and the new re-fitted model is evaluated. By doing so, we hope to achieve high accuracy for any kind of workload. The scheduler pulls out the task from the queue, and then predicts the task performance interference for each node in the cluster. The scheduler assigns the task to the node with the least interference and returns the resources to the AM in the form of a container. The message includes the nodes on which the AM should to start up the container.

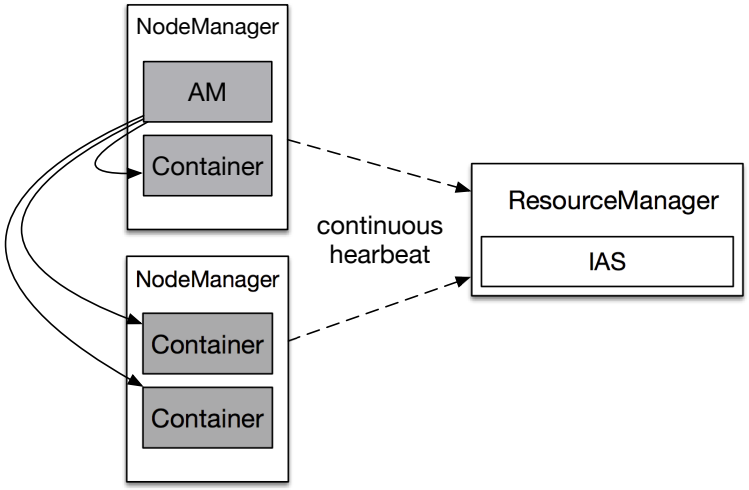

Figure 5.5 – Communication between the ResourceManager and NodeManager processes

5.4 Performance Evaluation

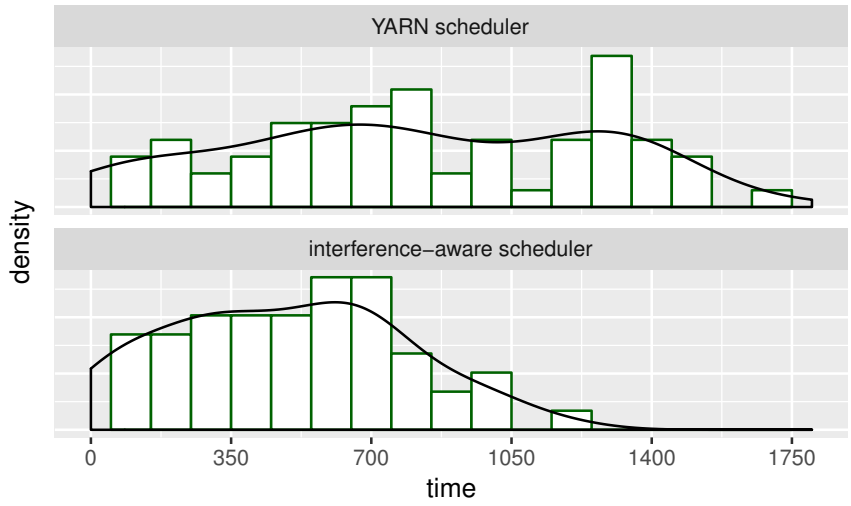

Figure 5.6 – Comparison between the interference-aware scheduler and YARN's default scheduler. Density represents the allocated slots.

We prototype the proposed interference-aware policy in YARN because it enables parallel Big Data application scheduling, and also because we are interested in analyzing the ability of the schedulers to deal with different workload patterns. Results from IntP were used to classify applications using the clustering method, such as presented in Section 4.3.

We selected a set of applications from different frameworks and programming engines to extend heterogeneity, including Hadoop, Spark, and Storm. In addition, we chose the YARN's Fair policy (default installed) to compare it with the proposed interference policy. We used a carefully-crafted external script to connect to the YARN's client API and work like the dispatcher moving jobs every 5 seconds on the 10-in-10 order (no job completion

waiting). The experiment aims to evaluate the jobs' turnaround times (makespan) and the total completion times. The results are shown in Figure 5.6. The graph shows that the reduced job turnaround times reflected on the total completion time, and also improved the efficiency (density), expected when evaluating performance in scheduling. We observed a performance optimization up to 35%, which resulted in about 39% of efficiency.

5.5 Summary

There have been many studies exploring resource contention workarounds through dynamic resource allocation, interference-aware task scheduling, or application parameter tuning to accelerate cluster computing [15, 23, 74, 85]. Hardware and operating system implementations have also been extensively studied in previews works, including dynamic cache partitioning [112], intelligent memory management [71], and improved operating system scheduling [122]. All of these and other works were summarized in Chapter 3, and to the best of our knowledge, none of them have explored performance isolation in data center operating systems. The heterogeneity of workload patterns in data processing make modeling performance interference a challenging task that has led us to explore a research topic that has not yet been addressed. We have seen two other strongly relevant studies in regard to the interference in I/O-bound applications in cloud computing platforms that we believe are important to consolidate the theoretical foundation.

A handful of other studies have proposed interference prediction models for heterogeneous workloads to improve resource provisioning in cloud computing environments. Nathuji et al. [74] proposed Q-cloud to capture performance interference in terms of resource allocation. Q-cloud compensates the interference-related performance degradation by adjusting the processor allocation for an application based on the required SLA. It adds additional computing resources to ensure that the SLA is not violated. Govindan et al. [38] studied memory cache interference in shared on-chip. The interference was predicted based on cache activities. Chiang et al. [23] proposed TRACON, which uses machine learning algorithms for modeling performance interference and to schedule the tasks on the node's slots. Koh et al. [51] proposed a per-application workload vector. The authors then used a clustering algorithm to group applications that are similar. As a result, a new application is first grouped into one of the clusters and then consolidated together with applications that are the least likely to cause performance interference. The algorithm is invoked periodically to obtain the optimal solution. Cucinotta et al. [26] proposed a mechanism for providing temporal isolation based on CPU real time scheduling latencies. The system guarantees computing and networking resources to ensure QoS for virtualized applications. Mars et al. [60] proposed the Bubble-Up methodology that predicts interference-related performance overheads of co-located applications that contend for the memory subsystem. The approach

uses a carefully designed stress app (”bubble”) that puts stress on the memory subsystem while running side by side with the application. It measures the sensibility of the application while sharing the memory subsystem. Delimitrou et al. [29, 30] presented the Paragon scheduler. Paragon derives from robust analytical methods, instead of profiling each application in detail, it leverages information the system already has about applications it has previously seen.

Similarly, we have addressed the interference effects in a resource sharing platform. Our interference-aware scheduling policy is distinctive in the following ways: Instead of focusing on a specific type of hardware resource, our implementation is driven by a holistic view of the workloads across the nodes. Furthermore, our proposed interference model focuses exclusively on resource contention at the OS level because CMS relies on container-based systems for isolation purposes. Finally, we have not see any studies that argue that there are not interference-aware scheduling algorithms in data center operating systems. We observed a performance optimization up to 35%, which resulted in about 39% of efficiency. The results showed that an interference characterization could assist schedulers in allocating resources to Big Data applications that significantly differ in workload patterns, making the cluster more better balanced and resource-efficient, than interference-agnostic clusters.

