Automatic Rescaling and Tuning of Big Data Applications on Container-based Virtual Environments

---

Jonatan Enes Álvarez

Doctoral Thesis

July 2020

PhD Advisors:

Roberto Rey Expósito

Juan Touriño Domínguez

PhD Program in Information Technology Research

UNIVERSIDADE DA CORUÑA

Dr. Roberto Rey Expósito
Profesor Ayudante Doctor
Dpto. de Ingeniería de Computadores
Universidade da Coruña

Dr. Juan Touriño Domínguez
Catedrático de Universidad
Dpto. de Ingeniería de Computadores
Universidade da Coruña

CERTIFICAN

Que la memoria titulada "*Automatic Rescaling and Tuning of Big Data Applications on Container-based Virtual Environments*" ha sido realizada por D. Jonatan Enes Álvarez bajo nuestra dirección en el Departamento de Ingeniería de Computadores de la Universidade da Coruña, y concluye la Tesis Doctoral que presenta para optar al grado de Doctor en Ingeniería Informática con la Mención de Doctor Internacional.

En A Coruña, a 6 de Julio de 2020

Fdo.: Roberto Rey Expósito
Director de la Tesis Doctoral

Fdo.: Juan Touriño Domínguez
Director de la Tesis Doctoral

Fdo.: Jonatan Enes Álvarez
Autor de la Tesis Doctoral

A Cloe, mi Tata, mi mama y mi novia,
luces en mi camino.

Acknowledgments

First and foremost, I would like to thank my advisors, Juan and Roberto, for their dedication and guidance throughout this Thesis. They are the best bosses you can ask for, and Roberto, you are the real MVP with your patience. I have to also thank my colleagues in the GAC lab, especially Jorge for those much needed lunch breaks.

I am thankful to my family, especially my parents and my beloved sister Noelia, who always supported me in becoming a better, happier person. I have to also thank my girlfriend, Vicky, the only person with whom I can spend every hour of every day, and still love her increasingly more. To my dear niece: Cloe, I am sure that you will be a strong girl, I just hope you grow up to prosper in a better world than the one we are shamefully creating. Lastly, Black!, who's a good dog? You are!

I would also like to thank Dr. Romain Rouvoy and his group for hosting me during my three-month research visit to the Inria Lille research center, and for the access to the Grid'5000 computing platform. They welcomed me with open arms and I had a really nice stay in France. I must also thank the CESGA Supercomputing Center and Dr. Javier Cacheiro for his technical assistance and the access to the Big Data computing platform.

Finally, I want to acknowledge the following funders of this work: the Computer Architecture Group, the Department of Computer Engineering, and the University of A Coruña for the human and material support; the NESUS network under the COST Action IC1305; the Galician Government (refs. ED431G/01, ED431D R2016/045, ED431C 2017/04 and ED431G 2019/01); and the Spanish Government (refs. TIN2016-75845-P, FPU grant FPU15/03381 and mobility grant EST18/00091).

Jonatan Enes Álvarez

Resumo

As aplicacións Big Data actuais evolucionaron dun xeito significativo, dende fluxos de traballo baseados en procesamento por lotes ata outros máis complexos que poden requirir múltiples etapas de procesamento usando diferentes tecnoloxías, e mesmo executándose en tempo real. Doutra banda, para despregar estas aplicacións, os clusters 'commodity' foron substituídos nalgúns casos por paradigmas máis flexibles como o Cloud, ou mesmo por outros emerxentes como a computación 'serverless', precisando ambos paradigmas de tecnoloxías de virtualización. Esta Tese propón dúas contornas que proporcionan modos alternativos de realizar unha análise en profundidade e unha mellor xestión dos recursos de aplicacións Big Data despregadas en contornas virtuais baseadas en contedores software. Por unha banda, a contorna BDWatchdog permite realizar unha análise de gran fino e en tempo real en termos do uso dos recursos do sistema e do perfilado do código. Doutra banda, descríbese unha contorna para o reescalado dinámico e en tempo real dos recursos segundo un conxunto de políticas configurables. A primeira política proposta céntrase no reescalado automático dos recursos dos contedores segundo o uso real que as aplicacións fan dos mesmos, proporcionando así unha contorna 'serverless'. Ademais, preséntase unha política alternativa centrada na xestión enerxética que permite implementar os conceptos de limitación e presuposto de potencia, que poden aplicarse a contedores, aplicacións ou mesmo usuarios. En xeral, as contornas propostas nesta Tese tratan de poñer de relevo o potencial de aplicar novos xeitos de analizar e axustar os recursos das aplicacións Big Data despregadas en clusters de contedores, mesmo en tempo real. Os casos de uso presentados son exemplos diso, demostrando que as aplicacións Big Data poden adaptarse a novas tecnoloxías ou paradigmas sen teren que cambiar as súas características máis intrínsecas.

Resumen

Las aplicaciones Big Data actuales han evolucionado de forma significativa, desde flujos de trabajo basados en procesamiento por lotes hasta otros más complejos que pueden requerir múltiples etapas de procesamiento usando distintas tecnologías, e incluso ejecutándose en tiempo real. Por otra parte, para desplegar estas aplicaciones, los clusters 'commodity' se han reemplazado en algunos casos por paradigmas más flexibles como el Cloud, o incluso por otros emergentes como la computación 'serverless', requiriendo ambos paradigmas de tecnologías de virtualización. Esta Tesis propone dos entornos que proporcionan formas alternativas de realizar un análisis en profundidad y una mejor gestión de los recursos de aplicaciones Big Data desplegadas en entornos virtuales basados en contenedores software. Por un lado, el entorno BDWatchdog permite realizar un análisis de grano fino y en tiempo real en lo que respecta a la monitorización de los recursos del sistema y al perfilado del código. Por otro lado, se describe un entorno para el reescalado dinámico y en tiempo real de los recursos de acuerdo a un conjunto de políticas configurables. La primera política propuesta se centra en el reescalado automático de los recursos de los contenedores de acuerdo al uso real que las aplicaciones hacen de los mismos, proporcionando así un entorno 'serverless'. Además, se presenta una política alternativa centrada en la gestión energética que permite implementar los conceptos de limitación y presupuesto de potencia, pudiendo aplicarse a contenedores, aplicaciones o incluso usuarios. En general, los entornos propuestos en esta Tesis tratan de resaltar el potencial de aplicar nuevas formas de analizar y ajustar los recursos de las aplicaciones Big Data desplegadas en clusters de contenedores, incluso en tiempo real. Los casos de uso que se han presentado son ejemplos de esto, demostrando que las aplicaciones Big Data pueden adaptarse a nuevas tecnologías o paradigmas sin tener que cambiar su características más intrínsecas.

Abstract

Current Big Data applications have significantly evolved from its origins, moving from mostly batch workloads to more complex ones that may involve many processing stages using different technologies or even working in real time. Moreover, to deploy these applications, commodity clusters have been in some cases replaced in favor of newer and more flexible paradigms such as the Cloud or even emerging ones such as serverless computing, usually involving virtualization techniques. This Thesis proposes two frameworks that provide alternative ways to perform in-depth analysis and improved resource management for Big Data applications deployed on virtual environments based on software containers. On the one hand, the BDWatchdog framework is capable of performing real-time, fine-grain analysis in terms of system resource monitoring and code profiling. On the other hand, a framework for the dynamic and real-time scaling of resources according to several tuning policies is described. The first proposed policy revolves around the automatic scaling of the containers' resources according to the real usage of the applications, thus providing a serverless environment. Furthermore, an alternative policy focused on energy management is presented in a scenario where power capping and budgeting functionalities are implemented for containers, applications or even users. Overall, the frameworks proposed in this Thesis aim to showcase how novel ways of analyzing and tuning the resources given to Big Data applications in container clusters are possible, even in real time. The supported use cases that were presented are examples of this, and show how Big Data applications can be adapted to newer technologies or paradigms without having to lose their distinctive characteristics.

Preface

Big Data applications have evolved from its origins and can be potentially demanding in all of the system resources they use, from CPU and memory to energy consumed. In addition, they have moved from the commodity clusters where these applications were first typically deployed to virtual counterparts that can span to hundreds of instances, either virtual machines or software containers.

The present Thesis, "Automatic Rescaling and Tuning of Big Data Applications on Container-based Virtual Environments", addresses the need for novel ways of resource analysis and management for Big Data applications and frameworks on those environments. While from a passive stance the resources are accurately monitored and the applications profiled with low overhead, it is also possible to take an active stance and proactively change the way resources are handled on such virtual clusters. This active stance enables to implement different tuning policies to deploy multiple interesting scenarios, such as a serverless environment for containers, where applications have their resources scaled according to their real usage, or scenarios where resources such as energy can be capped and budgeted as desired. Moreover, both passive and active approaches are implemented to work in real time in order to grant scalability and flexibility to Big Data applications and frameworks.

Objectives and Work Methodology

The main objectives of this Thesis are listed below, including some relevant sub-goals that must also be achieved.

1. Design and implementation of a framework for the in-depth analysis of Big

Data workloads.

- Container-based virtual environments must be efficiently supported
- Fine-grain, per-process resource monitoring
- Low-level code profiling using flame graphs
- Real-time support and low overhead

2. Development of a resource scaling framework for Big Data workloads deployed on container clusters.
   - Real-time and automatic resource scaling features
   - High responsiveness to minimize overheads
   - The design must support working with different tuning policies
3. Implementation of a specific policy for the scaling framework to provide energy capping and power budgeting features.
   - The budgeting must support static and dynamic energy limits
   - Budgeting for both users and applications must be provided

The first two objectives were fulfilled through the development of two specifically designed frameworks that implement all the features required by each sub-goal. On the one hand, the first framework revolves around a passive point of view where the system resources and applications are analyzed to extract valuable information as fine-grain as possible, while also being able to work in real time and with low overhead. For this objective, it was first needed to acquire a deep understanding into how to deploy container-based virtual clusters on top of which Big Data applications can be deployed successfully and efficiently. For this purpose, a Platform-as-a-Service (PaaS) architecture for containers was developed. On the other hand, the second framework switches to an active approach by managing and modifying dynamically the system environment on which Big Data applications are executed. In this case, there was a need to analyze how those applications react when their underlying execution environment is constantly changing in terms of the available resources (e.g., CPU, memory). Overall, both frameworks were developed following a loose agile methodology, where key features were discovered, designed, implemented and

improved on the fly in order to provide experimental testbeds to be later assessed. This methodology was also in part possible thanks to all the underlying technologies used such as time series and NoSQL databases as well as high-level languages such as Python, which overall gave way to fast prototyping. Finally, for the third objective, the two aforementioned frameworks were combined, extended and integrated with a third-party tool in order to work with energy as another resource.

It is worth mentioning that the three objectives concluded in their last stages with usable tools that are publicly available. These tools have been evaluated extensively by running representative Big Data workloads on container clusters in order to obtain all the experimental results presented in this Thesis.

Funding and Technical Means

The means that were used to carry out this Thesis have been the following:

- Working material, human and financial support primarily by the Computer Architecture Group of the University of A Coruña, along with a Research Fellowship funded by the Ministry of Education of Spain (FPU program, ref. FPU15/03381).
- Access to bibliographical material through the library of the University of A Coruña.
- Additional funding through the following research projects:
    - European funding: project "Network For Sustainable Ultrascale Computing" (NESUS COST Action ref. IC1305).
    - State funding by the Ministry of Science and Innovation of Spain through the project "New Challenges in High Performance Computing: from Architectures to Applications (II)" (ref. TIN2016-75845-P).
    - Regional funding by the Galician Government (Xunta de Galicia) under the Consolidation Program of Competitive Reference Groups (Computer Architecture Group, ref. ED431C 2017/04), the Network of Cloud and

Big Data Technologies for HPC (ref. ED431D R2016/045), and the Center for Information and Communications Technology Research (CITIC accreditations 2016-2019 and 2019-2022, refs. ED431G/01 and ED431G 2019/01).

- Access to computing facilities:
  - *CESGA Big Data* platform (Galicia Supercomputing Center, CESGA, Spain). Several nodes have been used to deploy and evaluate multiple scenarios presented in this Thesis. Overall, this infrastructure consists of 38 nodes, each one equipped with 2 Intel Xeon Haswell-EP processors (12 cores per node), 64 GiB of DDR4 memory and 12 HDD disks of 2 TiB. The nodes are interconnected via 2 ×10 Gbps Ethernet.
  - *Grid'5000* infrastructure (Inria, CNRS, RENATER and several French Universities). For the experiments of the Thesis, 12 nodes from 2 different clusters have been used. From the first cluster, 8 nodes equipped with 2 Intel Xeon Skylake-SP processors (24 cores per node), 192 GiB of DDR4 memory, 2 SSD disks of 480 GiB and 4 HDD disks of 4 TiB, interconnected via 2 ×10 Gbps Ethernet. From the second cluster, 4 nodes equipped with 2 Intel Xeon Broadwell-EP processors (28 cores per node), 768 GiB of DDR4 memory, 2 SSD disks of 400 GiB and 2 HDD disks of 4 TiB, also interconnected via 2 ×10 Gbps Ethernet.
- A three-month research visit to the Spirals Group at the Université de Lille/Inria, France, which granted access to the Grid'5000 infrastructure for experimentation. This research visit was funded by the Ministry of Education of Spain through a competitive grant within the FPU program (ref. EST18/00091).

Structure of the Thesis

The Thesis is organized as follows:

- Chapter 1 first introduces the evolution of Big Data technologies from being prominently disk-intensive and deployed on commodity clusters to currently also being CPU- and memory-intensive, as well as using infrastructure

paradigms such as the Cloud and execution environments based on virtual machines or containers. This evolution motivated this Thesis into studying how to implement a deep yet scalable way to analyze and manage system resources across container-based clusters.

- Chapter 2 describes the first proposed framework, BDWatchdog, which serves as the basis for providing other frameworks, tools or users with accurate information on how Big Data applications are using the resources they are given when deployed on a cluster of containers. This information comprises both real-time resource time series and code profiling. The overall architecture of BDWatchdog along with practical use cases are presented in detail.

- Chapter 3 introduces the second proposal of this Thesis, a real-time resource scaling framework designed for container-based environments. This framework, which relies on BDWatchdog for obtaining the resource metrics, adjusts the resources given to Big Data applications running on the cluster according to configurable policies. The main use case described in this chapter dynamically adjusts the resources according to the real usage, thus creating a serverless environment for containers. Several workloads have been assessed to prove that, if the framework adapts quickly, their resource utilization can be greatly increased with minor overheads.

- Chapter 4 presents a specific scenario where the energy consumption of Big Data workloads is studied in detail by combining both previous frameworks from Chapters 2 and 3 together with an external tool. In this scenario, energy capping and power budgets are introduced to demonstrate that energy can be managed in a similar way as other system resources such as CPU or memory. Several Big Data use cases are presented along with backing experiments.

- Chapter 5 concludes the Thesis with some final remarks and proposes additional research lines that are worth considering as future work.

Main Contributions

The main original contributions derived from the Thesis are the following:

- Development of a PaaS architecture to host Big Data applications on container clusters. These containers feature fully working environments to emulate virtual machines but benefiting from faster deployments and lower resource overheads [41].
- Design and implementation of the BDWatchdog framework for the resource analysis of Big Data applications via real-time resource monitoring and code profiling [42].
- Development of a framework to provide a serverless environment for containers, where resources are automatically scaled according to their real usage [43].
- Deployment and evaluation of a specific scenario for the management of energy caps and power budgets both on users and applications [44].

Developed software

The following software tools developed in this Thesis are publicly available:

- BDWatchdog framework for real-time monitoring and profiling of Big Data workloads. Available at `http://bdwatchdog.dec.udc.es`.
- Serverless platform for real-time resource scaling on container clusters. Available at `http://bdwatchdog.dec.udc.es/serverless`.
- Specifications and additional information on the energy scenario. Available at `http://bdwatchdog.dec.udc.es/energy`.

Registered software

Two software products have been registered in the IP registry as outcomes of this Thesis:

- Jonatan Enes, Roberto R. Expósito, and Juan Touriño. BDWatchdog: real-time monitoring and profiling tool for Big Data workloads, October 2019.

Record entry number: 03/2020/180. Owning entity: Universidade da Coruña. Priority country: Spain. In exploitation by Torus Software Solutions S.L. through contract INV11419 since 01/11/2019.

- Jonatan Enes, Roberto R. Expósito, and Juan Touriño. Real-time resource scaling platform for containers in serverless environments, July 2020. Record entry number: pending. Owning entity: Universidade da Coruña. Priority country: Spain.

Publications from the Thesis

Journal publications

- Jonatan Enes, Javier López Cacheiro, Roberto R. Expósito, and Juan Touriño. Big Data-oriented PaaS architecture with disk-as-a-resource capability and container-based virtualization. *Journal of Grid Computing*, 16(4):587–605, 2018. JCR Q1.

- Jorge Veiga, Jonatan Enes, Roberto R. Expósito, and Juan Touriño. BDEv 3.0: energy efficiency and microarchitectural characterization of Big Data processing frameworks. *Future Generation Computer Systems*, 86:565–581, 2018. JCR Q1 (first decile).

- Jonatan Enes, Roberto R. Expósito, and Juan Touriño. BDWatchdog: real-time monitoring and profiling of Big Data applications and frameworks. *Future Generation Computer Systems*, 87:420–437, 2018. JCR Q1 (first decile).

- Jonatan Enes, Roberto R. Expósito, and Juan Touriño. Real-time resource scaling platform for Big Data workloads on serverless environments. *Future Generation Computer Systems*, 105:361–379, 2020. JCR Q1 (first decile).

- Jonatan Enes, Roberto R. Expósito, Javier López Cacheiro, and Juan Touriño. Anomaly detection in HPC systems through job classification using time series and machine learning. 2020. (Submitted for journal publication).

International conferences

- Jonatan Enes, Guillaume Fieni, Roberto R. Expósito, Romain Rouvoy, and Juan Touriño. Power budgeting of Big Data applications in container-based clusters. 2020. (Submitted for conference publication).

Other minor publications

- Jonatan Enes, Roberto R. Expósito, and Juan Touriño. Towards smart and automatic optimization for Big Data: real-time application monitoring and profiling. In *3rd NESUS Winter School & PhD Symposium*. Zagreb, Croatia, 2018.

Contents

List of Tables

List of Figures

Listings

Chapter 1

Introduction

Big Data analysis has been constantly evolving and growing in the last years to the point where it is no longer considered as a single field of knowledge but rather a set of paradigms, frameworks and tools that can be applied to many other fields where Information Technologies (IT) are also present (e.g., medicine, economy, industry, defense) [62, 65]. Moreover, as it evolved, Big Data was integrated with many different well-known disciplines, from already established ones such as High Performance Computing (HPC) [11] and Artificial Intelligence [40] to more novel areas like fog/edge computing [98], lending its virtues to the community such as massive scalability or data processing capabilities. However, Big Data did not make its way across those fields untouched, as it has also significantly evolved in terms of the core paradigms its technologies are reliant on, as well as the underlying infrastructures used to deploy them.