6. INTP-ASSISTED MULTI-TIER APPLICATION PLACEMENT

The primary objective of this work is to optimize the performance of multi-tier applications using IntP as performance instrumentation. This optimization will be done by generating a better distribution of the applications tiers over the physical machines to minimized resource interference and network bottlenecks. In order to better illustrate this problem, Figure 6.1 shows two placements of the same multi-tier application with two tiers. In placement *a*, the application is suffering from performance degradation due to resource interference, because both tiers are stressing the same resources of a physical machine (represented with the red background). On the other hand, in placement *b*, the application is suffering from performance degradation due to network overhead, because the tiers are placed in different physical machines and have to communicate over the network (represented with the red arrow).

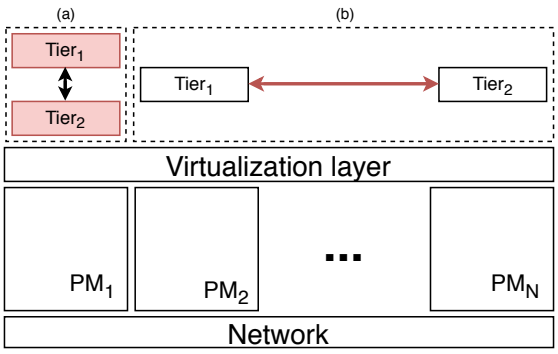

Figure 6.1 – Two placements of the same multi-tier application: both tiers placed in the same physical machine and therefore generating interference (a) vs. tiers placed in two distinct physical machines resulting in communication overhead (b).

The challenge here, and the goal of this work, is not to eliminate performance degradation, but rather to find the placement that leads to the lowest performance degradation possible. We accomplish that with a strategy that repels applications that will stress the same resource from the physical node and attracts two modules to the same multi-tier application that communicate a lot to the same physical node. In this context, we propose placement algorithms based on these policies and evaluate the proposed solutions for different workload scenarios using a visual simulation tool we developed called CIAPA (Capacity, Interference and Affinity-aware Placement Algorithms). CIAPA uses a performance degradation model, a cost function, and heuristics to find a placement with the minimum cost for a specific workload of multi-tier applications.

Our simulation results show that when these two aspects, interference and affinity, are combined in a placement strategy, which is a novel approach, resources are better

utilized, and the performance of the consolidated applications tend to improve, and this is the context and the main contribution of this work.

6.1 Modeling Multi-tier Data Processing Applications

We used IntP to quantify resource contention and infer performance interference among applications. At first glance, it seems a good idea to improve placement decisions using IntP outcomes. However, each resource may suffer from interference in different ways. A high level of disk contention may be much more prejudicial to an application than a high level of CPU contention. Thus, we did not use the interference levels by themselves, but rather the performance degradation a given interference level generates. Hence, we classified interference and affinity levels into four classes for simplification: **Absent, Low, Moderate, and High**. Even though this classification reduces the breadth of the problem, it is still an improvement to the state-of-the-art studies that consider only two levels (absent and present). Each class covers different interference and affinity levels that ranges from 0% to 100% as presented in Table 6.1.

Class	min	max
Absent	0%	0%
Low	1%	20%
Moderate	21%	50%
High	51%	100%

Table 6.1 – Classification of interference and affinity levels.

The higher the class, the greater the interference in performance, as well as the greater the affinity between application's tiers. Since higher levels of interference (80%~100%) are more difficult to achieve for any application being profiled with IntP, an equal distribution would cause most applications to be sorted at smaller intervals. That is why we distribute the levels in a more centralized and balanced way.

6.1.1 IntP-assisted Classifications

The interference-related performance degradation was obtained using a simulated one-tier application from the *node-tiers* benchmark. The stressing tool Artillery [56] was configured with 50 concurrent threads producing HTTP's request bursts to the application during a 40-minute runtime. We collected the average response time while the application runs in isolation. Afterwards, we injected Low, Moderate, and High applications into the same node, and calculated performance overheads using the equation $perf_{class}/perf_{absent}$,

where $perf_{class}$ is the average response time for each interference class, and $perf_{absent}$ is the average response time while running in isolation.

We used a similar approach to model performance degradation related to network affinity. For this case, we used a two-tiers application that put stress only on the network subsystem. From the network communication's point of view, when multiple tiers are co-located, they are running in isolation, which means that the network substrate does not carry any data. Hence, we placed the tiers in different nodes to produce Low, Moderate, and High affinity levels. The affinity levels varied according to the size of the Artillery's messages: 1KB, 128KB, and 256KB that denote the Low, Moderate, and High level, respectively. The performance degradation was calculated using the same method as the interference one, taking into consideration the response time of each class. Finally, we characterized the interference and affinity performance degradation as shown in Table 6.1.1.

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

Table 6.2 – Performance interference generated by resource contention and network affinity.

Let us take a simple usability example for a two-tier application. Tier one fits into the CPU-contention Moderate class, while tier two fits into the CPU-contention High class. Also, tier two fits into the High class of affinity with tier one. In this example, if the tiers would placed in different nodes, they would have no performance interference, but the response time of tier two would increase 1.57 times due to the network overhead. On the other hand, if they would placed into the same node, tier one would increase 1.33 times, while tier two would increase 1.15 times.

Taking these numbers into consideration, we are able to find out the best placement setting with the lowest performance degradation. Or even if we add or multiply the performance degradation generated by interference, the performance degradation generated by network affinity becomes greater; therefore, the best placement setting would be to place both tiers into the same node.

6.1.2 Modeling Placement Costs

One of the key goals of placement algorithms is to consolidate a set of applications or tasks on the fewest nodes to make the data center resource efficient. This study aims to accelerate multi-tier applications by minimizing performance degradation caused by interference and network overheads. Hence, we modeled placement costs considering capacity,

interference and affinity. Table 6.3 summarizes the modeling notations used for the problem formulation throughout this study.

Symbol	Meaning
T'	A set of tiers. A tier is defined as a tuple $T = (I, A, S)$.
I	A tuple of performance degradation generated by interference. The tuple is defined as $I = (cpu, memory, disk, cache)$. These values depend on the tiers' interference levels, and they are taken from the model seen in Table 6.1.1.
A	A set of network affinity between tier T and other tiers in set T'. An element of the set A is defined as $(affinity_i, T_i)$, such that $\{affinity_i, T_i \mid i \in T'\}$. $affinity_i$ is the performance degradation given by the model seen in Table 6.1.1.
S	Size of tier T, where $S > 0$.
I'	Set of interference elements from each tier in a set T'.
A'	Set of affinity elements from each tier in a set T'.
S'	Set of size elements from each tier in a set T'.
P'	A set of PMs that will host a sub-set of tiers. A PM is defined as $P = C$.
C	Capacity of a PM P, where $C > 0$.

Table 6.3 – Notations for the problem formulation.

Capacity constraint. The capacity cost function guarantees that a given node P has enough capacity to host all tiers represented by the size set S'. If P is not able to host all of them, i.e., the sum of tiers sizes is greater than the capacity, it returns a high cost value, defined as ∞, which basically invalidates the configuration. Otherwise, it returns 1, which is a number that when multiplied will not modify the total cost. The capacity constraint function is defined as follows:

$$f_{cap}(S', P) = \begin{cases} \infty & \sum_S, \forall_S \in S' > P_C \\ 1 & otherwise \end{cases} \tag{6.1}$$

Interference cost. The interference cost function returns the total interference cost for a set of consolidated tiers T', represented by their interference set I'. The interference level for each resource is denoted as follows:

$$g(I'_{res}) = \{I \mid I \in I'_{res}, I > 1\} \tag{6.2}$$

Where res denotes $\{CPU, memory, disk, cache\}$.

$$f_{int'}(I'_{res}) = \begin{cases} \prod_{I \in g(I'_{res})} I & |g(I'_{res})| > 1 \\ 1 & otherwise \end{cases} \tag{6.3}$$

The helper function g in Equation 6.2 returns a set of values that are greater than 1, i.e., which cause performance degradation. Furthermore, the function $f_{int'}$ returns the set of values that cause performance degradation for the set of tiers I'. And when more than

one tier contend for same resource, their performance degradation values are multiplied and returned as the cost; otherwise, if there is none or only one tier that contend for a given resource, it returns 1. Finally, the interference cost is given by the multiplication of the cost of each resource, as denoted in Equation 6.4.

$$f_{int}(I') = f_{int'}(I'_{cpu}) * f_{int'}(I'_{memory}) * f_{int'}(I'_{disk}) * f_{int'}(I'_{cache}) \tag{6.4}$$

Affinity cost. The affinity cost function returns the consolidation cost for the set of tiers T', represented by their affinity set A'. It iterates through each affinity entry of each tier, and by calling a helper function, it calculates the total affinity cost. The cost would be 1 (no cost) if the tiers that have affinity are a subset of T', which are placed in the same node. If a tier is not a member of T', then the cost is the multiplication of each performance degradation values when the tiers are placed in different nodes. The helper function, as well as the affinity cost function are denoted in Equation 6.5 and 6.6, respectively.

$$f_{aff'}(a, T') = \begin{cases} 1 & a_T \in T' \\ a_{affinity} & otherwise \end{cases} \tag{6.5}$$

$$f_{aff}(A', T') = \prod_{A \in A'} \prod_{a \in A} f_{aff'}(a, T') \tag{6.6}$$

Given these three cost functions (Capacity, Interference, and Affinity), the Equation 6.7, therefore, denotes a function that returns the total placement cost to consolidate the set of tiers T' into the node P, where f_{cap} works as a constraint function, which measures if the tiers fit a given node.

$$f(T', P) = f_{int}(I') * f_{aff}(A', T') * f_{cap}(S', P) \tag{6.7}$$

In an attempt to minimize the cost of placement by greadly testing the variety of tier consolidation options for each cluster node, the total placement cost is given by the cost average which is calculated for each node. This tries to keep the same cost among the nodes. Otherwise if the costs are multiplied, it could lead to some nodes with low cost, while others with really high cost. Even though this is not completely avoided by using the cost average, it is less likely to happen. The placement minimization function is as follows:

$$p(T', P') = min(avg(f(T'_1, P_1), ..., f(T'_n, P_m)), avg(f(T'_2, P_1), f(T'_3, P_2)), ..., avg(f(T', P_1))) \tag{6.8}$$

The placement function is able to return the best placement configuration. However, it lead us to an optimization for a bin-packing problem which has an NP-hard computa-

tional complexity. To cope with this, the next section presents the placement heuristics that we have tested to address the optimization problem.

6.2 Interference-aware Placement Heuristics

6.2.1 Round Robin Decreasing

Given the high complexity of the placement function, heuristics are an alternative to optimize the problem and find a near-optimal solution in a feasible computational time. Thus, we implemented two optimization-based heuristics for the placement cost function: Stochastic Hill Climbing (SHC) and Simulated Annealing (SA) [54]. The heuristics start from an initial placement configuration point and then iterate many times over the solution by making optimizations to reduce the cost of placement. The initial placement configuration point is generated through the Round Robin Decreasing (RRD) placement algorithm, as shown in Algorithm 6.1. RRD simply sorts the tiers down by size, and afterwards place them on the nodes in a sequential order.

```
Data: P′, T′
T′ = sortDecreasingBySize(T′)
pmIndex = 0
foreach T in T′ do
    P′[pmIndex].push(T)
    pmIndex += 1
    pmIndex = pmIndex % P′.length
end
return newSolution(P′)
```

Algorithm 6.1 – Round Robin Decreasing placement algorithm.

6.2.2 Stochastic Hill Climb

The SHC heuristic pseudo-code is shown in Algorithm 6.2. The algorithm expects three inputs: set of tiers, set of nodes, and number of iterations. After the initial point has been produced using RRD, it attempts to optimize the placement cost at each iteration, generating random changes in the solution. A new solution is considered best if it reduces the current cost. This loop continues for a finite number of iterations. In the end, the solution with the lowest cost is chosen.

Data: P', T', *iterations*
Result: s_{best}
bestSolution = roundRobinDecreasing(P', T')
while *iterations* $>$ 0 **do**
    newSolution = randomize(*bestSolution*)
    if *newSolution.cost* $<$ *bestSolution.cost* **then**
        bestSolution = *newSolution*
    end
    iterations -= 1
end
return *bestSolution*

Algorithm 6.2 – Stochastic Hill Climb heuristic.

6.2.3 Simulated Annealing

SHC is a optimization-based heuristic that returns the minimum local, and in some cases, it might not be the minimum global. This is because it only accepts a new solution if it has a lower cost. However, sometimes it is necessary to take a worse solution in order to improve it later. We implemented the SA heuristic, as it is designed to find the global minimum. The algorithm pseudo-code is described in Algorithm 6.3.

Data: P', T', *temperature*, *coolingRate*
Result: s_{best}
s = roundRobinDecreasing(P', T')
bestSolution = s
while *temperature* $>$ 1 **do**
    newSolution = randomize(s)
    if P(*solution*, *newSolution*, *temperature*) $\geq$ *random*(0, 1) **then**
        solution = *newSolution*
    end
    if *solution.cost* $<$ *bestSolution.cost* **then**
        bestSolution = *solution*
    end
    temperature =* 1 $-$ *coolingRate*
end
return *bestSolution*

Algorithm 6.3 – Simulated Annealing heuristic.

Instead of receiving the number of iterations as SHC does, SA expects the temperature and cooling rate. The temperature starts high, and the cooling rate determines how much it will decrease over each iteration. Furthermore, the main reason behind the use of a temperature is that, when the temperature is high, the acceptance probability function P, which is seen in Equation 6.9, will have higher changes of accepting a worse solution.

$$P(s, s', T) = \begin{cases} 1 & s'_{cost} < s_{cost} \\ exp((s - s')/T) & otherwise \end{cases} \tag{6.9}$$

At first, the algorithm attempts several different solutions, although they are worse. In the end, however, it tends to accept only the best ones.

6.3 Performance Evaluation

In this section we evaluate and analyze the proposed placement algorithms compared to state-of-the-art studies. To the best of our knowledge, there is no other work that simultaneously exploits performance interference and network affinity for multi-tier application placement optimizations, then we select the algorithms proposed by Somani et al. [94] and Su et al. [96] to represent individually an interference-aware placement algorithm and affinity-aware strategy, respectively.