Considering MapReduce [36] as the first Big Data processing paradigm, successfully implemented by open-source projects such as Apache Hadoop [101], the current ones are the far more versatile Apache Spark [114] and Apache Flink [25] frameworks. This evolution marked an important change when it comes to the resources typically demanded by those frameworks. While the MapReduce paradigm mainly relied on disk and network, as data had to be computed and moved in batches across the cluster, the newer data processing engines add CPU and memory along disk and network as intensively used resources, considering that now data can also be computed following a more efficient in-memory strategy, especially for iterative

workloads [95]. This in turn causes energy consumption to also be a major concern now, as it already is in other CPU-intensive areas like HPC. Nonetheless, with these newer data processing engines there are also far more different types of applications and use cases, which greatly increases the heterogeneity of the used resources. For instance, a Big Data cluster could be executing a CPU-intensive iterative machine learning algorithm, and next a network-intensive stream processing workload, or even both at the same time. When it comes to the underlying infrastructure, Big Data applications have left behind the commodity clusters that were first used and have evolved towards more flexible approaches. As of today, most users deploy their applications using well-established platforms like the Cloud and mature technologies such as virtualization, both of which offer on-demand, quicker deployments as well as pay-per-use billing. Furthermore, it is possible to benefit from any advance made in those technologies (e.g., serverless, edge computing), thus granting an extra degree of flexibility and, in some cases, of efficiency.

Unfortunately, if something characterizes Big Data applications is their potentially large infrastructure size, being able to directly scale with the data size being processed and/or the time constraints desired. If such deployments are made on a Cloud computing platform, a large number of virtual instances could be needed. As such infrastructure size increases, it may be increasingly difficult to passively manage its monitoring, to the point of sacrificing some fine-grain information at the expense of the larger picture, or to actively control the amount of resources used, which may cause higher costs than needed due to underutilized resources.

This Thesis explores and brings forward several related ideas that jointly seek to provide an improved management of Big Data applications when it comes to their resource usage and code profiling, all of this supporting deployments based on container-based virtual environments due to their great popularity [26]. The core ideas are based on underlying premises backed on well-known and proven-to-work technologies and paradigms, but that either have not been extensively implemented for Big Data or in container-based environments. The main contributions of this Thesis have been proposed through the development of two publicly available frameworks which have been empirically proved with several representative experiments and scenarios. What is most important, both frameworks have been designed around some core concepts: 1) being as simple as possible in terms of their user interfaces;

2) being able to work in real time, without having to restart any application or tool; and 3) being as non-intrusive as possible, aiming for low overhead and working transparently along with Big Data processing frameworks and workloads.

The first proposed framework, BDWatchdog, focuses on offering an exhaustive application analysis via both resource monitoring and code profiling. When it comes to monitoring, time series are used and centered around individual processes, rather than the whole system. This fine-grain monitoring in combination with the filtering and aggregation operations supported by time series databases offers many options to analyze a particular workload, from the resource variations between different underlying hosts (e.g., master and slaves), to the accounting of same-purpose processes spawned across the cluster (e.g., Spark executors), or even isolating specific components of a workload (e.g., brokers and processors in a streaming application). Regarding code profiling, BDWatchdog implements low-level stack sampling directly from the operating system kernel, without interrupting, instrumenting or disturbing the application in any way. Furthermore, this kind of profiling is performed unbeknownst to the applications, and thus it can be started and stopped at any time, while also having all the profiling data available on demand and in real time. The combination of both functionalities makes BDWatchdog the building block for other higher-level tools that require real-time features and precise resource usage of workloads without taking into account the underlying infrastructure where they are deployed.

The second proposal is a framework for providing automatic, real-time resource scaling features for Big Data applications. Concretely, this framework explores how it is possible to devise different policies to tune the resources given to the applications running on a cluster of containers. Our approach implements a scaling policy according to the real resource usage in terms of CPU and memory, thus creating a serverless environment. Current serverless platforms are mainly designed to run simple scripts that are agnostic of their underlying infrastructure, while at the same time have the potential of scaling their resource demands according to their needs. So, the user does not need to specify any initial resource configuration in a serverless scenario, thus granting a certain degree of auto-configuration and self-management features. Additionally, such resources are billed taking into account only the actually used amounts, not the allocated ones as in traditional Cloud environments like In-

frastructure as a Service (IaaS). Unfortunately, existing serverless platforms enforce strict rules for the applications that are supported, which often leaves out any task that requires a more complex environment such as a virtual instance (i.e., an operating system, third-party libraries and configuration files). To preserve the flexibility and benefits of the serverless paradigm, the proposed framework brings forward a platform that deploys containers which behave as traditional instances, but that at the same time follow a resource allocation policy according to the serverless philosophy. To do so, our framework is built upon BDWatchdog to take advantage of its fine-grain monitoring capabilities to implement an active management of the resources given to the set of containers that jointly execute a particular Big Data workload.

Finally, this Thesis presents a practical scenario where both frameworks work together to manage energy as another system resource alongside CPU and memory. To provide this novel energy management, a new tuning policy is created for the rescaling framework in order to implement the concept of power budgets or energy caps. The idea of a power budget can be defined as an energy limit imposed upon an entity so that it is enforced along time. More specifically, in this scenario different users and their applications have been modeled with varying power budgets being applied, considering the applications as groups of containers. In order to implement the energy limit enforcement onto the containers, CPU throttling combined with configurable policies have been defined. It is also interesting to note that this energy management has been implemented fully with software support, from the energy measurement to the energy capping.

Chapter 2

BDWatchdog: fine-grain resource monitoring and profiling

In order to build more advanced tools and frameworks that implement active or knowledge-based resource management strategies, first there is a need to have system resource usage information (i.e., CPU, memory, disk and network) as accurate and as readily available as possible. In our scenario, any piece of information (e.g., CPU usage) that is gathered from the execution of the applications can be later used as part of any policy decision when it comes to find more efficient ways of running them with the available resources, or simply understand better what they are doing and how they are behaving.

This chapter presents BDWatchdog [42] as one of such tools that fills the role of in-depth analysis of Big Data applications but with some specific goals in mind. Unlike most of the existing solutions designed for monitoring and/or profiling purposes, BDWatchdog aims at offering finer granularity. On the one hand, the metrics obtained when monitoring the resource usage are retrieved at both the system and process level. This feature is crucial to support forms of virtualization like containers (e.g., Docker [81]) or emerging paradigms like serverless computing [15], as both do away with the idea of a physical or virtual host and rather work with groups of processes. On the other hand, profiling is tailored for using newer forms both on the technical aspect, by replacing instrumentation with low-level kernel profiling, and on the visual aspect, by using flame graphs. This kind of profiling enables to have

a snapshot of a running application at any given time window, without disturbing it or having to wait until it ends.

The rest of this chapter is organized as follows: Section 2.1 presents the concepts, terminology and describes the overall use case scenario. Section 2.2 describes the current technologies and their limitations, as well as the proposed solutions to overcome them. In Section 2.3, the architectures used to implement the two basic functionalities, monitoring and profiling, while remaining flexible and scalable at the same time, are explained. The overhead of this framework on Big Data workloads is presented in Section 2.4. Real-world use case scenarios where BDWatchdog is most interestingly applied are described in Section 2.5. Finally, Section 2.6 summarizes the results and reviews the main use cases.

2.1. Background

To highlight the importance of why a more low-level performance analysis of processes is desirable, instead of just focusing on traditional system-wide monitoring and code targeted profiling, we first need to lay off the basic concepts of the involved technologies on which our framework relies on.

2.1.1. Big Data frameworks as groups of processes

Currently, Apache Hadoop [101] is the leading open-source solution and software architecture to host most of the Big Data applications and frameworks. Hadoop provides a software ecosystem composed of the combination of YARN (Yet Another Resource Negotiator) [106] and HDFS (Hadoop Distributed File System) [96], which take care of the resource management and the data handling, respectively. Although complex, Apache Hadoop was created with modularity in mind and, because of this, its isolated main components (i.e., YARN and HDFS) are in the end comprised of processes running on a different Java Virtual Machine (JVM). For instance, YARN is created with one ResourceManager process in the master node and a NodeManager process on each slave node, while HDFS makes use of one NameNode process and a DataNode process in the master and slaves nodes, respectively.

Similarly, data processing engines such as MapReduce [36] or Apache Spark [114] are also implemented by using isolated JVM processes (e.g., Hadoop MapReduce launches Map and Reduce tasks running as YarnChild processes). Moreover, this design also applies to other Big Data frameworks and applications outside the Hadoop ecosystem such as databases like Apache Cassandra or brokers like Apache Kafka [50].

In the end, most of the common Big Data-related software relies on multiple independent and coordinated JVM processes running on different hosts to provide redundancy and fault tolerance. BDWatchdog will take advantage of this trend aiming to trace and account the resource usages of the individual components by using either their process identifiers, their names or a combination of both.

2.1.2. Software containers and their monitorization

Considering 'instances' as the underlying minimal and independent infrastructure component used to host Big Data frameworks and applications, their possible implementations have evolved in the last years with newer paradigms such as the Cloud. Originally, frameworks like Apache Hadoop were born with the idea of commodity hardware in mind, building clusters comprised of off-the-shelf hardware and running everything on a bare-metal fashion, placing on top all the necessary redundancy using software solutions. Nevertheless, since the Cloud computing boom, it is much more common to run Big Data applications on virtualized resources, using Infrastructure-as-a-Service (IaaS) or Platform-as-a-Service (PaaS) providers [6, 14] to create private, on-demand and scalable clusters.

However, these Cloud-based services generally rely on 'heavy' hypervisor-based full virtualization, which besides imposing a performance penalty also gives the software the illusion of a dedicated machine with allocated resources. Although this latest characteristic may seem like a positive thing, recent 'light' virtualization technologies based on software containers (e.g., Docker) or Cloud-based paradigms like serverless computing [15] (e.g., AWS Lambda [8]), prove that it is beneficial to move beyond and further abstract the execution of services or code from a specific hardware or resources.

In this context, BDWatchdog aims to focus the resource monitoring on the processes rather than on the instances and thus to make it portable to more scenarios, from traditional dedicated hosts to virtual machines, and ultimately to software containers.

2.1.3. Low-level profiling

Profiling has traditionally been carried out using attachable programs that instrument or measure isolated applications or even just small fragments of source code, usually in an on-demand way and in a controlled environment. However, if we want to continuously profile (i.e., run the profiler indefinitely) a large number of processes and applications across a large number of instances, there is a need for a more simple and unmanaged way of retrieving such profiling data.

Fortunately, with the increasing Linux kernel developments and improved hardware support, new tools are available that work at a much lower abstraction level next to the kernel and offer more and richer information about the execution of any application. Tools such as *DTrace*, *eBPF* or *perf* [111] are able to offer tracing, software and hardware events accounting and lightweight profiling not only regarding CPU resources but also memory, disk and network. This type of low-level profiling has successfully been used on large-scale systems to obtain interesting metrics that allow to improve or better understand infrastructure utilization for a broad spectrum of workloads [67, 90]. Currently, BDWatchdog only focuses on *perf* and its powerful profiling capabilities.

2.2. Related work

There are currently many solutions to provide resource usage metrics and even more to provide application profiling. However, most of the existing monitoring tools focus on fully-fledged instances (i.e., where a dedicated operating system is exposed) and their resource utilization as a whole (e.g., percentage of CPU consumed for the whole system), instead of providing per-process metrics (e.g., percentage of CPU consumed by a specific process). As stated in Section 2.1, the process-

based approach is interesting to be applied in Big Data scenarios as the need to assess workloads more precisely is key. This application-oriented monitoring has already been proven useful in scenarios where data are generated and processed in a streaming manner on a Big Data cluster [70]. So, our framework aims to extend the system monitoring and all of its metrics to processes.

It is worth mentioning that for Big Data scenarios there are several solutions such as Starfish [60], ALOJA [89] or MR-Advisor [110] that specifically target Hadoop MapReduce applications and look to improve their efficiency. These solutions are interesting in that they effectively create a feedback-based system where MapReduce jobs are executed and then optimized after their performance has been analyzed using metrics and logs. Nevertheless, these tools require the jobs to finish and are restricted to MapReduce and the Hadoop ecosystem overall.

Regarding JVM profiling tools, available solutions either impose heavy penalties due to the use of source code instrumentation, are not scalable or are not designed to work in real time. In addition, although the line plots used to view time series are not new, BDWatchdog adopts flame graphs as a novel way of visualizing profiling data that replaces tree call graphs.

Finally, to the best of our knowledge, there is no other tool that allows to combine resource monitoring of operating system processes and mixed JVM and system profiling to analyze an application in any given time window, or even in real time, in addition to the experiment and workload time-stamping accounting to aid later analysis.

2.2.1. Infrastructure monitoring

One of the first monitoring solutions that is still widely used is the Simple Network Management Protocol (SNMP) [27], which appeared in the late 80's to allow for an easy way to monitor a large infrastructure, being originally targeted to networking equipment or network-attached devices. This protocol presented a decentralized methodology by which the system information was retrieved and sent periodically from agents to masters to be later processed and visualized. Currently, many monitoring solutions such as Cacti [74] or Zabbix [100] support the use of

SNMP underneath to retrieve basic information. However, SNMP was designed for network-related devices and, even with the addition of operating system extensions and plugins, the resource information that is able to retrieve remains limited to the system as a whole. In addition to the aforementioned technologies, other common solutions such as Ganglia [79] or Nagios [64] developed their own improved way of retrieving system metrics, but the focus is still put on system-wide analysis.

All of these solutions are based on the same architecture, following the SNMP philosophy, which consists of multiple agents that generate metrics, a means of sending such data over the network, a master service which aggregates the information, and finally a way of storing and possibly plotting it, most commonly using Round-Robin Databases (RRD). While this architecture is perfectly valid for scenarios where the infrastructure does not change dynamically, such as data centers or supercomputers, it may not be entirely appropriate in environments where instances are created and destroyed on-demand or where the size of the overall infrastructure may vary greatly over time, such as the case of Big Data on the Cloud. In addition, most of these solutions build architectures made of components that were specifically created (e.g., Ganglia uses its own monitoring daemon as its agent), usually to meet high efficiency requirements or provide specific features, and thus they may not be entirely interoperable.

BDWatchdog aims to use a more flexible architecture that, although it takes from the same idea of agents and masters, is scalable and uses interoperable or interchangeable components. Instead of coupling the agent and master implementations or limit the data persistence and visualization with the use of RRD, we define the agents as specialized data source programs that will collect different metrics, a time series database that will store these metrics indefinitely as well as aggregate them on demand, and a visualization tool on top of everything. All these components are replaceable by other solutions as long as the interfaces for information exchange, mainly REST APIs, are used and respected. Several agents can independently feed different metrics to a time series database, and any visualization tool that integrates with such databases can be used to plot the time series.

Finally, it is worth mentioning Amazon CloudWatch [3], as some of its monitoring and visualization features have a similar approach to our framework and the architecture it uses was created with the Cloud and scalability in mind. CloudWatch

was originally developed to provide Amazon Elastic Compute Cloud (EC2) users with a tool to monitor the health of their EC2 instances, although it has now been expanded to support other services. However, this solution only works inside the Amazon Web Services (AWS) Cloud ecosystem [9] and the implementation is not publicly available. Regarding its usage, although a basic functionality is given for free, more detailed monitoring with higher sampling frequency and advanced metrics is billed. Moreover, like previously described solutions, CloudWatch only provides system-wide information, considering each AWS instance or service as an individual system.

2.2.2. Instance monitoring

For single instance monitoring, there are several tools that work on UNIX-derived systems, giving basic information and providing accounting of system resources. The simplest yet useful tools to get instant information about the system are *top* and some of its counterparts like *iotop* and *iftop* for disk and network devices, respectively. These tools combined can provide a very accurate picture in real time of what resources are being used to what extent and, as a whole, the health status of the instance. However, these tools are targeted to being used interactively and for short time intervals, typically when a diagnosis of the system is required.

To collect system metrics for long time intervals, other tools exist such as *collectd* or *sar*. These programs usually work as background daemons or unattended processes that write periodically the system's state (including resource metrics) to log files, which can be later reviewed when necessary. Unfortunately, they are designed to locally monitor single instances and, in order to aggregate or export their information, plugins are needed such as *collectd-nagios*. In addition, these solutions are system-oriented, and plugins or modifications are necessary to account for per-process metrics.

In BDWatchdog, *atop* [13] is used as the underlying instance monitoring tool combined with a Python-based processing pipe to act as an agent that generates a stream feed of metrics. This tool has been selected as it is capable of providing CPU, memory and disk metrics on a per-process basis. To support per-process network metrics, as of the current state of the art and leaving aside raw access to

the underneath system accounting (i.e., the /proc directory), there are only few solutions that are able to provide network accounting individually per process. Two analyzed solutions were *nethogs* [85], already considered in other works for very similar purposes [70], and the *netatop* module [84], which must have access to the kernel and be loaded into it. This latter requirement is important when using container-based environments, as *netatop* may be loaded but is unable to provide any useful information. In order to provide higher flexibility and to overcome this limitation, BDWatchdog includes support for both tools. While *netatop* is used internally by *atop* and no additional support is required, *nethogs* acts as another individual agent to support per-process network metrics.

2.2.3. JVM profiling and visualization

There are several profiling tools specifically supporting the JVM that have evolved and are used to profile programs in a more on-demand fashion, generally after detecting poor performance or memory issues. The most common use case scenario involves attaching some type of profiler as a Java agent and gather information for short periods of time to later generate a report. Examples of successful profilers are VisualVM [108] or JProfiler [31], but there are many more commercial and open-source solutions. These tools provide very accurate measurements from CPU and memory usages to threads and software locks. Nevertheless, all of them are inherently limited to the JVM environment where the bytecode is executed on, leaving out the system calls and native libraries. On the other hand, system profilers like the Linux *perf* tool [111] are not able to understand anything that runs inside the JVM, as the intermediate layer created by the interpreter and the bytecode language is present. Furthermore, JVM profiling has traditionally been affected by a noticeable overhead, lack of real-time support and the use of tree call graphs as the means of visualization. All of this makes traditional profiling cumbersome to be used on a large-scale scenario like Big Data clusters, where there are many instances that may be deployed and destroyed on the go.

To overcome these issues, flame graphs [53] were originally created as a means of visualizing mixed system and application profiling over lengthy time intervals, which would quickly expose hot spots on an application's code. With later improvements,

Java was also supported and flame graphs are now able to combine system and JVM stacks. Moreover, because a system profiler, *perf*, is used to collect the stack samples to draw the flame graphs, no code instrumentation is performed or required and overall the performance impact remains very low. These new visualization tools effectively replace tree call graphs and give a more efficient way of describing how the code is executed and particularly which functions or call paths are more frequent, which would represent areas of the source code where longer time is spent during execution.

Unfortunately, the available tools and scripts from the flame graph project are designed to be used in an on-demand and batch manner, that is, performing the profiling and then creating the graphs when needed in a certain host. In BDWatchdog, the original tools and scripts are still used, but integrated and expanded with additional support. With our solution, profiling is performed on the background in an unattended and continuous manner in all the hosts across a cluster. Finally, the generated profiling data are sent as a stream to an external database which allows later analysis at any time, both per-host and aggregated.

2.2.4. Mixed monitoring and profiling

While there are viable solutions for monitoring both large infrastructures and single instances, as well as for profiling, to the best of our knowledge there is no tool available that combines both and at the same time targets individual applications running across a cluster. Furthermore, our framework is also ready to support software containers, as a form of lightweight virtualization increasingly popular but also significantly different from classical hypervisor-based virtualization, and works on scalable Big Data clusters.