We would like to illustrate two use cases that we intend to analyze:

- **Case I:** two multi-tier applications with high conflict between resource contention and network affinity. One application has two moderate CPU-intensive tiers, with a high network affinity, while the other has two high disk I/O-intensive tiers with low network affinity.
- **Case II:** three multi-tier applications with less conflicts between the tiers within the same application, but high levels of affinity and resource contention between tiers of different applications.

Our hardware setup comprises of two identical nodes equipped with two 2.27GHz processors (with 8 cores each), 8M of L3 cache per core, 64GB of RAM, one 146GB disk and one gigabit network adapter. The application's tiers were limited to four cores and 2GB of RAM. The *node-tiers* benchmark [57] was used to deploy multiple emulated tiers over the cluster. It was forked from the *stress-ng* benchmark suite [50], and enables the emulation of a complete multi-tier application that consumes different computing resources. Moreover, *node-tiers* emulates data communication between the tiers and collects data transfer counters from the network channel.

We executed the experiments for both cases on the testbed. Table 6.4 shows the distribution of tiers per node for the first use case. As expected, the interference strategy has placed applications that interfere with each other in different nodes, while the affinity strategy placed tiers with affinity in the same node. CIAPA, on the other hand, generated a different planning, where three tiers are consolidated in one node. This is because CIAPA seeks the

best trade-off between interference and affinity, producing the placement planning with the lowest cost.

Strategy	node 1	node 2
Interference	#1, #3	#2, #4
Affinity	#1, #2	#3, #4
CIAPA-SA	#1, #2, #4	#3

Table 6.4 – Placement planning for the first use case.

There was a similar placement planning for the second use case, as shown in Table 6.5. As we have noted, placement strategies that only look at one criteria generate a planning with higher cost. Considering interference-awareness, tier #3 suffer performance degradation because it needs to communicate with tier #4, but it was placed in a different node. Considering affinity-awareness, tiers #2 and #4 are two disk I/O-intensive workloads and were placed in the same node; therefore, they may experience a high level of performance interference. On the other hand, CIAPA-SA is interference- and affinity-awareness, and was able to identify both situations and produce a plan with lower performance impacts.

Strategy	node 1	node 2
Interference	#1, #2, #3	#4, #5, #6
Affinity	#1, #2, #3, #4	#5, #6
CIAPA-SA	#1, #2, #5, #6	#3, #4

Table 6.5 – Placement planning for the second use case.

Through these placement planning, we deploy the real multi-tier applications on the cluster. Figure 6.2 depicts the cost produced by each strategy, as well as the performance of the applications for both use cases. CIAPA was not only able to produce a placement with the lowest cost, but it also led to the best overall application's performance. In particular, in the second use case, we can observe a reduction in response time of about 10% when compared to the Interference strategy, and up to 18% when considering only the affinity strategy.

Our preliminary experiments with the placement of data-intensive multi-tier applications have confirmed that resource contention and network affinity are closely related and are key to optimizing performance and resource efficiency. Our contribution to state of the art is that the combination of these two criteria can improve results even more, and a good balance between the two is critical to this improvement.

6.4 Summary

In this chapter, we explored the opportunities to improve performance in multi-tier application placement by focusing on the trade-off between resource contention and net-

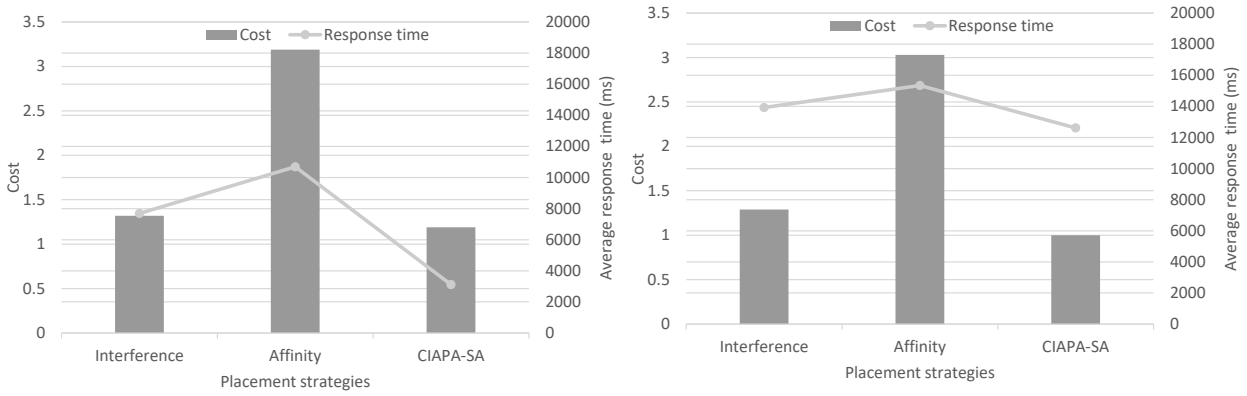

Figure 6.2 – Placement cost of CIAPA and application's average response time compared to state-of-the-art interference- and affinity-aware placement strategies.

work affinity. Despite several research works have been addressing similar studies through placement problem-solving implementations, we presented insights for optimization considering both criteria at the same time. CIAPA comprises a performance degradation model, a cost function, and heuristics to find optimal placement for a specific workload of multi-tier applications. IntP plays an important role within CIAPA. It has been used to perform a instrumentation-based characterization of diverse application's tiers, and its counters are of great importance for classifying the tiers by interference and network affinity that occurs at the preliminary stage of the placement strategy.

These experiments proved that performance interference and network affinity have a high impact on application performance. It has been realized that when these two criteria are combined into one placement strategy, this improves resource efficiency and application performance. We observed a reduction in response time of around 18% when compared to state-of-the-art interference-aware and affinity-aware strategies. The optimization strategy focused on multi-tier applications, but it could be easily extended to cluster-computing—essentially general-purpose—data centers such as those that run Big Data-driven applications.

7. CONCLUSION

We started this dissertation by posing our high level research goal, namely to investigate the hypothesis that an interference-aware data center would improve data-processing application performance when compared to state-of-the-art interference-agnostic data centers. We then refined this high level goal into different research questions, which were addressed by the chapters of this dissertation. We now present our concluding remarks.

7.0.1 Concluding Remarks

Based on the research work presented in this dissertation, our main conclusion is that interference-aware scheduling for container management system is technically feasible and can significantly speed up Big Data analytics and thus reduced time-to-insight. On this basis, we conclude that the contribution of this work to research in the area of Big Data, distributed resource management, and virtualization is comprehensive. We note the main contributions of this work as follows:

- study of the container management systems for Big Data applications and its ability to maintain performance driven by performance evaluations of its internal components, including the distributed file system, data block indexing mechanism, and job scheduling;
- study of performance needs in containerized Big Data applications, driven by performance isolation evaluations;
- study of the state of the art in resource containers and their ability to deal with performance isolation, and evaluation of a constraint-based technique for interference control between Big Data applications.
- proposal of an architecture and prototype for system-level resource contention monitoring to classify application sensitivity to contention with non-intrusion;
- proposal of an architecture, heuristic and prototype of interference-aware container scheduler for Big Data applications with a preliminary workload classification and resource contention understanding;
- study case on interference-aware multi-tier application placement with affinity control for heterogeneous data centers;

7.0.2 Future Research

There are many possible directions for future research based on this work. Firstly, our study of the BigData performance needs and the state of the art in container-based cluster platforms revealed several open issues that could be tackled in future research. Secondly, since this Ph.D. work focused on a subset of these issues, mainly to demonstrate our initial thesis, there are many opportunities to extend this work. Lastly, the toolset developed in this work (namely the IntP) can be used as a base profiling benchmark for conducting new research in this area. We suggest some possibilities for further research as follows.

- Peharps the most natural follow-up to this work is to improve and extend IntP to support a broader spectrum of application classes. So far, we have focused on BigData workloads (e.g., machine learning) and the optimization strategies described in Section 3.5. However, we recognize that this can be further improved and more complex strategies can be tested. Due to its modular architecture and OS layer where it runs, IntP can be used as an internal OS component to test new performance optimization and interference control strategies in the future.

- Our IntP prototype focused on instrument diferent OS layers to extract metrics from BigData workloads. However, given the information availability in other OS subsystems (e.g. energy) and data storage (e.g. HDFS), including these types of information in the interference model is another natural follow-up to this work.

- Through our experiments, we have identified that part of the potential of this work is limited by hypervisor I/O contention. However, such limitation is not expected to exist in next-generation container management systems (e.g. Kubernetes). Therefore, we believe there is a great opportunity in studying the applicability of interference control techniques such as the presented in this work to this new kind of orchestrator systems.

REFERENCES

[1] Abadi, M.; Barham, P.; Chen, J.; Chen, Z.; Davis, A.; Dean, J.; Devin, M.; Ghemawat, S.; Irving, G.; Isard, M.; et al.. “Tensorflow: a system for large-scale machine learning”. In: Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2016, pp. 265–283.

[2] Altair. “Breeze trace”. Source: http://www.ellexus.com/breeze-star-trace/, Jun 2017.

[3] Alves, M. M.; Drummond, L. M. d. A. “A quantitative model for predicting cross-application interference in virtual environments”, *arXiv e-prints*, vol. 1, 2016, pp. arXiv:1610.04309.

[4] Amannejad, Y.; Krishnamurthy, D.; Far, B. “Detecting performance interference in cloud-based web services”. In: Proceedings of the IFIP/IEEE Symposium on Integrated Network and Service Management (IM), 2015, pp. 423–431.

[5] Apache. “Apache hive”. Source: https://hive.apache.org/, Jul 2017.

[6] Bailey, D. H.; Barszcz, E.; Barton, J. T.; Browning, D. S.; Carter, R. L.; Dagum, L.; Fatoohi, R. A.; Frederickson, P. O.; Lasinski, T. A.; Schreiber, R. S.; et al.. “The nas parallel benchmarks”, *The International Journal of Supercomputing Applications*, vol. 5, 1991, pp. 63–73.

[7] Bardhan, S.; Menascé, D. A. “Predicting the effect of memory contention in multi-core computers using analytic performance models”, *IEEE Transactions on Computers*, vol. 64, 2015, pp. 2279–2292.

[8] Barham, P.; Dragovic, B.; Fraser, K.; Hand, S.; Harris, T.; Ho, A.; Neugebauer, R.; Pratt, I.; Warfield, A. “Xen and the art of virtualization”. In: Proceedings of the ACM Symposium on Operating Systems Principles (SOSP), 2003, pp. 164–177.

[9] Bernstein, D. “Containers and cloud: From lxc to docker to kubernetes”, *IEEE Cloud Computing*, vol. 1, 2014, pp. 81–84.

[10] Biederman, E. W. “Multiple instances of the global linux namespaces”. In: Proceedings of the Linux Symposium, 2006, pp. 101–112.

[11] Blagodurov, S.; Fedorova, A.; Vinnik, E.; Dwyer, T.; Hermenier, F. “Multi-objective job placement in clusters”. In: Proceedings of the ACM Conference for High Performance Computing, Networking, Storage and Analysis (SC), 2015, pp. 66.

[12] Blagodurov, S.; Zhuravlev, S.; Fedorova, A.; Kamali, A. “A case for numa-aware contention management on multicore systems”. In: Proceedings of the ACM

Conference on Parallel Architectures and Compilation Techniques (PACT), 2010, pp. 557–558.

[13] Borthakur, D.; Gray, J.; Sarma, J. S.; Muthukkaruppan, K.; Spiegelberg, N.; Kuang, H.; Ranganathan, K.; Molkov, D.; Menon, A.; Rash, S.; et al.. “Apache hadoop goes realtime at facebook”. In: Proceedings of the ACM Conference on Management of Data (SIGMOD), 2011, pp. 1071–1080.

[14] Borthakur, D.; et al.. “Hdfs architecture guide”, *Hadoop Apache Project*, vol. 53, 2008, pp. 1–13.

[15] Bu, X.; Rao, J.; Xu, C.-z. “Interference and locality-aware task scheduling for mapreduce applications in virtual clusters”. In: Proceedings of the ACM Symposium on High-performance Parallel and Distributed Computing (HPDC), 2013, pp. 227–238.

[16] Carbone, P.; Katsifodimos, A.; Ewen, S.; Markl, V.; Haridi, S.; Tzoumas, K. “Apache flink: Stream and batch processing in a single engine”, *The Bulletin of the Technical Committee on Data Engineering*, vol. 38, 2015, pp. 28–38.

[17] Chaisiri, S.; Lee, B.-S.; Niyato, D. “Optimal virtual machine placement across multiple cloud providers”. In: Proceedings of the IEEE Asia-Pacific Services Computing Conference (APSCC), 2009, pp. 103–110.

[18] Chazarain, G. “iotop - simple top-like i/o monitor”. Source: https://linux.die.net/man/1/iotop, Jun 2017.

[19] Chen, C. P.; Zhang, C.-Y. “Data-intensive applications, challenges, techniques and technologies: A survey on big data”, *Information Sciences*, vol. 275, 2014, pp. 314–347.

[20] Chen, K.; Xin, J.; Zheng, W. “Virtualcluster: Customizing the cluster environment through virtual machines”. In: Proceedings of the IEEE/IFIP Conference on Embedded and Ubiquitous Computing (EUC), 2008, pp. 411–416.

[21] Chen, X.; Rupprecht, L.; Osman, R.; Pietzuch, P.; Franciosi, F.; Knottenbelt, W. “Cloudscope: Diagnosing and managing performance interference in multi-tenant clouds”. In: Proceedings of the IEEE Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), 2015, pp. 164–173.

[22] Cheng, Y.; Chen, W.; Wang, Z.; Xiang, Y. “Precise contention-aware performance prediction on virtualized multicore system”, *Journal of Systems Architecture*, vol. 72, 2017, pp. 42–50.

[23] Chiang, R. C.; Huang, H. H. “Tracon: Interference-aware scheduling for data-intensive applications in virtualized environments”. In: Proceedings of the ACM/IEEE Conference for High Performance Computing, Networking, Storage and Analysis (SC), 2011, pp. 47.