2.3. Architecture design and implementation

The BDWatchdog framework focuses on two major functionalities: monitoring and profiling, and thus one specific architecture for each has been developed. Although both architectures share some basic design guidelines in order to achieve

scalability, the underlying technologies used are different, being independently explained in Sections 2.3.1 and 2.3.2 for monitoring and profiling, respectively.

However, to better assist the subsequent analysis of the collected results, two additional minor tools are provided. One tool implements time-stamping control of experiments and individual workloads, considering an experiment to be a series of sequential workload executions. With this feature, it is possible to register the start and end times of experiments and workloads to later properly isolate their time windows for specific analysis in a fully automatic manner. The other tool is a fully client-side web user interface where both time series and flame graphs can be plotted side by side. This web interface, described in Section 2.3.3, has been integrated with the monitoring and profiling architectures as well as with the time-stamping control. This allows to easily visualize the retrieved data of experiments or workloads.

2.3.1. Monitoring tool

To create a fully scalable monitoring system that can be used across many instances as well as allowing to dynamically resize with new instances being added or removed, the architecture has been divided into isolated components with specific functions distributed along three layers, as depicted in Figure 2.1. The bottom layer is responsible for data generation and is composed of: 1) the system processes to be monitored, running in a bare host, virtual machine or container; and 2) the agent programs that generate and send out the monitoring data to the next layer. This middle layer implements the data persistence and is mainly composed of: 1) a time series database manager; 2) an underlying scalable and distributed database; and 3) a load balancer, used to provide more flexibility and scalability. Finally, the uppermost layer allows for the high-level data analysis and thus it only reads data from the persistence layer. On this analysis layer we can find several applications that can use time series to extract information such as line plot visualization tools, alert scripts or report generators. As a whole, these layers can be scaled horizontally thanks to the use of technologies and paradigms such as load balancers, distributed databases or stateless agents that stream their data rather than storing them locally. This scalability enables users to monitor large clusters in real time while introducing low overhead, as long as all the layers are balanced in size and rescaled if needed.

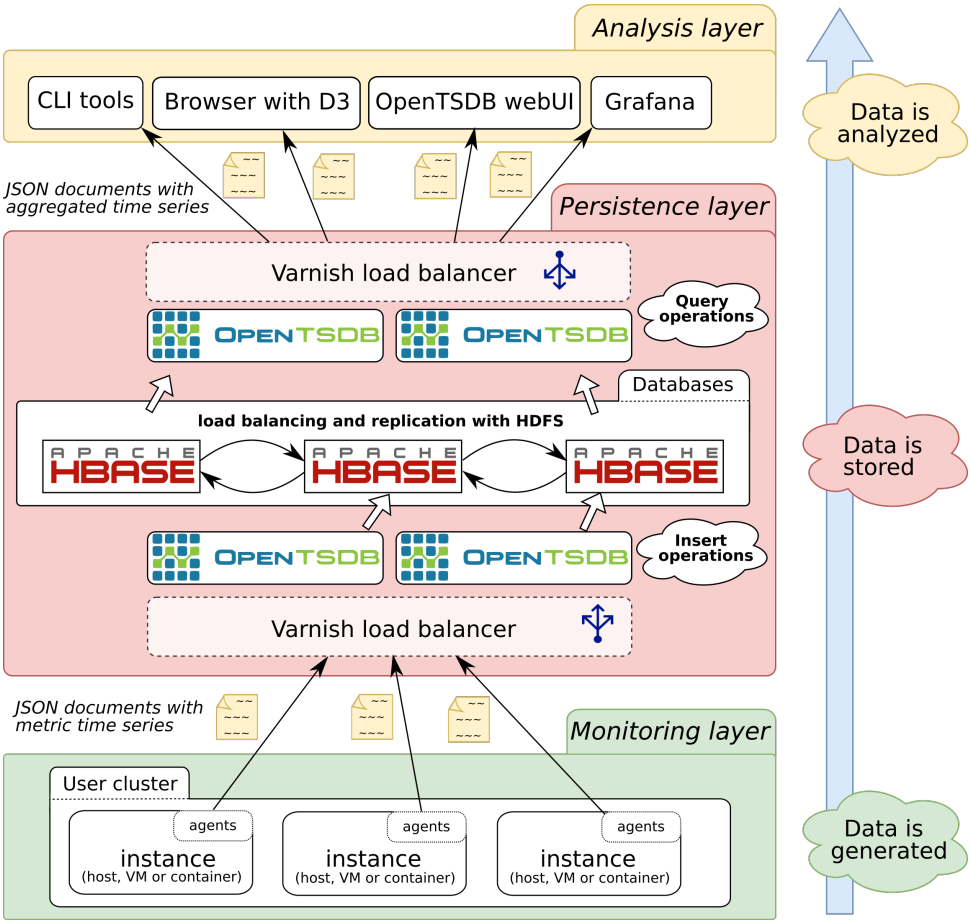


Figure 2.1: BDWatchdog monitoring architecture

In order to understand how the monitoring data are managed across the architecture, it is important to explain first the data structure and then, according to the different architecture layers, how these data are generated, persisted and analyzed. These four aspects are respectively described in more detail next.

Data structure and metrics

The monitoring of system resources has been implemented by recording defined metrics over time, thus creating time series. In our scenario, a metric is defined as the smallest 'atomic' chunk of data that is obtained from a certain system resource (e.g., CPU). Each metric is created by a string with a name, a numerical value, which represents the usage of the resource, and a UNIX timestamp. To properly

```
1 "metric": "proc.mem.virtual",
2 "timestamp": 1491472173,
3 "value": 133.15,
4 "tags": {
5   "host": "host3",
6   "pid": 907,
7   "command": "(python)"
8 }
```

Listing 2.1: Definition of a monitoring metric as a JSON document

Table 2.1: Example of the per-process disk write metric (in MiB/s)

Host name		*host1*						*host2*					
Process name		*P1*	*P2*	*P3*				*P1*	*P2*	*P3*			
Process Identifier		*78*	*56*	*100*	*101*	*102*	*103*	*56*	*33*	*102*	*103*	*104*	*105*
Time (s)	*10*	0	5	1.5	1	2	1	0	10	1	2	2	3
	20	0	6.5	2	2	2	2	0	12	2	1	1.5	1
	30	0	7	2.5	2.5	2.5	3	0	11	3	2	2	2

create a metric all these fields are mandatory (see Listing 2.1). Optionally, a set of tags can be attached to each metric in order to parametrize and differentiate it (e.g., host name to link the metric to a certain host or process name to get usages of a specific program). This way of structuring data is very similar to fact tables used in Online Analytical Processing (OLAP) cubes [30], where the metric would be the fact to be measured and the tags would be the dimensions, with timestamps being a particular dimension (see Table 2.1 for an example of several metric values in a cube form). Both system-wide and per-process metrics supported by BDWatchdog, as well as their tags and associations, are respectively shown in Tables 2.2 and 2.3.

However, instead of using relational SQL databases to store the monitoring data like the ones used for OLAP cubes, BDWatchdog makes use of a NoSQL and distributed database in order to grant scalability and flexibility. By viewing the data as cubes and using the tags as metadata, it is possible to apply different filters and aggregations allowing richer and higher-level queries such as getting the average

Table 2.2: System-wide metrics with their associated tags

System monitoring					
Resource	cpu	mem	disk	net	power/temp
Metrics	sys.cpu.user sys.cpu.kernel sys.cpu.idle sys.cpu.wait sys.cpu.usage	sys.mem.free sys.mem.usage sys.swap.free	sys.disk.read.mb sys.disk.read.ios sys.disk.write.mb sys.disk.write.ios sys.disk.usage	sys.net.in.mb sys.net.out.mb sys.net.usage	sys.power sys.cpu.energy* sys.cpu.temp*
Tags	host name core id	host name	host name disk	host name interface	host name core id (only *)

Table 2.3: Per-process metrics with their associated tags

Process monitoring				
Resource	cpu	mem	disk	net
Metrics	proc.cpu.user proc.cpu.kernel	proc.mem.resident proc.mem.virtual proc.mem.swap	proc.disk.reads.mb proc.disk.writes.mb	proc.net.tcp.in.mb proc.net.tcp.in.packets proc.net.tcp.out.mb proc.net.tcp.out.packets
Tags	host name process name PID			

CPU consumed by a group of processes across a set of instances on a particular time window, or just adding all the disk bandwidth consumed on an instance by its processes. Taking Table 2.1 as an example, some interesting queries would be the average aggregate disk write bandwidth used by all *P2* processes across *host1* and *host2* (8.58 MiB/s for the whole period of time), or the average disk write bandwidth of *P3* processes on *host1* (2 MiB/s).

Time series monitoring layer

As mentioned in Section 2.2.2, to collect the system metrics on a single instance, different agent programs are used. These programs, such as *atop* or *nethogs*, are able to create a stream of data more or less configurable but always time window-based.

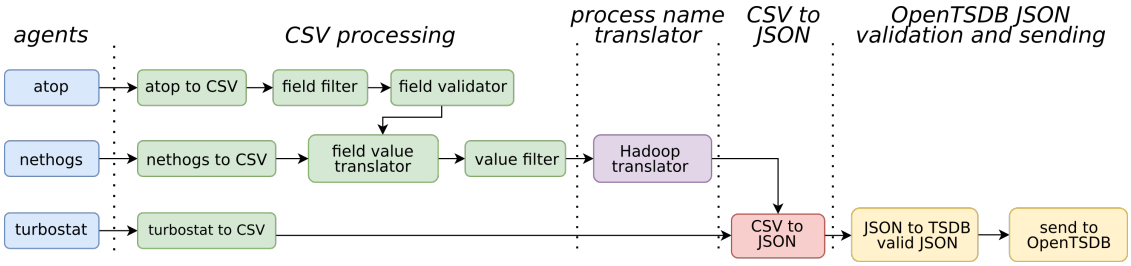


Figure 2.2: Processing pipeline of the monitoring tool

Nevertheless, these raw data generated by the agents have to be processed to fit the metric structure as defined previously. So, a lightweight Python-based processing pipeline has been coupled to the agents (see Figure 2.2). An important stage of this pipeline that is worth mentioning is the 'Hadoop translator'. In this stage, the command names referring to JVM processes, which are reported by tools like *top* as generic Java processes, are properly renamed using their Process IDentifier (PID) to the actual Java main class that is running in that JVM (e.g., a NodeManager in the case of YARN).

This same pipeline structure is used for all agents with the end metrics ideally being abstracted from the underlying agent, as shown in Figure 2.2. With this abstraction in mind, different agents can be used as needed to provide different sets of data. More specifically, *atop* and *nethogs* are used to provide with all the system and process metrics for CPU, memory, disk and network supported by BDWatchdog (see Tables 2.2 and 2.3). Other agents like *turbostat* have also been used for other system metrics such as CPU power consumption and temperature, which proves the flexibility of this approach.

The generated monitoring data are in the end sent to a REST API using JavaScript Object Notation (JSON) documents, an open-standard data serialization format widely used due to its lightweight and flexibility features. This API acts as the interface between the monitoring and persistence layers. The flexibility from using this web protocol allows us to focus on sending the data to a single point of reception without having to worry about how data are later processed. This approach also allows to scale the number of agents that are feeding data transparently as long as data are properly ingested by the persistence layer (i.e., it may need to be rescaled appropriately). In addition, because HTTP protocols are used with REST APIs to

send the data, lossless stream compression can be used as long as the persistence layer supports it. This feature is useful to reduce the size of the stream that is sent through the network. Nevertheless, it has to be noted that once the JSON documents reach the persistence layer, they are interpreted by OpenTSDB and stored properly as time series points.

Time series persistence layer

The data persistence layer provides the components that take care of ingesting, storing and granting access to all the data. As can be seen in Figure 2.1, this layer mainly consists of: 1) a non-relational and scalable Hadoop database (Apache HBase [102]); 2) OpenTSDB [97, 112] as the time series database manager; and 3) Varnish as a load balancer. The combination of two databases, HBase and OpenTSDB, has been used to properly manage the sparse nature of time series points, due to the high number of values and tags generated over time, while queries may retrieve only a small set of them. On the one hand, HBase has proven to be highly efficient with such sparse data. HBase is also a distributed, fault-tolerant and easily scalable database. By relying on HDFS as the storage engine, it can easily grow by adding more nodes and storage resources. On the other hand, OpenTSDB only acts as an interface to handle metrics and time series properly. So, it does not actually persist any data and uses the underlying HBase database to store time series points. Taking benefit of this approach, OpenTSDB allows to span multiple time series daemons to perform read or write operations in parallel. Regarding the storage requirements of BDWatchdog, as a guideline it can be estimated taking into account that each metric point is about the size of a few bytes (e.g., 1 million points add up to about 30 MiB for a metric point size of 35 bytes). The storage needed is thus directly proportional to the amount of data generated. Additionally, LZO compression is available to be used in order to reduce the size of the data.

Finally, Varnish is used as a load balancer. Varnish is specifically described as an HTTP accelerator, used to relieve the pressure on very active websites by caching content and distributing HTTP connections. However, it can also be used as a load balancer thanks to its support of several scheduling algorithms, backend weighting and backend health checking. In BDWatchdog, Varnish only serves the main purpose of load balancing, as content is not cacheable. In addition, Varnish

allows to expose a unique point of entry through which data can be sent or retrieved, as described before when explaining the generation of time series.

Time series analysis layer

The uppermost layer allows to extract information from the stored time series data by using the REST API provided by OpenTSDB. With this feature it is possible to perform queries and retrieve time series in between a specific time window with the possibility of using high-level operators for selection and aggregation. Besides the direct visualization of the monitoring data using line plots, other use cases such as usage reports and alerts may also be implemented on top of the retrieved data to extract more, richer information from them.

There are several solutions to visualize time series (e.g., Grafana) that are able to integrate with different time series databases such as OpenTSDB by using their REST APIs. For more specific use cases or for integration with other programs and scripts, third-party plotting libraries can also be used programmatically like Data-Driven Documents (D3) [20] for JavaScript or matplotlib for Python. A straightforward option is to use the basic web user interface provided by OpenTSDB, which is directly available after the database is deployed without further configuration. Nevertheless, BDWatchdog also provides its specific web user interface that uses D3 to plot time series, as described in Section 2.3.3. In addition, several scripts are provided to plot time series for a specific time interval as image files from a Command Line Interface (CLI).

2.3.2. Profiling tool

In a similar way to the monitoring tool, profiling can also be divided into three separated layers with identical roles, as depicted in Figure 2.3. The bottom layer, where all the profiling data are generated, consists of the same previous instances and processes but now to be profiled, and a profiler agent coupled with a data processing pipeline. The middle layer, where the profiling data are stored, is composed of a scalable document-based database and a REST API microframework that acts as the entry to feed data. Finally, the upper layer allows the analysis of the data by

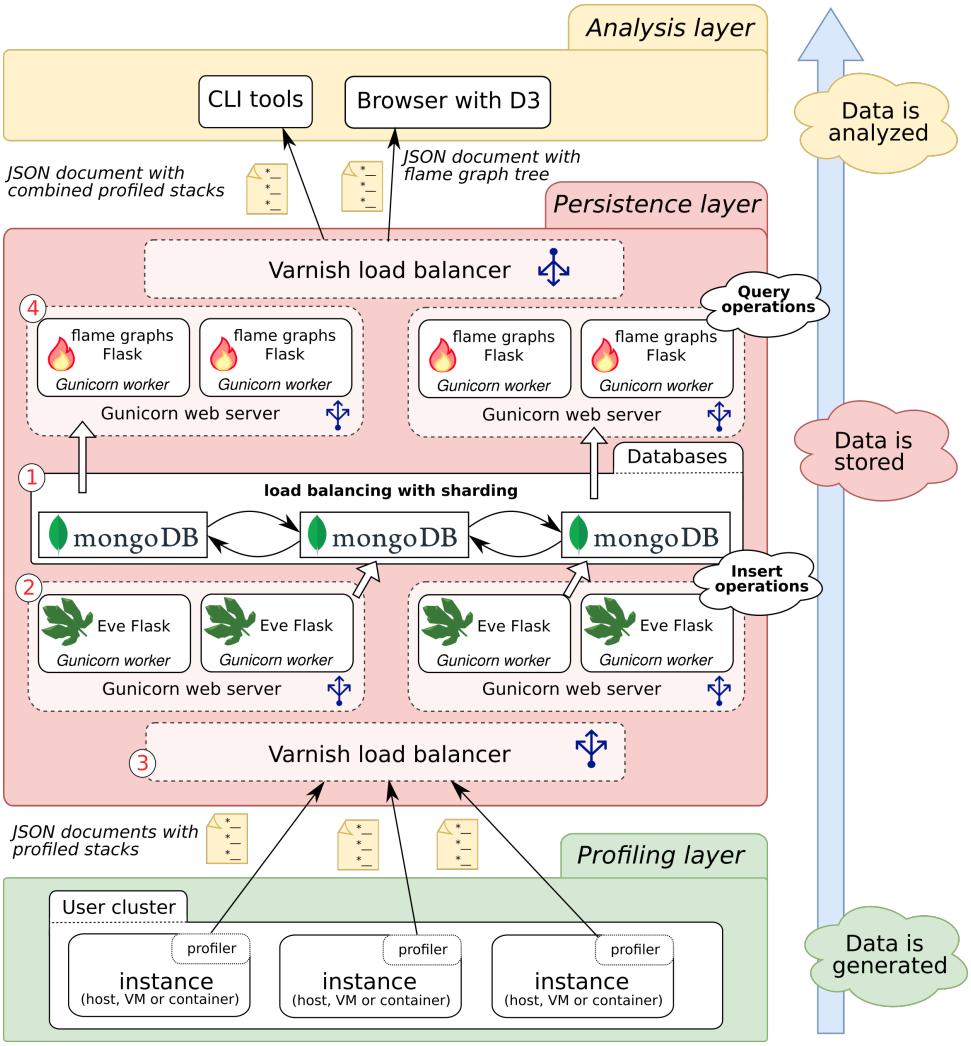


Figure 2.3: BDWatchdog profiling architecture

visualizing them as flame graphs using CLI tools or the D3 JavaScript library. The profiling and persistence layers have also been designed to be scalable horizontally and thus to allow dynamic profiling for a time-varying number of instances. It is also worth specifying that this architecture allows for real-time profiling, as the profiling data can be sent and afterwards retrieved and analyzed with low delays. However, profiling configuration as a whole must be more limited and sensibly specified due to the varying size of the profiling data and their processing requirements to be visualized, as compared to the static size and more or less simple aggregation used for time series.

A profiled stack is in this case used as the underlying data structure to create and store profiling information in BDWatchdog. This data structure along with all the architecture layers are described next.

Data structure and profiled stacks

A profiled stack is defined as the smallest chunk of data that is obtained from a certain application. More specifically, it represents the series of functions or subroutines calls that a program has made both inside its execution environment and outside of it (i.e., through a system call) in a precise sampled instant. This broader view is possible because of the higher abstraction level of the profiler used in BDWatchdog, *perf* [111], which works alongside the kernel and in a closer level to the hardware. In addition to the call stack, a timestamp and a host name are added to identify the environment of the measure. To reduce the amount of output, *perf* aggregates the stacks and provides a count value of the number of times the profiler has found such stack. For an example of a JSON document containing a definition of a profiled stack, see Listing 2.2.

```
1 "timestamp": 1495813324,
2 "hostname": "hadoop0",
3 "stack": "java-18714;start_thread;java_start;
4     JavaThread::run;JavaThread::thread_main_inner;
5     attach_listener_thread_entry;JvmtiExport::load_agent_library;
6     Agent_OnAttach;jvmti_GenerateEvents;
7     JvmtiCodeBlobEvents::generate_compiled_method_load_events;
8     JvmtiExport::post_compiled_method_load;cbCompiledMethodLoad;
9     generate_single_entry;sig_string;jvmti_GetMethodName;
10    Method::checked_resolve_jmethod_id",
11 "value": 6
```

Listing 2.2: Definition of a profiled stack as a JSON document

Low-level process profiling with JVM support

As mentioned before, *perf* is mainly used to collect profiling data. However, other scripts and tools had to be developed to particularly combine system and JVM calls. Because bytecode is executed on the JVM and is different from the system calls, the memory addresses that are retrieved by *perf* are not directly usable and need to be mapped to the address space used by such JVM. This is now possible thanks to the following two features.

The first one is that, from the profiler point of view, *perf* is aware of this translation problem and allows to use address maps when raw data are processed and missing addresses appear. Besides JVM languages, this also allows to support other interpreted languages (e.g., Python, JavaScript, Perl) which also suffer from this issue, as long as maps are provided. With Java it is possible to get such maps even while the program is running thanks to an attachable agent. The second improvement is that, from Java version 8 onwards, the JVM supports an environment option to leave a certain hardware CPU register, the frame point register, out of the JVM pool of usable registers. This is required because the frame point register is used by *perf* and other profilers to trace the stack of the program.

With the combination of *perf*, a Java agent provided with the flame graph project to generate the translation maps, and additional scripts and tools to process the data, it is possible to generate a stream and send it to an external database (see Figure 2.4). Note that although a system-wide profiling configuration is used for *perf* in the *'perf record'* phase of the figure, the generated data are later tagged accordingly in the *'perf script'* phase so as to properly differentiate the stacks of a particular process from another one by using their PIDs. This generated stream can also be configured at the source to increase the profiling frequency (i.e., number

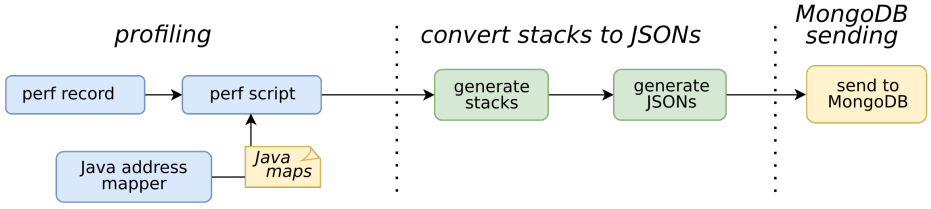


Figure 2.4: Processing pipeline to send profiling data

of stacks retrieved per second) to allow for more accurate measurements at the expense of higher processing times. In addition, although profiling data are not time-dependent like time series, configurable time windows are used. However, instead of using these windows for visualization purposes, they are used to split and isolate the profiling data and make them work in a real-time manner rather than in a batch or on-demand manner. In this way, later analyzes can query profiling data between two arbitrary time points. The lower the time windows are, the higher the resolution that can be used by later queries, although this comes at the expense of more processing and storing requirements as more stacks are generated due to lower aggregation. Finally, as with monitoring, data are sent as a JSON document containing a number of stacks to a REST API exposed as the interface to the persistence layer. However, in contrast to time series, the JSON documents containing the profiled stacks are stored preserving their format, thus requiring a document-based database.

Profiling data persistence layer

Using a similar architecture as the monitoring tool and taking scalability into consideration, MongoDB [80] has been chosen as the database to persist the profiling data (see label 1 in Figure 2.3). MongoDB is a popular document database that can be used to effectively store and retrieve a high number of JSON documents, like the ones used to store profiled stacks (see Listing 2.2 for an example). In addition, it provides sharding capabilities [37], which are used for data distribution and high throughput. In a similar way to OpenTSDB and time series, the storage requirements can also be estimated with the number of documents generated and their average size. However, a profiled stack is considerably larger in size in comparison to a metric point and because of this profiling has to be more sensibly configured to avoid generating too many data.

To implement write and read queries from outside the instance, Eve [45] has been used to successfully expose a REST API (see label 2), through which profiled stacks can be stored and retrieved. Eve is a Python Flask microframework [55] used to create highly customizable RESTful web services, where resource items can be persisted to a MongoDB database. Unfortunately, such Flask-based microframeworks only allow one HTTP operation at a time by default (i.e., threads are not used). To overcome this issue, Gunicorn [56] has been used. This multiple-worker web server

is generally used to effectively deploy applications such as Flask microframeworks with minimal configuration and resource usage. As a web server, it also allows to start multiple threads serving the same application to enable multiple connections simultaneously. By coupling Gunicorn with a load balancer like Varnish (see label 3) and by exposing only the load balancer's address, we also achieve in this case a unique point of entry to write or read data, increasing the flexibility and abstraction between layers. Finally, a simple custom-made Flask microframework was also developed to expose the stacks stored accordingly to the uppermost layer (see label 4), as depending on the client data are expected in a different format.

Flame graphs generation and visualization

The uppermost layer retrieves data to draw the flame graphs. Currently, there are two options to create them only differing in the environment and language used. The first option is to use the available tools and scripts from the original flame graph project [47] to create a Scalable Vector Graphics (SVG) image from the command line. This option also allows to color the flame graph so as to differentiate three types of code: 1) applications's code in green; 2) JVM management code in yellow; and 3) system's code in red. The second option consists of using a web browser with JavaScript enabled, the D3 library and some additional code to draw the flame graph as an HTML-embedded SVG. Both options use a reversed form of the pipeline that generated and sent the data (see Figure 2.5). It is worth noting that both options create interactive graphics that can be better visualized and navigated using a web browser. By clicking each horizontal bar a zoom operation is performed and only the stacks above it are displayed. Like with time series, the web user interface provided with BDWatchdog also supports the generation of flame graphs by using D3, as described next in Section 2.3.3.

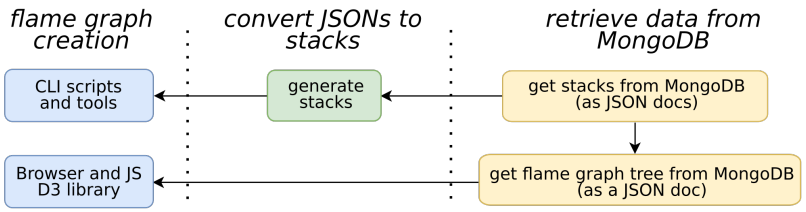


Figure 2.5: Processing pipeline to get profiling data

2.3.3. Web user interface

There exist many visualization solutions to plot time series on a web browser that directly integrate with time series databases. However, to the best of our knowledge, there is no interface available to support flame graphs in the same manner. To solve this issue, BDWatchdog includes a web user interface that allows to plot both time series and flame graphs side to side, which can be of great interest to correlate resource usage and profiling in a particular time window, as will be shown in Section 2.5.

For time series graphs, the following basic options to retrieve and treat them are provided in the interface: 1) selection in order to plot one or more metrics on the same graph; 2) metric filtering by using tags so that specific time series for a metric are retrieved (e.g., time series of the user CPU metric for all the processes on a given host); 3) aggregation with several operations (e.g., sum, average, count) that are applied to the retrieved time series; and 4) downsampling so that time series that present high jittering are plotted smoothly. For flame graphs, a time window is mandatory and a host name is optional. If no host name is provided, an aggregation is performed for all the available profiling information for that time window.

Additional features that can be useful for overall analysis are combined retiming and resizing, which can be applied to all the graphs (i.e., time series line plots and flame graphs) for a better correlation and visualization experience. The retiming feature allows to quickly isolate time windows and perform an analysis of specific workloads or even phases of the workloads, in terms of resource usage, profiling or both. This feature is also integrated with the time-stamping tool to automatically get the start and end times of experiments and individual applications.

2.4. Study of the impact on Big Data workloads

This chapter presents a set of tools that can be used to perform an in-depth analysis of Big Data frameworks and workloads, which inherently incur a performance penalty on their execution. With both monitoring and profiling, the impact can be mainly reduced to that of the CPU and network used, as the programs that

compose BDWatchdog are either data generators or data stream processors and thus their memory and disk requirements remain very low. However, this also means that their computation demands and the stream size directly scale with the amount of data that is handled.

To control the CPU processing overhead, the configuration of the time windows and the profiling frequency can be adjusted in BDWatchdog, also taking into account that this in the end affects the resolution of the data being retrieved. For monitoring, the *atop* agent can be configured to only output active processes (i.e., programs that are running and using a resource). In addition, the monitoring pipeline filters out processes with less than a configurable usage threshold using the 'value filter' stage (see Figure 2.2). With these options, the amount of data generated can be further reduced if necessary and be restricted to actually those applications consuming resources. For the network overhead, data compression can be used on the monitoring pipeline so the impact of network bandwidth is greatly reduced, as mentioned previously when describing the time series generation at the monitoring layer and the corresponding network transmission. However, no compression is currently used for profiling but our design allows it as well. In any case, the end data consist of a stream of short plain text documents which do not usually exceed the size of a few KiB.

To evaluate the end footprint that BDWatchdog imposes on Big Data frameworks and applications, representative workloads have been executed in different scenarios with both monitoring and profiling at the same time, only monitoring, only profiling and none of them. Furthermore, two configurations have been evaluated when running BDWatchdog. The first one is the default configuration that uses time windows of 20 seconds for monitoring and 60 seconds for profiling with a frequency of 101 Hz. This is considered to be adequate for most Big Data scenarios where long tasks are usually executed, as it is accurate enough and still remains non-intrusive. The second configuration is more aggressive and can be used for high accuracy, for example when running very short tasks, although this is not the most common Big Data scenario. For the latter configuration, time windows of 5 and 30 seconds have been used for monitoring and profiling, respectively, with an increased profiling frequency of 202 Hz. Overall, this second configuration represents a 4x increment on the accuracy of BDWatchdog over the first one.

Next, Section 2.4.1 details the experimental configuration, while the results of the experiments are analyzed in Section 2.4.2.

2.4.1. Experimental configuration and software settings

The experiments have been conducted on a Big Data platform [28] deployed at the CESGA supercomputing center [29]. This platform provides a Cloud-based PaaS service to execute on-demand Hadoop virtual clusters [41]. However, unlike traditional PaaS or IaaS services [6, 14] that use virtual machines (e.g., Amazon EC2 relies on hypervisor-based virtualization), the CESGA PaaS deploys software containers (i.e., Dockers) imitating a fully virtualized operating system. As previously stated, BDWatchdog has been developed with the objective of being operative in traditional virtualized environments as well as with emerging software container solutions.

In the CESGA PaaS, a Hadoop 2.8.0 cluster has been deployed using a total of 7 Docker-based containers, with one master and 6 slaves with 4 cores, 16 GiB of memory and 4 dedicated local disks each. All the containers are interconnected using a 10 Gigabit Ethernet network. Apache Spark 2.1.1 has been selected as a representative and popular Big Data framework for data processing on Hadoop storage (i.e., HDFS). Each Spark executor, which runs on slave nodes, has been configured with one core and 4 GiB of memory. Representative Big Data workloads have been executed using the HiBench benchmark suite [61]. Both batch and streaming workloads have been selected (see Table 2.4 for their configuration). In order to run streaming applications, Apache Kafka 2.11 has also been deployed. Kafka acts as a broker, receiving input data from producers outside the system and storing them following a queue-based manner, while at the same time exposing data to consumers (i.e., Spark). A cluster of 3 Kafka nodes has been deployed in the PaaS with the same container specifications as the Hadoop cluster containers.

Regarding software configuration, the containers run Docker 1.8.2 with CentOS 7.2.1511. The Linux kernel version is 3.10.0, while the JVM version is OracleJDK 1.8.0_131.

Table 2.4: Experimental configuration

Batch workloads		
SQL Join	hibench.join.custom.uservisits	500,000,000
	hibench.join.custom.pages	35,000,000
TeraSort	hibench.terasort.custom.datasize	1,250,000,000
KMeans	hibench.kmeans.custom.num_of_clusters	5
	hibench.kmeans.custom.dimensions	12
	hibench.kmeans.custom.num_of_samples	240,000,000
	hibench.kmeans.custom.samples_per_inputfile	4,000,000
	hibench.kmeans.custom.max_iteration	5
	hibench.kmeans.custom.k	10
	hibench.kmeans.custom.convergedist	0.5
PageRank	hibench.pagerank.custom.pages	5,000,000
	hibench.pagerank.custom.num_iterations	2
	hibench.pagerank.custom.block	0
	hibench.pagerank.custom.block_width	16
Streaming workload		
Repartition	hibench.streambench.datagen.intervalSpan	100
	hibench.streambench.datagen.recordsPerInterval	3,300
	hibench.streambench.datagen.recordLength	300
	hibench.streambench.datagen.producerNumber	12

2.4.2. Experimental results

Figure 2.6 shows the execution times for the default (left graph) and the more aggressive (right graph) BDWatchdog configurations, with the different combinations of monitoring and profiling for the selected workloads (see Table 2.4). The results show the mean execution time of 10 measurements. For the streaming workload (i.e., Repartition), the added up processing time of 15 1-minute time windows has been taken into account. As can be observed in the left graph, no remarkable impact is appreciated for any of the workloads when the default configuration is considered, being the highest overhead a 10% for TeraSort when both tools are used. In fact, the monitoring overhead can be considered negligible. As expected, the use of both monitoring and profiling combined consistently present a higher overhead over using only one of them, although in general it is less than the sum of the overheads

of the tools individually. For the aggressive configuration, a noticeable overhead is present in workloads such as SQL Join: a 23% increase in execution time when monitoring and profiling are enabled, being 4.4% and 19% for only monitoring and only profiling, respectively. Although for this configuration more accurate data will be generated, the resulting overhead could be of significance for monitoring as the resources used by BDWatchdog are also accounted for and part of the metrics data. However, it is worth noting that thanks to the use of tags for the per-process monitoring, the BDWatchdog overhead can be left out of the analysis if only the time series for the computing processes and applications are retrieved using tag filtering features. Note also that this filtering is only possible by using per-process metrics, so related monitoring solutions (e.g., Ganglia) cannot filter out their own overhead.

A similar filtering can also be used for profiling, as it is generally centered on the retrieved stacks for the JVMs, leaving out the stacks for the rest of the system processes. Additionally, the profiling overhead and workload duration can be considered of lesser importance as the analysis is centered on the percentage of time spent running code segments, which remains the same regardless of the end execution time.

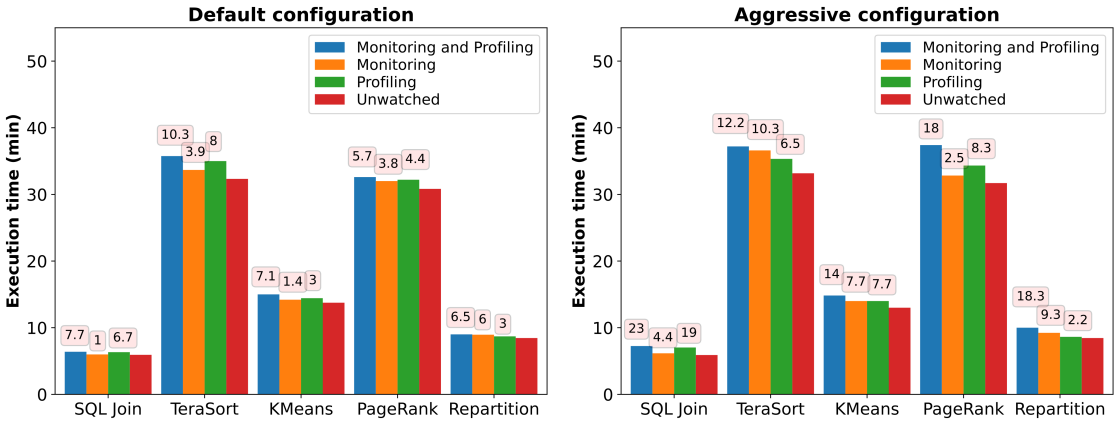


Figure 2.6: Impact study for different configurations and workloads (the overhead is presented as a percentage over each bar)

2.5. Big Data use cases

This section presents use case scenarios with monitoring and profiling individually or combined, where the end user can take advantage of BDWatchdog features compared to previous solutions. On the one hand, using time series can be interesting for the user to easily detect two important things by looking at the plots: resource usage patterns that describe some sort of repeating or out of the ordinary behavior, and resource bottlenecks that may hinder application performance. Examples of both use cases are described in Section 2.5.1. On the other hand, with flame graphs the user can benefit from a summarized and easy to understand view of all the executed code, whether it is application code or JVM management code. This type of analysis may expose code bottlenecks causing performance penalties that may otherwise be hidden when simple monitorization is used or inherently attributed to data processing. Examples of this use case are presented in Section 2.5.2. It is also interesting for the user to combine both visualization tools and get an overall view of any application on a specific time window, both in terms of resource usage and code execution. An example of this case is provided in Section 2.5.3. Finally, all the graphs presented in this section make use of the time-stamping feature (mentioned at the beginning of Section 2.3) to retrieve specific time windows where an experiment with various workloads has been conducted, or to show individual jobs. The web user interface has also been used to get all the monitoring plots, while CLI tools have been used for flame graphs as they provide a better visualization.

2.5.1. Identification of resource patterns and bottlenecks

BDWatchdog allows to filter and aggregate data for a certain system resource and for specific processes. For instance, the user can aggregate the resource usage of all the Spark executors running on the cluster, or the executors of a single container. As an example, Figure 2.7 presents an overall view of the batch experiments shown on a single graph with the CPU usage aggregated for all Spark executors across the cluster. This allows for pattern identification and easy comparison of the different workloads regarding the use of a particular resource. As can be observed, workloads such as SQL Join and TeraSort show a more or less constant resource usage that diminishes at the end phases, while other tasks such as KMeans and PageRank

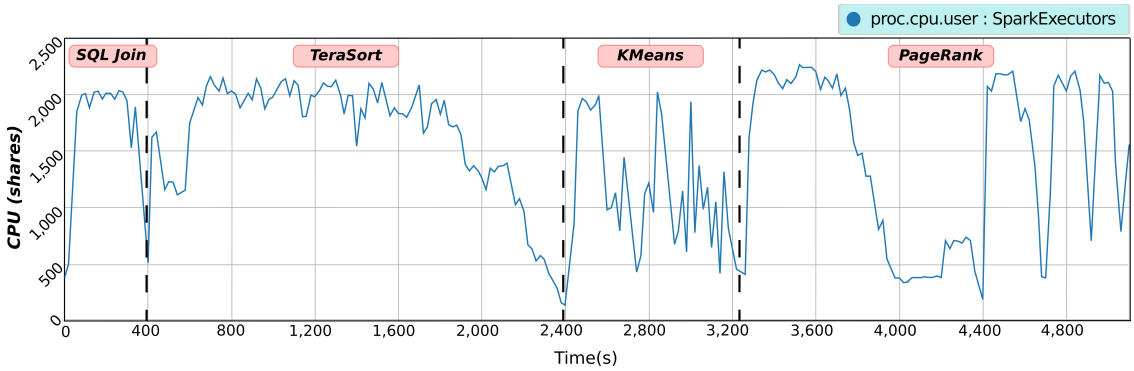


Figure 2.7: Aggregated CPU usage for Spark executors running batch workloads

present periodical phases of CPU utilization due to their iterative nature.