[24] Cito, J.; Ferme, V.; Gall, H. C. “Using docker containers to improve reproducibility in software and web engineering research”. In: Proceedings of the ACM/IEEE Conference on Software Engineering Companion (ICSE), 2016, pp. 609–612.

[25] Computing, A. “Torque resource manager”. Source: http://www.clusterresources.com/products/torque-resource-manager.php, Aug 2017.

[26] Cucinotta, T.; Giani, D.; Faggioli, D.; Checconi, F. “Providing performance guarantees to virtual machines using real-time scheduling”. In: Proceedings of the European Conference on Parallel Processing (Euro-Par), 2011, pp. 657–664.

[27] Cunha, J.; Silva, C.; Antunes, M. “Health twitter big data management with hadoop framework”, *Procedia Computer Science*, vol. 64, 2015, pp. 425–431.

[28] Delimitrou, C.; Kozyrakis, C. “ibench: Quantifying interference for datacenter applications”. In: Proceedings of the IEEE Symposium on Workload Characterization (IISWC), 2013, pp. 23–33.

[29] Delimitrou, C.; Kozyrakis, C. “Paragon: Qos-aware scheduling for heterogeneous datacenters”, *Special Interest Group on Programming Languages Notices*, vol. 48–4, 2013, pp. 77–88.

[30] Delimitrou, C.; Kozyrakis, C. “Qos-aware scheduling in heterogeneous datacenters with paragon”, *Transactions on Computer Systems*, vol. 31, 2013, pp. 12.

[31] Delimitrou, C.; Kozyrakis, C. “Quasar: Resource-efficient and qos-aware cluster management”, *Special Interest Group on Programming Languages Notices*, vol. 49–4, 2014, pp. 127–144.

[32] Desfossez, J.; Desnoyers, M.; Dagenais, M. R. “Runtime latency detection and analysis”, *Software: Practice and Experience*, vol. 46, 2016, pp. 1397–1409.

[33] Desnoyers, M. “Linux trace toolkit”. Source: http://www.opersys.com/LTT/index.html, Jul 2017.

[34] Eklov, D.; Black-Schaffer, D.; Hagersten, E. “Statcc: a statistical cache contention model”. In: Proceedings of the Conference Parallel Architectures and Compilation Techniques (PACT), 2010, pp. 551–552.

[35] Fedorova, A.; Blagodurov, S.; Zhuravlev, S. “Managing contention for shared resources on multicore processors”, *Communications of the ACM*, vol. 53, 2010, pp. 49–57.

[36] Furman, M. “Openvz”. Source: http://www.openvz.org, Jul 2017.

[37] Gormley, C.; Tong, Z. “Elasticsearch: The Definitive Guide: A Distributed Real-Time Search and Analytics Engine”. O’Reilly Media, 2015.

[38] Govindan, S.; Liu, J.; Kansal, A.; Sivasubramaniam, A. “Cuanta: quantifying effects of shared on-chip resource interference for consolidated virtual machines”. In: Proceedings of the ACM Symposium on Cloud Computing (SoCC), 2011, pp. 22.

[39] Greenspan, J.; Bulger, B. “MySQL/PHP database applications”. Wiley, 2001.

[40] Helmuth, T. “Wordcount benchmark”. Source: http://wiki.apache.org/hadoop/WordCount, Dec 2017.

[41] 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: Proceedings of the IEEE Symposium on High Performance Computer Architecture (HPCA), 2016, pp. 657–668.

[42] Hindman, B.; Konwinski, A.; Zaharia, M.; Ghodsi, A.; Joseph, A. D.; Katz, R. H.; Shenker, S.; Stoica, I. “Mesos: A platform for fine-grained resource sharing in the data center”. In: Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI, 2011, pp. 22–22.

[43] Honarkhah, M.; Caers, J. “Stochastic simulation of patterns using distance-based pattern modeling”, *Mathematical Geosciences*, vol. 42, 2010, pp. 487–517.

[44] Huang, S.; Huang, J.; Dai, J.; Xie, T.; Huang, B. “The hibench benchmark suite: Characterization of the mapreduce-based data analysis”. In: Proceedings of the IEEE Conference on Data Engineering Workshops (ICDEW), 2010, pp. 41–51.

[45] Intel. “Intel corporation”. Source: http://www.intel.com/, Feb 2017.

[46] Islam, M. S.; Gibson, M.; Muzahid, A. “Fast and qos-aware heterogeneous data center scheduling using locality sensitive hashing”. In: Proceedings of the IEEE Conference on Cloud Computing Technology and Science (CloudCom), 2015, pp. 74–81.

[47] Jain, S. M.; Jain, S. M. “Cgroups”. Springer, 2020.

[48] Jin, H.; Qin, H.; Wu, S.; Guo, X. “Ccap: a cache contention-aware virtual machine placement approach for hpc cloud”, *International Journal of Parallel Programming*, vol. 43, 2015, pp. 403–420.

[49] Kanemasa, Y.; Wang, Q.; Li, J.; Matsubara, M.; Pu, C. “Revisiting performance interference among consolidated n-tier applications: Sharing is better than isolation”. In: Proceedings of the IEEE Conference on Services Computing (SCC), 2013, pp. 136–143.

[50] King, C. I. “Stress-ng”. Source: http://kernel.ubuntu.com/~cking/stress-ng/, Nov 2017.

[51] Koh, Y.; Knauerhase, R. C.; Brett, P.; Bowman, M.; Wen, Z.; Pu, C. “An analysis of performance interference effects in virtual environments.” In: Proceedings of the IEEE Symposium on Performance Analysis of Systems & Software (ISPASS), 2007, pp. 200–209.

[52] Lezcano, D. “Linux containers”. Source: http://lxc.sourceforge.net, Jul 2017.

[53] Li, C.; Ding, C.; Shen, K. “Quantifying the cost of context switch”. In: Proceedings of the Workshop on Experimental Computer Science (ExpCS), 2007, pp. 2.

[54] Lim, A.; Rodrigues, B.; Zhang, X. “A simulated annealing and hill-climbing algorithm for the traveling tournament problem”, *European Journal of Operational Research*, vol. 174, 2006, pp. 1459–1478.

[55] Loukides, M.; Loukides, M. “System Performance Tuning”. O’Reilly Media, 1992.

[56] Ltd, S. O. “Artillery”. Source: https://artillery.io/, Jun 2017.

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

[58] Malewicz, G.; Austern, M. H.; Bik, A. J.; Dehnert, J. C.; Horn, I.; Leiser, N.; Czajkowski, G. “Pregel: a system for large-scale graph processing”. In: Proceedings of the ACM Conference on Management of Data (SIGMOD), 2010, pp. 135–146.

[59] Mars, J.; Tang, L.; Hundt, R.; Skadron, K.; Soffa, M. L. “Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations”. In: Proceedings of the IEEE/ACM Symposium on Microarchitecture (MICRO-44), 2011, pp. 248–259.

[60] Mars, J.; Tang, L.; Skadron, K.; Soffa, M. L.; Hundt, R. “Increasing utilization in modern warehouse-scale computers using bubble-up”, *IEEE Micro*, vol. 32, 2012, pp. 88–99.