It is also possible to isolate the CPU pattern of a single workload such as KMeans, as shown in Figure 2.8. Taking advantage of the BDWatchdog tagging system, the aggregation has been performed for all the Spark executors across the cluster (see left graph) and for all the executors of a single container (right graph). Although just one container is displayed in this case for clarity purposes, all of them can be easily plotted in the same graph so as to identify unusual performance deviations by comparing them to the global aggregate or between themselves. In this case, CPU peak values of 2,000 shares are identified in the left graph, which means a maximum resource utilization slightly above 80%, considering the maximum value being 2,400 from adding up 6 containers with 4 cores each and 100 shares per core. On the right graph it is possible to spot local peaks of above 300 for the container *hadoop5*, which represents a utilization of 75%. This container would be an example of an underperforming container, however the difference is close to the aggregate and falls within an expected deviation.

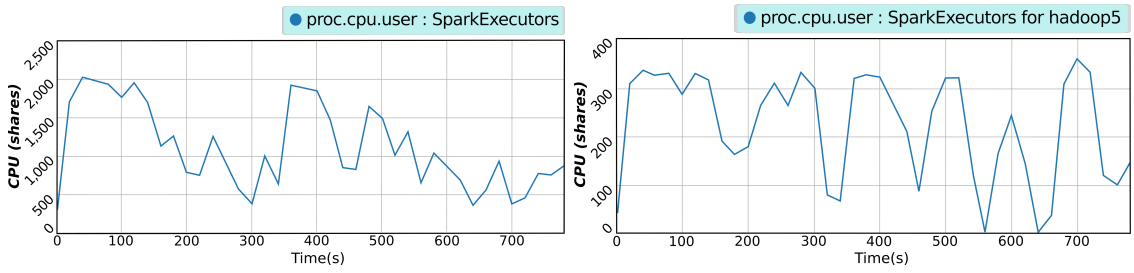
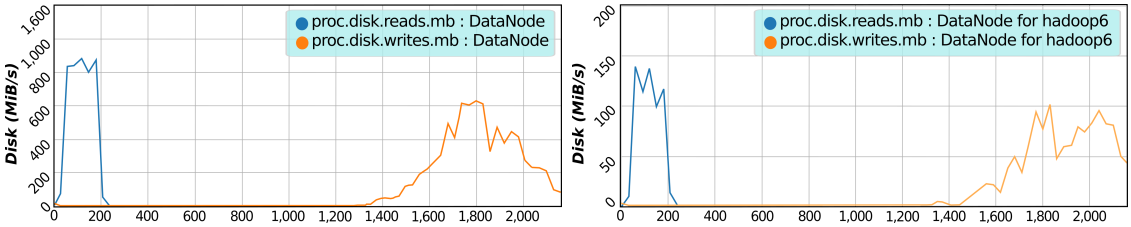
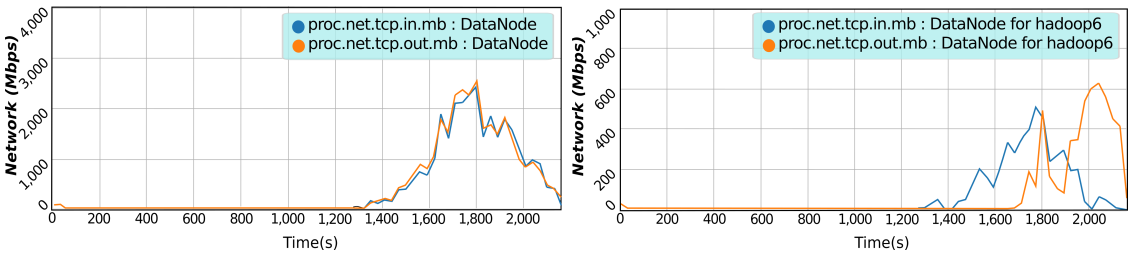


Figure 2.8: Aggregated CPU usage for Spark executors running KMeans



(a) Disk bandwidth



(b) Network bandwidth

Figure 2.9: Aggregated disk and network bandwidth for DataNode processes running Terasort

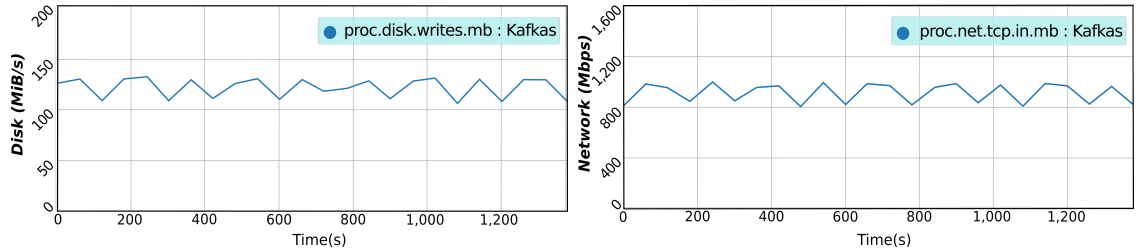
Figure 2.9 focuses on the Hadoop DataNodes during the execution of the Terasort workload analyzing disk and network usages for the entire cluster on the left graphs, and for one container on the right graphs. These resources are prone to be heavily used by DataNode processes for the data handling across the cluster, which may limit the overall application and Hadoop cluster performance. As only one DataNode is typically deployed per instance on a Hadoop cluster, the container graphs translate directly to one DataNode process.

Figure 2.9a shows the disk read and write bandwidths. Both cluster and container graphs show an initial read phase, where data are retrieved from the HDFS to perform the task, and a final write phase where the result is persisted. As this workload is executed using an in-memory data processing engine such as Spark, no data are necessarily persisted during the execution phase. It is during these initial and final phases that bottlenecks could appear for DataNodes. Both graphs show that the initial phase lasts for about 200 seconds with near constant values, which could mean that a bottleneck is taking place when retrieving the data, but further analysis is required to confirm this. In Figure 2.9b, the incoming and outgoing network traffic is displayed. It is easily identifiable the final write phase as mentioned for Figure 2.9a as the only period when network activity is present. In the left graph

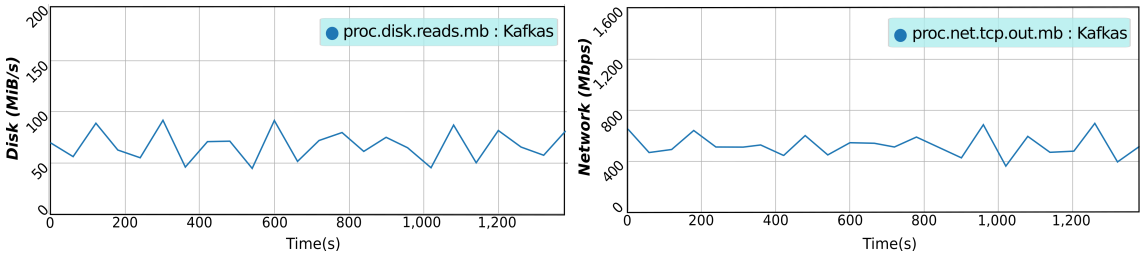
both input and output traffic match almost exactly, as all the traffic that is sent by a DataNode is retrieved by another. However, the right graph shows that for a single DataNode there are two phases, one where data are persisted through incoming traffic and another where some of these data are replicated to other DataNodes accounting for outgoing traffic. In this case, the replication phase for container *hadoop6* occurs after the persistence phase, but this is not necessarily the case for other DataNodes. Overall, no bottlenecks can be seen for the network, as the left graph shows network aggregate bandwidth around 2.5 Gbps (i.e., 5 Gbps in total for input and output traffic), far from the theoretical network limit (10 Gbps). Finally, it is also worth commenting the possibility of using the correlation between the network and disk graphs, which can be used to further diagnose the behavior of an application. In this case, it is possible to deduce that all the retrieved data during the initial read phase are local to the nodes, as no network activity takes place during that interval.

Figure 2.10 depicts the data movement for the streaming Repartition workload as handled by the Kafka cluster. Figure 2.10a shows the input of data in the Kafka broker. The left graph presents disk write bandwidth up to 125 MiB/s, which nearly perfectly correlates with the right graph that shows the network input traffic (1,000 Mbps $\approx$ 125 MiB/s). Similarly, Figure 2.10b shows how part of the data received is now sent to the Spark executors to be processed. In the left graph, read bandwidths up to 75 MiB/s also correlates with the outgoing traffic in the right graph (600 Mbps $\approx$ 75 MiB/s). As a whole, it is noticeable that after receiving and persisting the data to disk, they are then sent out although at a lower rate as requested by the Spark executors (i.e., the role of the broker). This situation has to be handled as the broker persists data for a configurable time window or it may ultimately run out of space, as data are not consumed at the same pace that are generated, causing a bottleneck.

This bottleneck is further analyzed in Figure 2.11, where the stream data processing is shown in the Spark executors. Figure 2.11a rules out any disk or network bottleneck. In the left graph, disk write bandwidth is displayed for all the Spark executors in the cluster together with the disk usage for all the disks of a single container. As the disk usage does not exceed 25% according to the sys.disk.usage metric that aggregates all the disks, it is unlikely that a bottleneck is occurring.

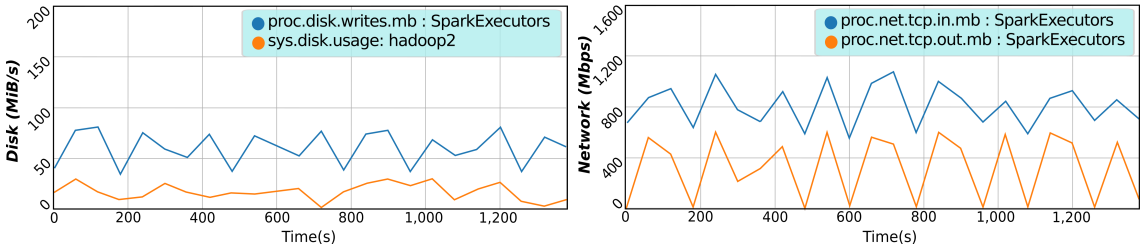


(a) Disk and network bandwidth for incoming data

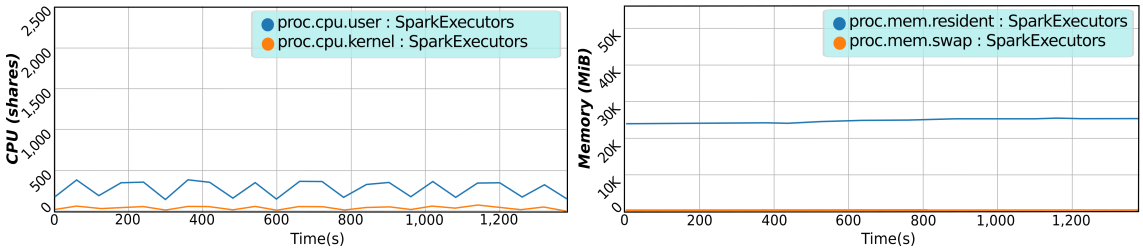


(b) Disk and network bandwidth for outgoing data

Figure 2.10: Aggregated resource usage for the Kafka broker while running Repartition



(a) Disk and network bandwidth



(b) CPU and memory usage

Figure 2.11: Aggregated resource usage for Spark executors running Repartition

Moreover, the right graph shows network aggregate bandwidths far from the network limit (i.e., 10 Gbps). Figure 2.11b also rules out any CPU (left graph) or memory bottleneck (right), considering the maximum memory value to be roughly

96,000 MiB (from 6 containers and 16,384 MiB each). So, the bottleneck may be the Repartition source code or the configuration of the workload itself.

2.5.2. Spotting code bottlenecks

As described at the end of Section 2.3.2, for the visualization of profiling data BDWatchdog allows to plot interactive flame graphs in SVG format that can be displayed in a web browser for a more user-friendly analysis. This section shows zoomed versions of the flame graphs focused on the JVM stacks that represent the Spark executors for the entire execution of a particular workload.

Figure 2.12 shows a flame graph for SQL Join. This graph is an example where clear hot spots are detected in the code. A Java class stands out clearly identified as *au.com.bytecode.opencsv.CSVReader*. The most executed methods of this class are *parseLine* and *<init>*, with a total execution time of 12% and 21%, respectively, relative to the JVM execution time.

On the other hand, Figure 2.13 shows the TeraSort workload. This graph reveals that the majority of the execution time is spent in JVM management code, specifically by the garbage collector. This represents probably an unintended code hot spot as it does not represent actual data processing. In this case, it is observable that the class *SpinPause* is accountable for over 30% of the execution time. This identified loss of performance may probably be due to a suboptimal configuration of

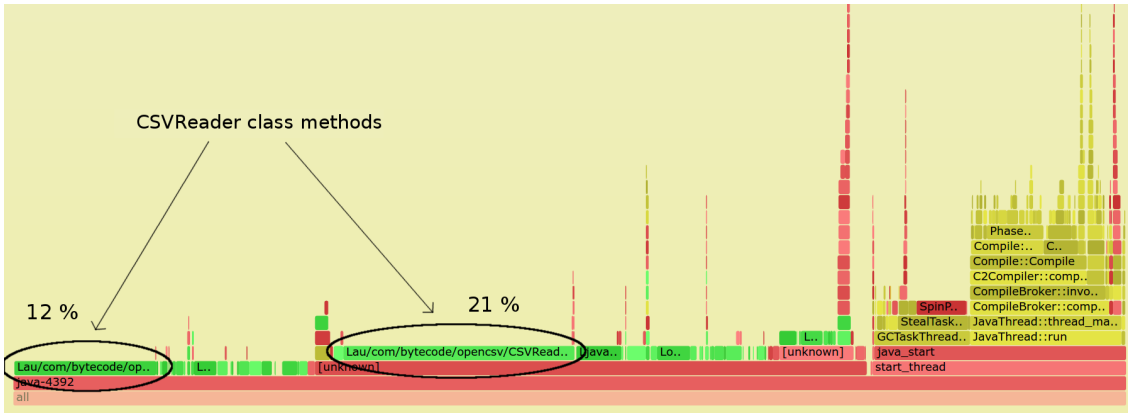


Figure 2.12: Flame graph for the SQL Join profiling

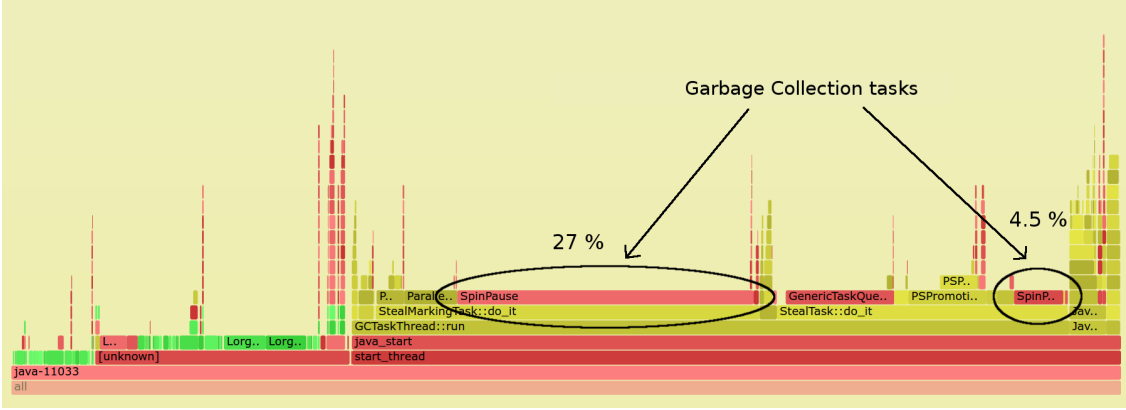


Figure 2.13: Flame graph for the Terasort profiling

the Spark workload or the JVM environment, being an example of a scenario where there is great potential for performance improvement if a fix can be applied.

2.5.3. Time window-based analysis

It can be of great interest for the user to analyze a certain time window, previously identified with a troublesome or resource-consuming phase of a particular application. This can also be used to better understand the operations performed by a workload in such period of time if profiling is used. BDWatchdog allows for this type of analysis, by combining both time series and flame graphs using the same time window for a better correlation.

Figures 2.14 and 2.15 present the analysis of a PageRank execution during one of the Spark stages that is performed: flatMap. Figure 2.14 shows the CPU usage aggregating the Spark executors of a container. This graph reveals that during this time window the workload achieves high levels of CPU utilization, which makes flatMap a compute-intensive stage that can become a potential bottleneck. On the other hand, Figure 2.15 presents a flame graph zoomed for a single Spark executor and using the same time window. As can be seen, a large percentage of the execution time (up to 30%) is being spent on the JVM management class *GenericTaskQueueSet*, while most of the classes that the workload uses (up to 12.4%) are related to *org.apache.spark.util.collection*. The combination of both monitoring and profiling confirms that although this flatMap stage presents a high CPU utilization,

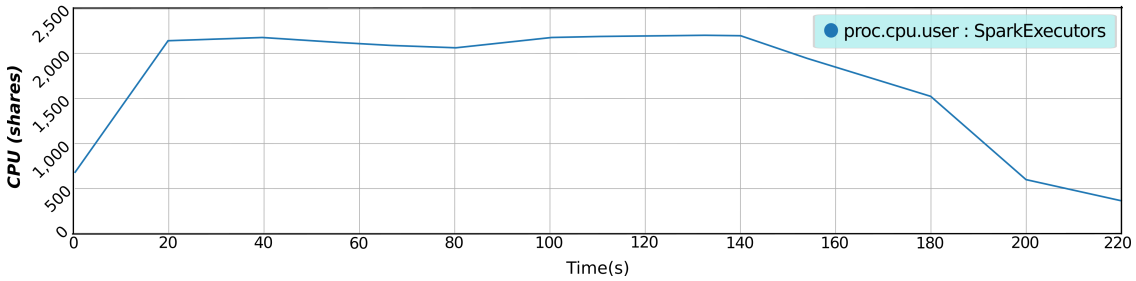


Figure 2.14: Aggregated CPU usage for Spark executors during a flatMap phase of PageRank

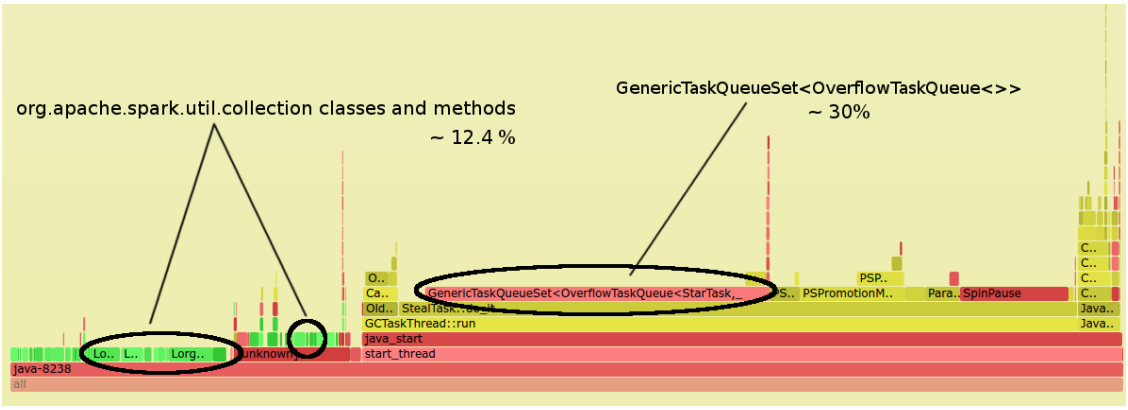


Figure 2.15: Flame graph for a Spark executor during a flatMap phase of PageRank

a significant percentage of the execution time is not being spent on actual data processing. In this scenario, further analysis could assess if such high percentage is reasonable or if it is the result of non-optimal configuration instead. Moreover, the Spark classes identified as the most executed ones can also be the target of further analysis for possible source code optimizations. Overall, the combination of monitoring and profiling allows to analyze the behavior of specific stages and assert that they are both resource and code efficient.

2.6. Conclusions

Big Data applications are nowadays very demanding in terms of resources due to both their potentially large size when deployed and to the many possible processing engines and paradigms currently available. In order to fully understand their

behavior and to assess their efficiency, more in-depth analysis tools may be useful. It is also interesting to take into consideration the support of a wider range of possible virtualization technologies, from bare metal to virtual machines and the currently popular software containers. To tackle these issues, it may be desirable to move further from traditional process-based analysis tools and to design, implement and deploy new ones that, while remaining scalable, offer richer and advanced functionalities.

This chapter has presented BDWatchdog, a tool that is capable of real-time analysis of Big Data frameworks and workloads through a novel combination of fine-grain, per-process resource monitoring and window-based, low-level code profiling. For accurate resource accounting, time series are provided, which can be used flexibly through selection and aggregation operators to account for single processes, groups of a same process across a cluster (e.g., Spark executors), or all the processes on an instance. Similarly, for accurate application code analysis, low-level profiling is used, which does not require to alter the application code, is less intrusive, and can be used continuously on the background. Furthermore, this profiling solution is able to mix system and JVM profiling data, providing a broad picture of the code executed. In order to support these functionalities and at the same time remain scalable and work in real-time, a specific architecture has been designed and implemented.

Finally, the use cases exposed have shown that BDWatchdog allows to analyze an application or framework with a unified view in terms of resource consumption and code executed. Using resource analysis, it is possible to spot bottlenecks, usage patterns, or account for resource utilization. Code analysis allows to further study the performance of applications, even when resource accounting does not show any bottleneck or room for improvement, by either looking for optimizations in the source code that is most executed or by spotting JVM overheads (i.e., long periods of time spent in JVM internal management). Regarding the overhead that BDWatchdog imposes on the applications, the experiments have shown how it can be tuned between a more aggressive or relaxed configuration, considering nonetheless that most of the overhead is imposed by profiling.

The BDWatchdog framework is available at `http://bdwatchdog.dec.udc.es`.

Chapter 3

Real-time resource scaling on serverless environments

Even though the first Big Data technologies and applications were originally conceived to be deployed and executed using commodity hardware on dedicated clusters, currently it is common to have such applications deployed on a wider range of platforms. This deviation makes sense if we consider several possible scenarios for Big Data that came with its widespread adoption, such as: 1) applications that are part of a larger and more complex process (e.g., Business Intelligence); 2) testing or proof-of-concept deployments; 3) applications that have to run alongside other technologies such as HPC, queue managers or a private Cloud platform; and 4) event-based or on-demand deployments. All these use cases share the need for a more flexible platform for resource allocation and execution, which is also commonly solved by using both private (e.g., OpenStack [93]) and commercial public Clouds (AWS, GCE, Azure). Nevertheless, even current Clouds keep evolving with new paradigms that try to improve different aspects such as user-friendliness, faster deployments, increased efficiency, or simply new ways of managing and billing the resources. Among these new paradigms that look into improving some of these aspects we can find the serverless one [15, 24, 92, 104, 105].

The serverless paradigm deeply changes both the way applications are executed and the management and billing of the resources used if compared to IaaS/PaaS, as code execution is abstracted from the underlying infrastructure, thus its name. For

applications deployed on a serverless platform, it is expected that: 1) they are only given the resources strictly needed and used by them; and 2) if the user is billed for the resource usage like in public Cloud scenarios, only the amount actually used should be accounted for. If both conditions apply, the user ideally only pays for the resources consumed by the application, while also avoids suffering any performance penalty from the infrastructure management. Although several options exist to execute easily adaptable applications or plain functions as code on a serverless environment, there is still little research that shows how more complex, distributed applications (e.g., Big Data workloads) behave in a scenario where resources could vary according to the workload demands.

In this chapter we propose a novel platform that is capable of dynamically and continuously scale the resources used by Big Data applications hosted in container-based clusters. This platform provides users with fully capable instances like in a Cloud scenario but with a resource management that can follow different tuning policies, including the serverless paradigm. Such dynamic scaling is possible by relying on real-time, precise container-based resource monitoring, as provided by BDWatchdog, and a feedback loop architecture, which is able to be dynamically configured as needed and scale according to the size of the infrastructure to be managed. Several representative Big Data workloads are deployed on this platform to analyze their resource usages when they are applied real-time resource scaling according to their real usage, that is, when they are executed on a serverless environment. Furthermore, we aim to prove that for CPU and memory resources the usage amount is generally independent and loosely coupled from the available or reserved amount. This in turn implies that, from the user's point of view, when a serverless environment is used there should not be significant differences in performance or overall behavior.

The rest of the chapter is organized as follows: Section 3.1 introduces the background and the terminology used throughout the chapter. The current state of the art along with the related works are presented in Section 3.2. Section 3.3 describes both the design and implementation details of the architecture developed for this platform. Section 3.4 presents the procedure used to perform two resource analysis functionalities: a real-time scaling while the workload is running and a subsequent resource accounting once the workload has finished. All the configuration options

are also explained in this section. The experimental configuration and the analysis of results are presented in Section 3.5. Finally, conclusions are exposed in Section 3.6.

3.1. Background

To properly understand the chosen technologies and architecture design that make possible our resource scaling platform, some specific clarifications have to be made regarding container-based virtualization solutions (Section 3.1.1) and the serverless paradigm as a whole (Section 3.1.2).

3.1.1. Container-based virtualization and monitoring

In order to lighten the performance overhead of virtualization and provide real-time scaling of resources, while still offering some kind of application isolation, Linux Containers (LXC) [19] are used. Such containers combine the use of namespaces and cgroups, which together allow to create isolated environments where processes can be executed and file systems deployed. With LXC in particular, it is possible to deploy containers that replicate any Linux-kernel-based operating system (e.g., Debian, CentOS), being such containers very close from the user's point of view to a hypervisor-based virtual machine. Nevertheless, it is important to note that the platform presented in this chapter could be easily integrated with other container technologies also backed by cgroups like Docker [81], as the resource scaling is ultimately done at the kernel level by setting appropriate values to the corresponding cgroup files for CPU and memory limits.

Unfortunately, the fact that software containers share the kernel among all the containers running on the same physical host means that some caution is required for any operation that accesses the kernel or collects system-wide information. An example of this is traditional resource monitoring, which is typically carried out by tools (e.g., top, collectl) that report overall resource usage and optionally per-process metrics. If looking for aggregated container usages in a system-wide manner, it is not advisable to execute such tools inside a container as they most probably report the host's usages rather than the container ones. Nevertheless, these tools can still be

used for process-based monitoring and to produce a container resource usage report via metric aggregation. For our platform, we rely on the BDWatchdog framework presented in Chapter 2, which offers this exact feature. With this solution, resource usage metrics for CPU, memory, disk and network are collected and stored as time series in real time and on a per-process manner.

By combining LXC and BDWatchdog we have the basis to execute any application or workload on a virtual infrastructure composed of software containers that imposes a minimum virtualization performance overhead and is accurately monitored.

3.1.2. Serverless paradigm

The serverless paradigm is currently presented as a novel platform for application execution. Its main difference from already existing paradigms like IaaS (e.g., Amazon EC2 [4], Google GCE [52]) or PaaS (e.g., Amazon ECS [5]) is that the user is abstracted from the underlying infrastructure and has no control over it. While either a hypervisor or a container manager is used in the aforementioned IaaS/-PaaS services, in a serverless environment the user may not be given access to the underlying infrastructure as the provider reserves the right to make the resource management in order to maximize resource utilization. As a direct consequence of this, the user typically does not choose the amount of resources allocated for the application and is only charged for the used ones. Although this billing policy is also advertised for IaaS/PaaS services, it is crucial to point out some differences and present some terminology that is later used.

On the one hand, with IaaS/PaaS the user asks for a resource amount to be given for an instance, from now on 'reserved' resources. These resource limits do not change while the infrastructure is running and are in the end billed according to the time they were allocated. On the other hand, with serverless the user executes an application and is only billed for the resources that it used, from now on referred to as 'used' resources. Because the underlying infrastructure is abstracted, there are no 'reserved' resources in the serverless scenario. In case of an application that is idle, the user would still be charged for the 'reserved' resources in an IaaS/PaaS service, while would only be charged for the minimally 'used' resources on a serverless

platform. As a whole, this difference is the main advantage for the users as it can significantly lower the costs for applications that adapt to the serverless paradigm, as is the case for web applications or event-based workloads. Other advantages are quicker deployments or increased flexibility, taking benefit of the abstracted infrastructure.

However, in the end, even applications deployed on a serverless platform need an underlying infrastructure to be executed, thus it is only abstracted from the user's point of view. So, it is the task of the service provider to allocate the necessary resources for the application, taking into account that the allocated but unused resources do not return any profit as they are not billed. Considering that containers are used in our platform, whose resource limits can be changed at any moment, from now on we will refer to these resources as 'allocated'. In addition, it is worth to mention that while the service provider might look for a high server consolidation to minimize any non-billable resources, at the same time it must abide by the Service Level Agreement (SLA) with the user and guarantee a minimum performance at all times, whether the application is idle or not.

Finally, it is necessary to introduce some additional terminology to describe components or operational processes of our platform. Up to this point we referred to applications as abstracted workloads that are deployed and executed in any form of infrastructure. However, from the point of view of the proposed platform, an 'application' is composed of a set of 'containers' that jointly execute a workload. Although this abstraction is not actively used for the experimentation, which targets the scaling on each container individually, it is still useful as an aggregated view for any subsequent analysis. In addition, it is important to remember that containers have to be deployed and executed in an underlying infrastructure, a 'host', usually a dedicated instance with a large amount of resources. Along with the containers, we consider applications and hosts as 'structures' in our serverless platform, as ultimately they all represent some form of infrastructure unit. For example, all of them share the need for an accounting of each resource, whether such information is to point out the allocated or total deployed amount (e.g., a container or application structure, respectively), or the unused amount (e.g., a host structure).

3.2. Related work

There are several fields of work related to our platform, from the Function-as-a-Service (FaaS) paradigm (Section 3.2.1) to novel research frameworks and solutions that aim to extend FaaS to support applications deployed as clusters of containers (Section 3.2.2). Lastly, it is interesting to study more closely the dynamic scaling of applications (Section 3.2.3), already present in established technologies like the Cloud.

3.2.1. FaaS platforms

Currently, most serverless products offered by major Cloud providers tend to focus on specific use cases around the FaaS paradigm, such as Amazon Lambda service [8], Google Cloud Functions [51] or Azure Functions [82]. With these products and the FaaS overall, small programs are executed with properties similar to functions (i.e., the applications do not have an internal or persistent state), but with the difference that they are abstracted from the underlying infrastructure that actually hosts them. Service providers guarantee availability and scalability by automatically resizing the infrastructure as needed, and only charge the user for the resources really used. Unfortunately, this also means that the user may not be given access to the instances or infrastructure used to host the running functions, as by design the use cases do not usually require this. Furthermore, it is not possible to execute complex types of applications (e.g., Big Data workloads) as the platforms require adherence to predefined templates thus making the integration incompatible. Besides commercial platforms, there are also open solutions for the FaaS paradigm currently being developed, like OpenLambda [59] and OpenWhisk [16, 103], but they have similar limitations.

3.2.2. Containerized applications on serverless scenarios

While the previously described serverless platforms restrict the user to running small tasks following some guidelines, several authors have found the serverless paradigm interesting to be extended to more general-purpose use cases. To do

this, applications are commonly bundled using container images as the means to easily package and distribute them, and are then instantiated to be executed under any container manager platform (e.g., Kubernetes [19], LXD [76]). By using the container as the smallest infrastructure component, instead of an abstracted entity that represents a piece of code like in FaaS, it is possible to support more types of applications which can be in turn much more complex. As the literature suggests [15, 24, 92, 104, 105], serverless is one of the trends to follow in the future when it comes to applications that present burst-like behavior, event-based processing or simply adapt to predictable patterns. The applications already deployed in FaaS solutions are included in such scenarios, but efforts are being made to adapt other kinds of workloads to the serverless paradigm, as described next.

In [77], a scientific workload from high performance computing environments is successfully deployed on FaaS platforms. Interestingly, it is pointed out that in the future a better resource management should be offered by the service provider, considering that with the serverless paradigm the user loses control on this matter. User hints are proposed as a means to guide and modulate such resource management and possibly auto-scaling. Additionally, a cost/performance analysis is presented, showing that for some applications there is neither a performance penalty nor a higher cost. In a similar way to this work, we look forward to adapt Big Data workloads to a serverless scenario where resource scaling is present, assessing how they behave in terms of performance and resource usage, something important in case they had to be billed for.

Other works present frameworks that aim to run applications in clusters of containers on a serverless environment using Cloud infrastructures. In [87] the authors describe SCAR, a solution to deploy several kinds of applications through Docker containers in the Amazon Lambda service in order to benefit from its scalability and fault-tolerance features. It is also further proved that containers can be integrated into an existing serverless platform and used to better suit the user's needs, as they are more flexible and richer than standalone pieces of code. Nonetheless, we consider that although this approach is interesting, there should not be a need to integrate an already mature and standalone technology like Docker with a serverless platform like Amazon Lambda. In our solution, containers are deployed and monitored natively by the serverless platform, with the resources being scaled dynamically to cater for

the application's requirements, without the need for any integration or rebooting process.

Finally, it is also worth noting that while the Amazon Lambda service imposes limits, like a maximum timeout of 300 seconds for the functions to finish, our platform does not impose any limit and the containers can be maintained with minimum resource usage when idle while waiting for work.

3.2.3. Automatic resource scaling

Previous work regarding automatic resource scaling has already been explored extensively for virtual machines backed by hypervisor-based virtualization, proving that it is interesting both for users and service providers due to lower costs and higher resource utilization ratios. In [57, 58] the authors present a system that is able to automatically scale applications running in several virtual machines by either scaling the number of instances or the amount of resources. This scaling is done by using several algorithms, policies and thresholds to scale up or down, in a similar way to our platform.

Newer studies focus on using containers instead of virtual machines due to their advantages including the possibility of real-time resource scaling. Dhalion [48] is a framework capable of automatically scaling applications running on top of Apache Heron [71]. With this solution, the workload's health is monitored through a series of metrics in order to detect symptoms that may be the cause of performance losses. Among the issues monitored for are both the resource overuse or underuse, which can then be treated to achieve a higher resource utilization without incurring any performance penalty. Similarly, Trevor [17] expands on the Dhalion solution using models created and trained using metric measurements from applications to automatically and continuously search for a configuration that optimizes their execution, even while running. As with our platform, Trevor is able to deal with changing resource demands by either increasing or decreasing the amount of resources. However, due to the fact that Trevor is implemented on top of Apache Heron, the focus is put more on the number of instances (horizontal scaling) than on the size of them (vertical scaling). As opposed to Dhalion or Trevor, our platform aims for a more flexible implementation that supports the dynamic and continuous scaling of containers with

no specific restriction for any underlying solution or technology, thanks to relying on the use of cgroups. By using the container as the minimal infrastructure instance unit, our platform provides the user with the freedom to deploy a broad range of applications ranging from a few containers to a large cluster.

When it comes to commercial products, there are several available solutions in the Cloud environment to automatically adjust the user's infrastructure to meet the application's demands at any time. Amazon Auto Scaling [2] is a representative example that fully integrates with several other Amazon services like EC2/ECS and with Amazon CloudWatch [3], which acts as the monitoring daemon that feeds usage metrics used for the scaling policies. Unfortunately, only horizontal scaling is possible by adding more instances. Note that this combination of resource usage information and the Auto Scaling service accomplishes the same purpose as combining BDWatchdog and our resource scaling platform, as previously mentioned in Section 3.1.1.

It is also interesting to further study Amazon ECS [5], a service that allows to run containers. ECS supports two modes for container deployment and management: EC2 and Amazon Fargate [7]. On the one hand, with EC2 the user chooses an instance type to host the containers, being the access granted to such instance. Once the instance is deployed, it is possible to automatically scale the application relying on the Auto Scaling service by adding or removing containers according to predefined user policies. However, the resources of an EC2 instance cannot be changed without a reboot. So, if there is a need to further increase the number of containers hosted, either an additional EC2 instance has to be spawned or the one in use has to be stopped, scaled and restarted. Regarding the billing policy, the EC2 instances are charged according to their type. On the other hand, Fargate takes over the task of choosing an EC2 instance type to run the containers and further abstracts any infrastructure management. With Fargate it is possible to deploy an application specifying only the initial resources it needs and the number of containers to use. In contrast with EC2, only the used resources are billed, although at an incremental rate. Unfortunately, with Fargate the user does not have access to the underlying instances or infrastructure used. Although the difference between both modes mainly lies in the billing policy and the access to any underlying instance, the first difference is of particular interest for the user and was previously exposed in

Section 3.1.2. In contrast, our platform implements automatic scaling of resources without any need of rebooting, while preserving the access to the underlying containers. If billing is to be applied, it is possible to account for the used resources as well as for the allocated ones.

3.3. Architecture design and implementation

The objective of our platform is to create a serverless environment where Big Data applications are executed with resources being allocated on demand according to their real usage. It is also desirable that the platform does not impose a significant overhead while being scalable and remains overall autonomous.

Figure 3.1 represents a high-level overview of the system and the environment on which it works. Lying at the center is the resource scaling platform, whose only inputs are the control actions taken by either a user or another system and the information about resource usage, as provided by external monitoring tools. For this requirement, the monitoring features of BDWatchdog, as described in Chapter 2, are used to feed such information in the form of time series, which after being stored in the *Time series Database* can be later retrieved by any service that needs them. In this case, OpenTSDB [112] is used to persist and expose the time series. The

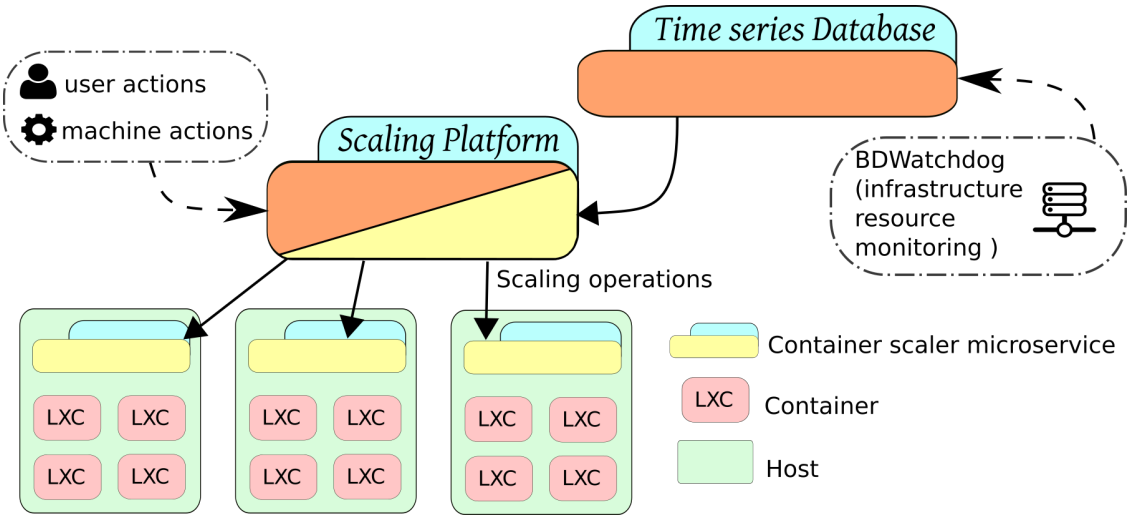


Figure 3.1: High-level overview of the resource scaling platform

output of the platform is represented by scaling operations sent to each infrastructure host, on which a container scaler microservice forwards such operations to the container manager or to the containers themselves via cgroups. The specific details of the architecture and technologies used to implement this platform are described in Section 3.3.1. An in-depth analysis of the feedback loop that lies at the core of the platform is presented in Section 3.3.2. Finally, a scalability discussion is provided in Section 3.3.3.