[61] Mars, J.; Tang, L.; Soffa, M. L. “Directly characterizing cross core interference through contention synthesis”. In: Proceedings of the Conference on High Performance and Embedded Architectures and Compilers (HiPEAC), 2011, pp. 167–176.

[62] Mars, J.; Vachharajani, N.; Hundt, R.; Soffa, M. L. “Contention aware execution: online contention detection and response”. In: Proceedings of the IEEE/ACM Symposium on Code Generation and Optimization (CGO), 2010, pp. 257–265.

[63] Martella, C.; Shaposhnik, R.; Logothetis, D.; Harenberg, S. “Practical graph analytics with apache giraph”. Springer, 2015.

[64] Matthews, J. N.; Hu, W.; Hapuarachchi, M.; Deshane, T.; Dimatos, D.; Hamilton, G.; McCabe, M.; Owens, J. “Quantifying the performance isolation properties of virtualization systems”. In: Proceedings of the Workshop on Experimental Computer Science (ExpCS), 2007, pp. 6.

[65] McAfee, A.; Brynjolfsson, E.; Davenport, T. H.; Patil, D.; Barton, D. “Big data: the management revolution”, *Harvard Business Review*, vol. 90, 2012, pp. 60–68.

[66] Meng, X.; Pappas, V.; Zhang, L. “Improving the scalability of data center networks with traffic-aware virtual machine placement”. In: Proceedings of the IEEE Conference on Computer Communications (INFOCOM), 2010, pp. 1–9.

[67] Merkel, D. “Docker: lightweight linux containers for consistent development and deployment”, *Linux Journal*, vol. 2014, 2014, pp. 2.

[68] Mogul, J. C.; Borg, A. “The effect of context switches on cache performance”, *Computer Science Division, University of California-Berkeley*, vol. 25, 1991, pp. 75–84.

[69] Mohammed, M.; Khan, M. B.; Bashier, E. B. M. “Machine Learning: Algorithms and Applications”. CRC Press, 2016.

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

[71] Moscibroda, T.; Mutlu, O. “Memory performance attacks: Denial of memory service in multi-core systems”. In: Proceedings of the USENIX Security Symposium on USENIX, 2007, pp. 18.

[72] Murthy, A. C.; Vavilapalli, V. K.; Eadline, D.; Markham, J. “Apache Hadoop YARN: Moving beyond MapReduce and Batch Processing with Apache Hadoop 2”. Addison-Wesley Professional, 2014.

[73] Mutnury, B.; Paglia, F.; Mobley, J.; Singh, G. K.; Bellomio, R. “Quickpath interconnect (qpi) design and analysis in high speed servers”. In: Proceedings of the IEEE Electrical Performance of Electronic Packaging and Systems (EPEPS), 2010, pp. 265–268.

[74] Nathuji, R.; Kansal, A.; Ghaffarkhah, A. “Q-clouds: managing performance interference effects for qos-aware clouds”. In: Proceedings of the European Conference on Computer Systems (EuroSys), 2010, pp. 237–250.

[75] Neuwirth, S.; Wang, F.; Oral, S.; Vazhkudai, S.; Rogers, J.; Bruening, U. “Using balanced data placement to address i/o contention in production environments”. In: Proceedings of the Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), 2016, pp. 9–17.

[76] Nolé, M.; Sartiani, C. “Processing regular path queries on giraph”. In: Proceedings of the Conference on Extending Database Technology (EDBT/ICDT), 2014, pp. 3.

[77] Norcott, W. D. “Iozone”. Source: http://www.iozone.org, Jul 2017.

[78] Novakovic, D. M.; Vasic, N.; Novakovic, S.; Kostic, D.; Bianchini, R. “Deepdive: Transparently identifying and managing performance interference in virtualized environments”. In: Proceedings of the USENIX Annual Technical Conference (ATC), 2013, pp. 219–230.

[79] O’Malley, O. “Terabyte sort on apache hadoop”, *Yahoo*, vol. 1, 2008, pp. 1–3.

[80] Pahl, C.; Brogi, A.; Soldani, J.; Jamshidi, P. “Cloud container technologies: a state-of-the-art review”, *IEEE Transactions on Cloud Computing*, vol. 7, 2017, pp. 677–692.

[81] Potter, K. H. “Dynamic addressing mapping to eliminate memory resource contention in a symmetric multiprocessor system”. US Patent 6,505,269, Jan 2003.

[82] Preeth, E.; Mulerickal, F. J. P.; Paul, B.; Sastri, Y. “Evaluation of docker containers based on hardware utilization”. In: Proceedings of the Conference on Control Communication & Computing India (ICCC), 2015, pp. 697–700.

[83] Pu, X.; Liu, L.; Mei, Y.; Sivathanu, S.; Koh, Y.; Pu, C.; Cao, Y. “Who is your neighbor: Net i/o performance interference in virtualized clouds”, *IEEE Transactions on Services Computing*, vol. 6, 2013, pp. 314–329.

[84] Quintero, D.; Bolinches, L.; Sutandyo, A. G.; Joly, N.; Katahira, R. T.; et al.. “IBM data engine for Hadoop and Spark”. IBM Redbooks, 2016.

[85] Rao, J.; Bu, X.; Xu, C.-Z.; Wang, L.; Yin, G. “Vconf: a reinforcement learning approach to virtual machines auto-configuration”. In: Proceedings of the Conference on Autonomic Computing (ICAC), 2009, pp. 137–146.

[86] Rao, J.; Wang, K.; Zhou, X.; Xu, C.-Z. “Optimizing virtual machine scheduling in numa multicore systems”. In: Proceedings of the IEEE Symposium on High Performance Computer Architecture (HPCA2013), 2013, pp. 306–317.

[87] Regola, N.; Ducom, J.-C. “Recommendations for virtualization technologies in high performance computing”. In: Proceedings of the IEEE Conference on Cloud Computing Technology and Science (CloudCom), 2010, pp. 409–416.

[88] Saha, B.; Shah, H.; Seth, S.; Vijayaraghavan, G.; Murthy, A.; Curino, C. “Apache tez: A unifying framework for modeling and building data processing applications”. In: Proceedings of the ACM Conference on Management of Data (SIGMOD), 2015, pp. 1357–1369.

[89] Sasaki, H.; Buyuktosunoglu, A.; Vega, A.; Bose, P. “Mitigating power contention: a scheduling based approach”, *IEEE Computer Architecture Letters*, vol. 16, 2016, pp. 60–63.

[90] Sheikhalishahi, M.; Grandinetti, L.; Wallace, R. M.; Vazquez-Poletti, J. L. “Autonomic resource contention-aware scheduling”, *Software: Practice and Experience*, vol. 45, 2015, pp. 161–175.

[91] Slominski, A.; Muthusamy, V.; Khalaf, R. “Building a multi-tenant cloud service from legacy code with docker containers”. In: Proceedings of the IEEE Conference on Cloud Engineering (IC2E), 2015, pp. 394–396.

[92] Snell, Q. O.; Mikler, A. R.; Gustafson, J. L. “Netpipe: A network protocol independent performance evaluator”. In: Proceedings of the IASTED Conference on Intelligent Information Management and Systems (IASTED), 1996, pp. 49.