3.3.1. Microservice-based architecture

The platform architecture (see Figure 3.2) is designed around the idea of being able to process events in real time and acting quickly enough to respond accordingly. To do so, a microservice-based architecture is chosen along with a feedback loop arrangement for the services. The platform developed from this architecture design is able to self-regulate, thanks to the loop created, and to exploit the inherent parallelism of the microservice division, which in turn allows to work in real time. Furthermore, thanks to having the possibility of isolating the services and to the use of the REST protocol for inter-service communication, the development and testing of the individual services is much simpler.

In order to perform the service domain division and thus break down a complex loop into smaller systems, which can then be easily managed or configured, a global intermediary service is used (see *State Database* in the figure). A document database, CouchDB [10], is chosen to act as the agent to keep the whole container and microservice metadata and thus act as such intermediary service. Using this light, document-based database, all the platform state information is kept in a single point, allowing for an easier supervision as well. This centralized configuration of the services means that they can be implemented fully stateless and with a simpler, daemon-oriented design. It has to be noted that although the *State Database* represents a single point of failure, this can be mitigated by taking into account the different features implemented in CouchDB, such as load balancers or data replication, which readily grant high availability. It must also be considered that such database only acts as a temporary means of holding the platform state, which is usually reduced to a low amount of information even if a large infrastructure were

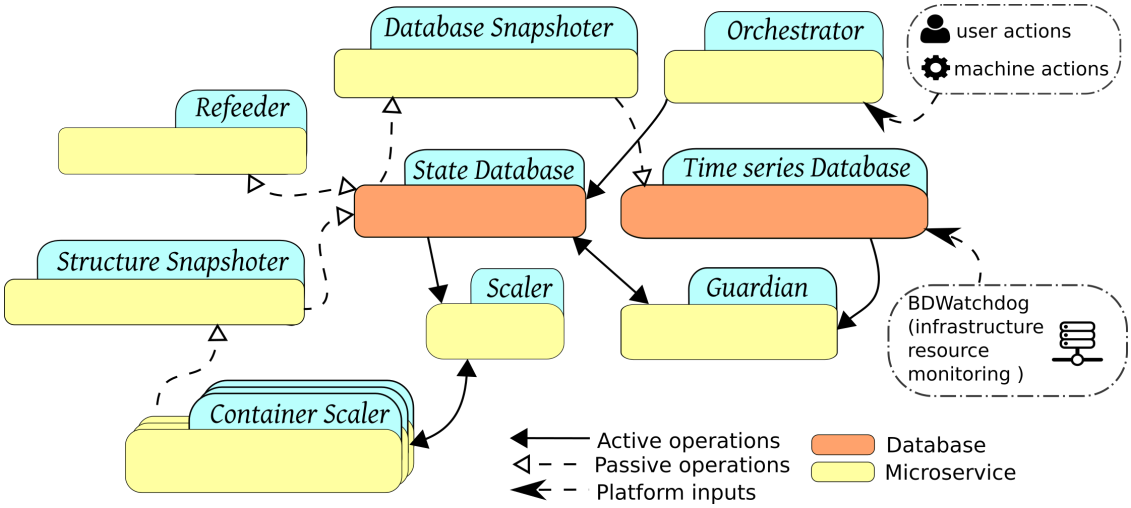


Figure 3.2: Microservice-based architecture

to be deployed, and thus a recovery and startup process could be easily carried out with a new database.

All the microservices shown in Figure 3.2 are implemented with Python following the REST philosophy, using JSON as the information format to exchange data. By using such format it is also possible to directly store the information in the *State Database* as JSON documents. In addition, thanks to their more human-readable characteristics, the administrator can inspect and modify such documents (e.g., a service configuration document) if necessary.

3.3.2. Feedback-based loop control

In order to maintain stability and to avoid high overheads to the running applications, the platform must minimize the time ranging from the moment a resource usage event is detected (e.g., a bottleneck, when the usage is close to the allocated amount) to the moment a request is generated to tackle such situation (e.g., a scaling-up operation, which increases the allocated amount). Benefiting from the domain division mentioned in Section 3.3.1, which causes the services to be highly specialized, the loop behavior can be explained by analyzing both the services deployed and the information that is generated and exchanged.

Regarding the services (see Figure 3.2), they can be classified into those that

are active, meaning that they take actions to change the platform state, and those that are passive, only generating information that refeeds the system constantly. On the one hand, the active services are: 1) *Guardian*, the core service responsible for the time series analysis so as to generate events and requests; 2) *Scaler*, which processes such requests and then forwards them to the hosts and their corresponding *Container Scaler* services; 3) *Container Scaler*, the service that runs on each physical host and exposes a REST API through which requests are applied to the containers; and 4) *Orchestrator*, which provides a REST API that allows users or other external services to make changes to the platform configuration. To implement both REST APIs, Flask [55] is used along with Gunicorn [56] to expose them via a WSGI-compliant server [49]. On the other hand, the passive services are: 1) *Structure* and 2) *Database Snapshoters*, which when combined persist the container metadata both in the *State Database* to refeed the loop and in the *Time series Database* for later analysis; and 3) *Refeeder*, the service responsible for creating the aggregated metrics that represent an application from its containers. Further details about the functionalities of both active and passive services are explained in Section 3.4.1.

Regarding the information used and generated throughout the platform, it can be classified into time series and documents. On the one hand, the most important time series are the application resource usages generated by BDWatchdog, as mentioned before. These act as the main source of information of the platform as they are needed to overall detect the events that will later trigger the requests to scale the resources. Secondly, the platform also generates time series by periodically persisting relevant documents from the *State Database* to the *Time series Database*. Although not so important, this information is used for later analysis and reporting. On the other hand, documents are used throughout the platform, mainly by the active services, as the basic means of interchanging information, although are not persisted over time. Documents are classified into: 1) 'structures', which represent the metadata of either a host, a container or an application, mainly storing the amount of allocated resources; 2) 'services', the per-service, uniquely stored document containing its configuration; 3) 'rules', which store the policies to be applied for time series analysis; 4) 'limits', having for each container or application structure the resource values for the upper and lower thresholds, as well as the boundary to be kept between them; 5) 'events' and 6) 'requests', the resulting documents after the continuous resource analysis, as explained in Section 3.4.1.

```
1 "type": "structure",
2 "subtype": "container",
3 "scale": true ,
4 "host": "c10-10",
5 "host_scaler_ip": "c10-10",
6 "host_scaler_port": 8000,
7 "name": "cont2",
8 "resources": {
9   "cpu":  { "allocated": 300,  "scale": true , "max": 300,   "min": 50   },
10  "mem":  { "allocated": 8192, "scale": true , "max": 10240, "min": 1024 },
11  "disk": { "allocated": 100,  "scale":false, "max": 100,   "min": 20   },
12  "net":  { "allocated": 200,  "scale":false, "max": 200,   "min": 100  }
13 }
```

Listing 3.1: Example of a structure document

Structure documents (see an example in Listing 3.1) store crucial information such as the maximum and minimum values for each resource, which correspond to the amount of resources that a structure (e.g., an individual container or an application) never surpasses or is always given, respectively. They also store the allocated amount for each resource (see lines 9-12). Regarding rule documents, it is interesting to mention that although stored as JSON files, they contain the logic ultimately used to define the scaling policies. Thanks to the use of JsonLogic [66] it is possible to inject complex boolean rules and dynamically evaluate them in the continuously running loop. This feature grants the system flexibility when it comes to changing the configuration in real time. The rest of documents are more precisely explained in Section 3.4 by using an example.

3.3.3. Scalability discussion

After having discussed the design and inner workings of the platform, it is interesting to address potential scalability limitations and improvements. This is even more important when containers are deployed, which can typically be smaller in terms of resources when compared with virtual machines, and thus a significantly larger number of containers could be running at any moment. In addition, if Big

Data workloads are to be deployed on such virtual infrastructure based on containers, it is increasingly possible that a large number of them have to be managed.

From the design point of view, it is possible to analyze some features that enhance the scalability of the platform. First, the use of a microservice approach makes it possible to have not only the inherent parallelism of several services working simultaneously, which mitigates the effect of any local bottleneck, but also the replication of such services as needed to exploit the parallelism of their operations. For example, as can be seen in Figure 3.2, the active operations performed by the *Scaler* can be potentially parallelized by splitting them either across several threads inside an instance or even across multiple instances of the service. This behavior also applies to other critical services such as the *Guardian* or the *Snapshoters*. Second, and related to the parallelism of services and operations, the metadata database used to store the documents could also benefit from a scenario with several database instances, with each instance storing the documents of a part of the managed infrastructure, thus creating some sort of domains which would be independent.

Nevertheless, considering that our scenario is mostly experimental, the scalability of the platform is further analyzed with empirical data and the use of specific experimentation, as presented in Section 3.5.5. Finally, it should also be considered that the platform could benefit from better hardware specifications or newer technologies in order to improve its internal response time to events, which would help it to further scale.

3.4. Resource time series analysis

This analysis is at the core of the platform, having a twofold role. First, while the applications are running, the platform continuously monitors the infrastructure to determine if there are scaling actions to be performed in real time according to the policies set by the user. Secondly, it also gives more insight into how resources are used in any past time window. Considering that the system is continuously persisting the most important information, such time windows can be really close to present, which in turn allows to perform any analysis or detect performance issues quickly after they arise. Finally, after an application is executed, a subsequent

resource analysis can provide fine-grain detail regarding how resources were used.

Both functionalities, real-time resource scaling and subsequent resource analysis are respectively described in Sections 3.4.1 and 3.4.2. In addition, these functionalities are extensively referred to in Sections 3.5.3 and 3.5.4, respectively, where they are used to assess the platform efficiency and to study the behavior of the resource usage for different Big Data workloads. Finally, all the configuration aspects of the platform are described in Section 3.4.3.

3.4.1. Real-time resource scaling

A continuous analysis is performed while the application is running, requiring all the platform services to implement the feedback loop. To start any scaling operation, the *Guardian* service performs a two-step analysis of the resource usage metrics. To do so, first this service retrieves the resource time series from the *Time series Database*, which were previously collected by BDWatchdog, and matches them against a set of event-triggering rules. This may generate events (see number 1 in Figure 3.3), which are stored in the *State Database*. Second, all the events are retrieved and, after filtering out the expired ones, they are equally matched against a set of request-triggering rules, which generates in turn requests (see number 2), also stored in the *State Database*. Meanwhile, the *Scaler* is continuously polling the *State Database* for any request (number 3). Once new requests are found and after filtering out any duplicated or expired ones, they are processed and applied to the structure resources before being actually sent to the *Container Scaler*, which ultimately applies the request to the container (number 4). This additional processing carried out by the *Scaler* is needed as an intermediary check to ensure that no request, if applied, would cause the resource amount to exceed a maximum or drop below a minimum value. Requests can even be discarded if the underlying host of the container has run out of resources to be allocated.

In parallel to the active services and in no particular order, the passive ones are continuously propagating the actually allocated structure resources (see number 1 in Figure 3.4), generating the aggregated application resource metrics (number 2) and ultimately persisting the structure metadata (resources and limits) in the *Time series Database* for later analysis (number 3).

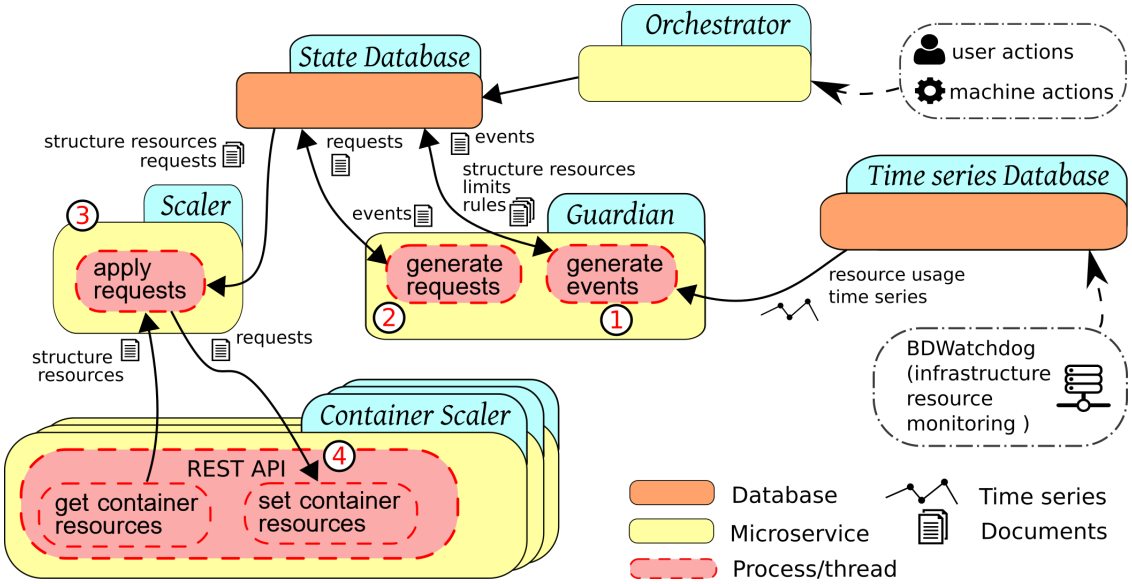


Figure 3.3: Active services

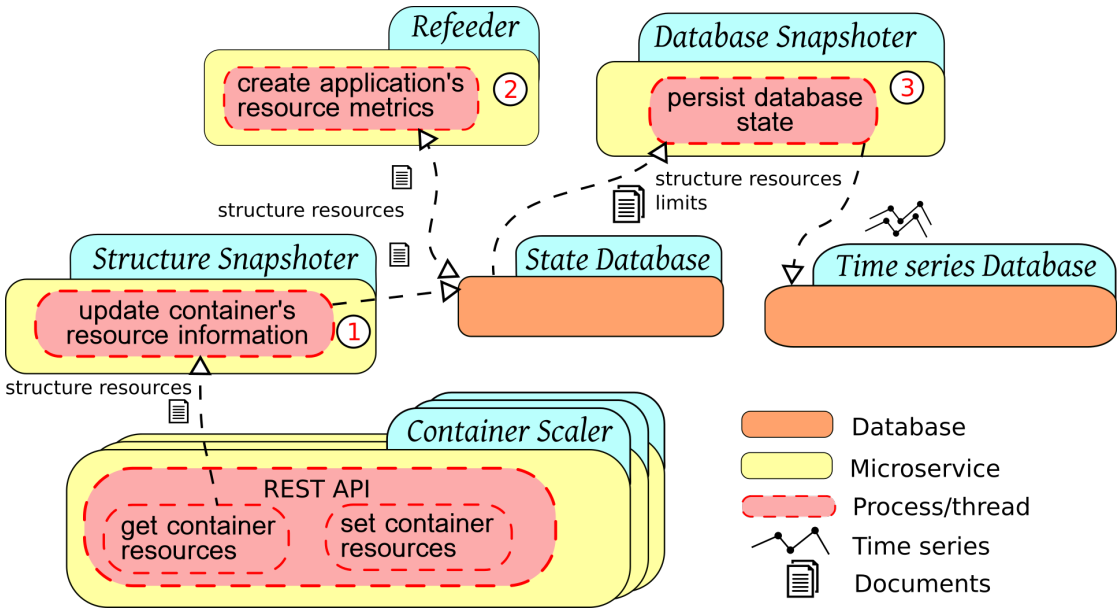


Figure 3.4: Passive services

The described two-step analysis to generate requests is designed and implemented in order to mitigate 'hysteresis' and provide the option to modulate the 'responsiveness' in the platform configuration. Hysteresis, which appears when a resource usage is constantly oscillating around a fixed range, could severely affect the platform if

scaling operations took place once any limit was crossed. If such limits are not wide enough, the oscillating usage would constantly exceed them and cause constant scaling operations, and overall an unstable state. With this two-step analysis, hysteresis is mitigated thanks to the fact that rules that trigger requests can be configured to trigger only when a defined number of events of a certain type (e.g., a bottleneck) are detected, and none or a defined number of opposite events (e.g., an underuse) arise. Regarding responsiveness, considering it as the time it takes the platform to respond and adapt to the application resource usage, it can be modulated by changing in the rules the number of events needed to trigger the corresponding requests. This configuration feature is further described in Section 3.4.3.