[93] Snir, M.; Otto, S. W.; Walker, D. W.; Dongarra, J.; Huss-Lederman, S. “MPI: the complete reference”. MIT press, 1995.

[94] Somani, G.; Khandelwal, P.; Phatnani, K. “Vupic: Virtual machine usage based placement in iaas cloud”, *arXiv e-prints*, vol. 1, 2012, pp. 1.

[95] Stonebraker, M.; Rowe, L. A. “The design of Postgres”. ACM Sigmod Record, 1986.

[96] Su, K.; Xu, L.; Chen, C.; Chen, W.; Wang, Z. “Affinity and conflict-aware placement of virtual machines in heterogeneous data centers”. In: Proceedings of the IEEE Symposium on Autonomous Decentralized Systems (ISADS), 2015, pp. 289–294.

[97] Tanenbaum, A. “Modern Operating Systems”. Pearson Prentice Hall, 2008.

[98] Tillenius, M.; Larsson, E.; Badia, R. M.; Martorell, X. “Resource-aware task scheduling”, *ACM Transactions on Embedded Computing Systems*, vol. 14, 2015, pp. 5.

[99] Toshniwal, A.; Taneja, S.; Shukla, A.; Ramasamy, K.; Patel, J. M.; Kulkarni, S.; Jackson, J.; Gade, K.; Fu, M.; Donham, J.; et al.. “Storm@twitter”. In: Proceedings of the ACM Conference on Management of data (SIGMOD), 2014, pp. 147–156.

[100] Urgaonkar, B.; Shenoy, P.; Roscoe, T. “Resource overbooking and application profiling in shared hosting platforms”, *Operating Systems Review*, vol. 36, 2002, pp. 239–254.

[101] Vallone, J.; Birke, R.; Chen, L. Y.; Falsafi, B. “Contention detection by throttling: A black-box on-line approach”. In: Proceedings of the IEEE Symposium on Quality of Service (IWQoS), 2015, pp. 237–242.

[102] Vavilapalli, V. K.; Murthy, A. C.; Douglas, C.; Agarwal, S.; Konar, M.; Evans, R.; Graves, T.; Lowe, J.; Shah, H.; Seth, S.; et al.. “Apache hadoop yarn: Yet another resource negotiator”. In: Proceedings of the ACM Symposium on Cloud Computing (SOCC), 2013, pp. 5.

[103] Verma, A.; Cherkasova, L.; Campbell, R. H. “Aria: automatic resource inference and allocation for mapreduce environments”. In: Proceedings of the ACM Conference on Autonomic Computing (ICAC), 2011, pp. 235–244.

[104] Walters, J. P.; Chaudhary, V.; Cha, M.; Guercio Jr., S.; Gallo, S. “A comparison of virtualization technologies for hpc”. In: Proceedings of the IEEE Conference on Advanced Information Networking and Applications (AINA), 2008, pp. 861–868.

[105] Wang, G.; Koshy, J.; Subramanian, S.; Paramasivam, K.; Zadeh, M.; Narkhede, N.; Rao, J.; Kreps, J.; Stein, J. “Building a replicated logging system with apache kafka”, *Proceedings of the VLDB Endowment*, vol. 8, 2015, pp. 1654–1655.

[106] White, T. “Hadoop: the definitive guide: the definitive guide”. O’Reilly Media, 2012.

[107] Xavier, M.; Neves, M.; Rossi, F.; Ferreto, T.; Lange, T.; De Rose, C. “Performance evaluation of container-based virtualization for high performance computing environments”. In: Proceedings of the Euromicro Conference on Parallel, Distributed, and Network-Based Processing (PDP), 2013, pp. 233–240.

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

[109] Xavier, M. G.; N, C. A. F. D. “A performance isolation analysis of disk-intensive workloads on container-based clouds”. In: Proceedings of the Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP), 2015, pp. 253–260.

[110] Xavier, M. G.; Neves, M. V.; Rose, C. A. F. D. “A performance comparison of container-based virtualization systems for mapreduce clusters”. In: Proceedings of the Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP), 2014, pp. 299–306.

[111] Xavier, M. G.; Neves, M. V.; Rossi, F. D.; Ferreto, T. C.; Lange, T.; De Rose, C. A. “Performance evaluation of container-based virtualization for high performance computing environments”. In: Proceedings of the Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP), 2013, pp. 233–240.

[112] Xie, Y.; Loh, G. H. “Pipp: Promotion/insertion pseudo-partitioning of multi-core shared caches”, *Special Interest Group on Computer Architecture News*, vol. 37–3, 2009, pp. 174–183.

[113] Xu, J.; Fortes, J. A. “Multi-objective virtual machine placement in virtualized data center environments”. In: Proceedings of the Conference on Green Computing and Communications & Cyber, Physical and Social Computing (GreenCom-CPSCom), 2010, pp. 179–188.

[114] Yang, H.; Breslow, A.; Mars, J.; Tang, L. “Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers”, *Special Interest Group on Computer Architecture News*, vol. 41–3, 2013, pp. 607–618.

[115] Yang, L.; Dai, Y.; Zhang, B. “Mapreduce scheduler by characterizing performance interference”, *China Communications*, vol. 13, 2016, pp. 253–262.

[116] Zaharia, M.; Chowdhury, M.; Franklin, M. J.; Shenker, S.; Stoica, I. “Spark: cluster computing with working sets”. In: Proceedings of the Workshop on Hot Topics in Cloud Computing (HotCloud), 2010, pp. 10–10.

[117] Zaharia, M.; Konwinski, A.; Joseph, A. D.; Katz, R. H.; Stoica, I. “Improving mapreduce performance in heterogeneous environments”. In: Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI), 2008, pp. 7.

[118] Zhang, W. “Linux vserver”. Source: http://linux-vserver.org, Aug 2017.

[119] Zhang, X.; Tune, E.; Hagmann, R.; Jnagal, R.; Gokhale, V.; Wilkes, J. “Cpi 2: Cpu performance isolation for shared compute clusters”. In: Proceedings of the European Conference on Computer Systems (EuroSys), 2013, pp. 379–391.

[120] Zhao, J.; Cui, H.; Xue, J.; Feng, X. “Predicting cross-core performance interference on multicore processors with regression analysis”, *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, 2016, pp. 1443–1456.

[121] Zhu, Q.; Tung, T. “A performance interference model for managing consolidated workloads in qos-aware clouds”. In: Proceedings of the IEEE Conference on Cloud Computing (CLOUD), 2012, pp. 170–179.

[122] Zhuravlev, S.; Blagodurov, S.; Fedorova, A. “Addressing shared resource contention in multicore processors via scheduling”, *Special Interest Group on Programming Languages Notices*, vol. 45–3, 2010, pp. 129–142.

[123] Zikopoulos, P.; Eaton, C.; et al.. “Understanding big data: Analytics for enterprise class hadoop and streaming data”. McGraw-Hill Osborne Media, 2011.

Pontifícia Universidade Católica do Rio Grande do Sul
Pró-Reitoria de Pesquisa e Pós-Graduação
Av. Ipiranga, 6681 – Prédio 1 – Térreo
Porto Alegre – RS – Brasil
Fone: (51) 3320-3513
E-mail: propesq@pucrs.br
Site: www.pucrs.br