To better understand the function of the active services and overall the real-time resource scaling, Figure 3.5 shows an example where CPU scaling requests are applied to a container represented by the structure shown in Listing 3.1 (CPU maximum/minimum values: 300/50 shares). While from roughly second 60 to 220 the CPU usage fits into the normal scenario (i.e., between the lower and upper limits) and nothing is done, from 220 to 270 the CPU usage exceeds the upper limit. At this point, the rule shown in Listing 3.2, which detects CPU bottlenecks, is triggered. Specifically, this is done when both the CPU usage is higher than the upper limit (see the first condition of the 'and' logic in line 8) and the allocated CPU amount is lower than the maximum allowed (see the second condition in line 9). This rule is activated several times before ultimately triggering the rule presented in Listing 3.3. This latter rule is specifically triggered after 2 CPU bottleneck events (see the first condition in line 10) and less or equal to 2 CPU underuse events are detected (see

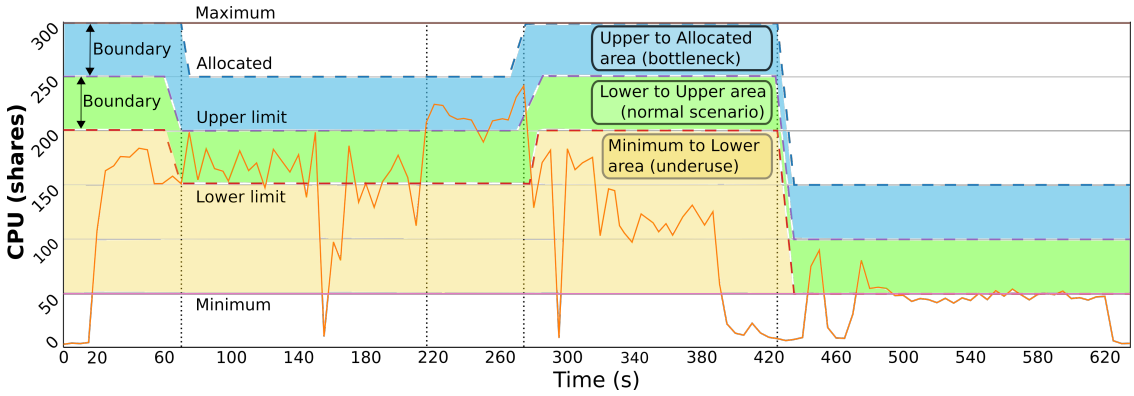


Figure 3.5: Continuous resource analysis with areas for scaling operations

the second condition in line 11), generating in turn a request to scale CPU shares upwards. However, after this resizing, from seconds 280 to 420, CPU shares are not fully consumed incurring thus in an underuse scenario. As in the previous bottleneck scenario, this triggers a rule that detects underused resources and, after a certain number of such events, a CPU scaling-down operation is performed. From second 420 onwards the container is given the minimum resources as the lower limit is set to the minimum value (50 CPU shares), and thus no underuse events are possible anymore.

```
1 "action": {
2     "events": {"scale": {"up": 1}}
3 },
4 "generates": "events",
5 "name": "cpu_exceeded_upper",
6 "resource": "cpu",
7 "rule": {
8   "and": [{">":[{"var":"structure.cpu.used"},{"var":"limits.cpu.upper"}]},
9      {"<":[{"var":"structure.cpu.allocated"},{"var":"structure.cpu.max"}]}]
10 }
```

Listing 3.2: Rule to generate CPU bottleneck events

```
1 "action": {
2     "requests": ["CpuScaleUp"]
3 },
4 "amount": 50,
5 "generates": "requests",
6 "name": "CpuScaleUp",
7 "scale_by": "amount",
8 "resource": "cpu",
9 "rule": {
10   "and": [{">=": [{"var": "events.scale.up"}, 2]},
11           {"<=": [{"var": "events.scale.down"}, 2]}]
12 }
```

Listing 3.3: Rule to generate CPU scaling-up requests

It is important to mention that the behavior of scaling requests is different according to the request type. If the resources are to be reduced, there are several options, as it is possible to know beforehand the amount of resources the structure has recently been using up to that point. Several simple policies are supported such as always reducing by a fixed value or reducing by a percentage of the unused amount. Nevertheless, in order to fit the allocated resources closely to the used ones, a more complex 'fit to usage' policy is implemented. This policy tries to set a specific allocated resource value with the aim of making the application usage stay between the lower and upper limits (see this effect from second 60 to 220 in Figure 3.5). If this is achieved and the usage remains more or less stable, no further operations are likely to be needed. When it comes to increasing the resources, it may not be as straightforward considering that if the usage is in a bottleneck or close to it, it is uncertain how many resources the application actually needs. For this scenario there is only one policy implemented for scaling up, increasing the allocated resources by a fixed amount, and it is left to the user to configure such amount (e.g., see label 'amount' in Listing 3.3). Both configuration options of changing the resource amount when scaling up and the policy used when scaling down can be used to modulate how benevolent the platform is with the applications in terms of vertical scaling.

Finally, it can be noted that the time it took to trigger the scaling-up operation (50 seconds, from second 220 to 270) is significantly lower than the one needed to trigger a scaling-down (150 seconds, from second 290 to 440). This behavior corresponds to the responsiveness previously described, choosing in this case a more friendly configuration that acts sooner when a bottleneck is detected (3 times faster) and then gives the structure more time before reclaiming resources.

3.4.2. Subsequent resource analysis

Thanks to persisting the structure and limits documents from the *State Database* to the *Time series Database* (performed by the *Database Snapshoter* service), it is possible to carry out different kinds of analyzes for any past time window. However, considering that the continuous analysis may change both kinds of documents in a short time span, the snapshots have to be performed with a frequency equal to

or higher than the time windows used by the active services. Fortunately, this in turn also means that any a posteriori resource analysis can be performed from any moment in the past to close to present, even when the application is still running.

Although there are many ways to analyze a time series, we introduce some terminology and concepts that can be used to evaluate the differences between our serverless platform and more traditional IaaS/PaaS models with instances that have reserved resources. Using the resource time series shown in Figure 3.5, Figure 3.6 defines three areas: 1) the area under the time series itself, representing the amount of resources currently in use (i.e., the used area); 2) the area under the allocated resources (the allocated area); and 3) the area under the maximum allocatable resources (the reserved area). The allocated area is only applicable to the serverless scenario as it represents the amount of resources that are available to the instance at any precise moment, as enforced by cgroups. The reserved area is applicable both for serverless and Cloud-like instances. On a serverless scenario, this area represents the maximum allocatable amount of resources (i.e., allocated when a resource is fully used). On a Cloud scenario, it represents the amount of resources initially assigned to such instance.

For each area, an integral can be performed to add up the total amount of resources, thus excluding the time variable. Furthermore, thanks to the discrete nature of these time series, it is possible to perform the total integral as the sum of the partial integrals for each subinterval using the trapezoidal rule, having a perfect fit and thus no error in the end result. With these integrals, we can in turn define the resource utilization ratio as the quotient between the resources used and the

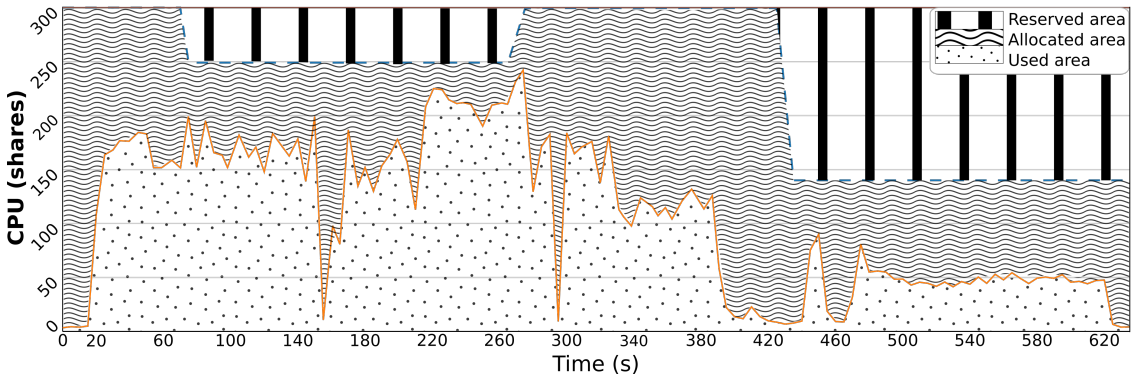


Figure 3.6: Subsequent resource analysis marking the different areas for accounting

resources allocated/reserved in the serverless/Cloud scenario. This ratio is useful to measure how many of the resources given to an application are actually used in the end, and is employed for the experiments of Section 3.5.4 to compare different configurations and scenarios.

3.4.3. Platform tweaking and configuration

Configuration parameters are crucial if there is a need to adapt the platform to a specific workload pattern that proves to be unpredictable or too resource demanding. As previously mentioned, our platform configuration can be tuned to modulate either its 'benevolence' or 'responsiveness'.

Regarding benevolence, the parameters seek to configure how the platform adapts itself to the application in terms of the resource amounts applied to scaling operations. This configuration modulates the platform's response in the vertical scaling. To configure benevolence, parameters in the structure and rule documents can be modified. For the container structure, there is a value for the maximum and minimum amount for each resource (e.g., CPU). The maximum value is used to set a limit on the amount of resources potentially available, thus making the platform equivalent to running a container on an IaaS/PaaS if the resources are fully used. The minimum value represents the amount that is ensured to be allocated. This is a crucial parameter because while a high value may result in unused resources when the application is idle, a very low value may cause the application to fail due to simply not having enough resources or even cause the container to freeze. Additionally, container structures also have a boundary parameter which specifies the amount to be maintained between the lower-to-upper and upper-to-allocated areas (see Figure 3.5). This parameter is the one that most contributes to the benevolence of the platform as it ultimately determines how wide is the range where resource usage is considered as normal (see 'normal scenario' in the figure), and thus how much the platform tries to adjust the allocated amount to the real usage. While both the maximum and minimum parameters are expected to remain fixed, the boundary can be changed if needed, as shown in the experiments of Section 3.5. When it comes to rules, specifically those that generate requests, they can be modified to change the amount of resources to be increased when scaling up, or the policy used when

scaling down.

Regarding responsiveness, it seeks to modulate the platform's response in terms of the time it takes to adapt the allocated resources to the real usage. This configuration option is based on the concept of time window, defined as the time duration used to isolate services or measure intervals. To configure responsiveness, parameters in the service and rule documents can be modified. In service documents, the time window duration for each service is specified in seconds. Rule documents can be modified to specify the number and type of events needed to trigger each kind of request. As mentioned in Section 3.4.1, responsiveness allows to mitigate hysteresis, as the rules can be configured to trigger only after a certain event is consistently detected, and the time to respond can be modulated by specifying the number of events needed for a request to be triggered. All the parameters involved in the benevolence and responsiveness aspects are summarized in Table 3.1.

For responsiveness, it is important to define an appropriate time window configuration across all the services. Figure 3.7 describes in a timeline the chain of operations carried out from the moment an event is detected to the moment when the allocated resources are finally scaled and the new information fed back into the platform. Before any analysis can take place, the resource usage time series must be generated by BDWatchdog and properly persisted. To allow for this time margin and ensure that these data are available, the *Guardian* is configured with a window delay parameter (see Table 3.1), needed to set back the time window used to retrieve the time series. In addition, the duration of the analysis window that the *Guardian* works with is configured via the window duration parameter. In the example shown in Figure 3.7, which considers both a window delay and a window duration of 10 seconds, the *Guardian* performs from second 20 to 30 the two-step analysis (i.e., event detection and request generation) described in Section 3.4.1 using the resource usage metrics previously persisted by BDWatchdog from second 0 to 10. This 10-second analysis is repeated a total of three times in a pipelined fashion (see second 20 to 50 for the *Guardian* and 0 to 30 for BDWatchdog), and after enough events are detected, a scaling-up request is generated by the *Guardian* around second 50. This request is in turn retrieved by the *Scaler* in its next polling (second 50 to 55). Although such polling frequency can also be configured, it should always have a lower value than the *Guardian* window duration in order to process requests as soon as

Table 3.1: Parameters to modulate benevolence and responsiveness

	Benevolence			**Responsiveness**	
Dimension	Vertical (resource amount)			Horizontal (time to respond)	
Document	Structures	Limits	Rules	Services	Rules
Parameters	Maximum amount Minimum amount	Boundary	Amount (scaling up) Policy (scaling down)	Window delay (*Guardian*) Window duration (*Guardian*) Polling frequency (*Scaler*)	#Events for scaling up #Events for scaling down

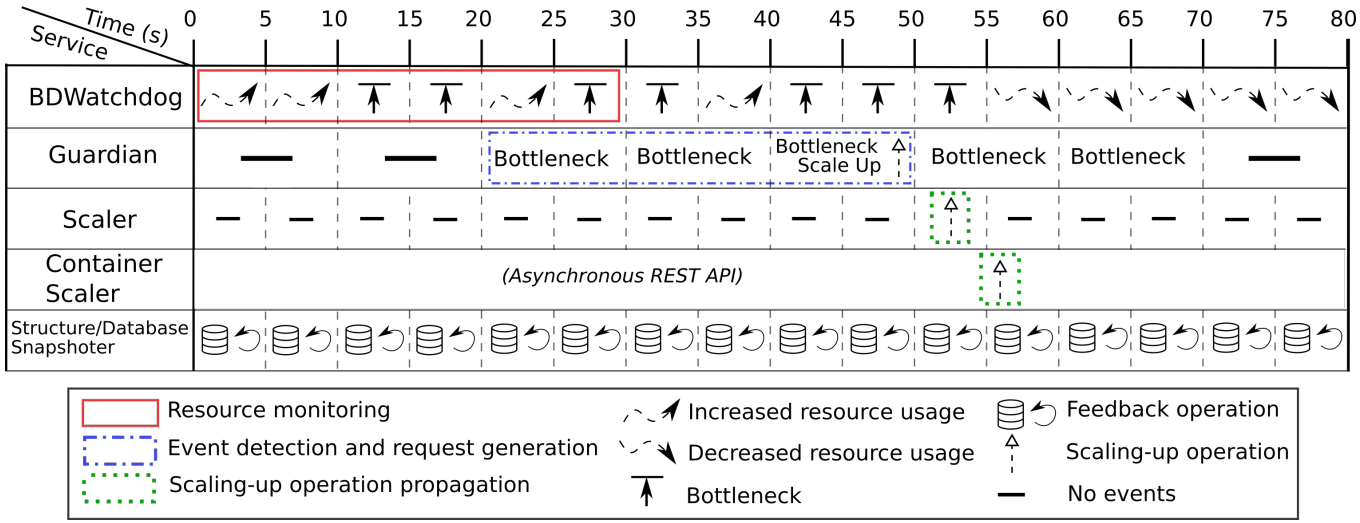


Figure 3.7: Time window analysis (window delay and duration: 10 seconds, polling frequency: 5 seconds)

possible. Once the request is processed, it is sent to the corresponding *Container Scaler* (second 55). As this service is a REST API waiting for requests, no delay is expected. Finally, after the scaling is properly performed, the snapshoter services feed back the changes into the system in their next polling (second 60 to 65).

3.5. Experimental results

The main objective of our scaling platform is to adjust the resources given to Big Data applications to their real usage while also minimizing the overhead imposed and maintaining scalability overall. On the one hand, to measure the fitness and gains provided by the platform, as well as to study the overhead, both batch and streaming workloads are deployed on a first testbed cluster using 12 containers. On the other hand, to analyze the scalability of the platform as well as its flexibility in comparison with other commercial services, the same workloads are deployed on a larger second testbed using 32 containers.

The experimental and platform configurations for both testbeds are detailed in Sections 3.5.1 and 3.5.2, respectively. Sections 3.5.3 and 3.5.4 present real-time scaling results and a subsequent resource analysis of the workloads, respectively, for the first testbed, while Section 3.5.5 presents scalability results for the experiments conducted on the second testbed.

3.5.1. Experimental configuration

The experiments are carried out on container clusters based on LXC with LXD 3.6 as container manager. Each LXC container runs Ubuntu 16.04. The first testbed consists of a 12-container cluster deployed on the Big Data infrastructure at the CESGA Supercomputing Center [29]. Each container is configured with a maximum of 2 cores (i.e., 200 CPU shares) and 10 GiB of memory (24 cores and 2,400 CPU shares in total). The second testbed consists of a 32-container cluster deployed on the Grid'5000 infrastructure [54]. In this case, each container provides up to 6 cores and 45 GiB of memory (192 cores and 19,200 CPU shares in total). In both testbeds the containers have a dedicated disk mounted from the physical host. Resources are

guaranteed to be reserved across containers at all times, even when they are all scaled up to the maximum.

Regarding the workloads, TeraSort and PageRank are selected as representative batch workloads, while FixWindow is deployed as an example of a streaming application (see Table 3.2 for their specific configuration for each testbed). In order to present results using different Big Data processing frameworks, the first testbed uses Hadoop 2.9.0 [101] to execute the TeraSort workload, and Spark 2.3.0 [114] for PageRank and FixWindow. The second testbed uses Spark for all workloads. These workloads are selected because they exhibit very different resource usage patterns, particularly for the CPU. On the one hand, TeraSort is an I/O-bound workload that shows a CPU usage with low variation over time. PageRank, on the other hand, is an iterative and CPU-bound workload showing a high degree of variability in CPU usage. This different behavior is to be taken into account when scaling policies are applied in order to minimize the execution time overhead. Regarding FixWindow, its streaming nature requires to process a constant input of data, which implies a very low variation of resource usage when compared to batch applications. Unlike batch workloads, the stream processed by FixWindow is endless as long as the processing remains stable (i.e., processing times are lower than the incoming data interval span). For this reason, the execution of FixWindow is restricted to 15 minutes in the experiments. Specifically, in order to simulate a variation in the resources needed to process the stream, the 15-minute execution is divided in 3 consecutive 5-minute stages with different stream sizes (see Table 3.2).

To deploy Hadoop and Spark, the Big Data Evaluator (BDEv) tool [107] is used. For the FixWindow streaming workload, HiBench 7.0 [61] is also required to produce the data source to be processed. In addition, a Kafka 2.1.0 broker [50] is needed to receive the data from the HiBench producers and temporarily store it before being offered to the FixWindow consumers. For these streaming experiments, the containers have to be split accordingly to serve the roles of producers, brokers and consumers. The first testbed uses a configuration of 1+2+9 (12) containers for Kafka, HiBench and FixWindow, respectively, while the second testbed uses 4+8+20 (32) containers. In these experiments the broker and producer containers are ignored as their resource usage patterns are not interesting.

Table 3.2: Workload configuration

		Testbed 1	Testbed 2
Batch workloads			
TeraSort	Dataset size	50 GiB	300 GiB (hybrid 1st use) 200 GiB (concurrent) 130 GiB (hybrid 2nd use)
PageRank	Number of pages	5 million	60 million
	Number of iterations	5	2
Streaming workloads			
FixWindow	Stream data size	stage 1: 66 MiB stage 2: 167 MiB stage 3: 28 MiB	stage 1: 3.72 GiB stage 2: 5.58 GiB stage 3: 1.49 GiB
	Stream window size	5 seconds	10 seconds

3.5.2. Platform configuration

Considering the different objectives and characteristics of both testbeds, two platform configurations are used for the experimentation. The objective of the first testbed is to present a study of the resources according to different configuration scenarios. For this matter, Table 3.3 presents two serverless scenarios in terms of the level of benevolence and responsiveness used by the platform, configured using the rule documents. Although close in terms of responsiveness as the number of events to trigger the rules are similar for both scenarios, the difference regarding benevolence is to be noted. On the one hand, for TeraSort and FixWindow, which show a more constant resource usage, the amounts to be scaled when a bottleneck is present are 75 CPU shares and 2 GiB of memory. On the other hand, for PageRank, which shows a more variable resource usage, the values are 100 CPU shares (i.e., one core) and 3 GiB of memory. Table 3.4 presents different configurations for the boundary parameter that are set up by using the limits documents. In the same way as before, two serverless scenarios are laid out. For the first scenario, used for TeraSort and FixWindow, a total of four configurations are defined (e.g., *cpu_MEM* means a low boundary value for CPU and a high value for memory). For the second

Table 3.3: Configuration of the scaling platform (Testbed 1)

	Rule configuration			
	CPU underuse	CPU bottleneck	Memory underuse	Memory bottleneck
TeraSort FixWindow	#events up = 0 #events down >= 8	#events up >= 2 #events down <= 2 amount: 75 shares	#events up = 0 #events down >= 8	#events up >= 2 #events down <= 2 amount: 2 GiB
PageRank	#events up = 0 #events down >= 6	#events up >= 2 #events down <= 3 amount: 100 shares	#events up = 0 #events down >= 5	#events up >= 2 #events down <= 2 amount: 3 GiB

Table 3.4: Boundary values for different serverless configurations (Testbed 1)

	TeraSort & FixWindow				**PageRank**
	cpu_mem	*CPU_mem*	*cpu_MEM*	*CPU_MEM*	*CPU_MEM'*
CPU (shares)	20	40	20	40	50
Memory (GiB)	2	2	3	3	3

Table 3.5: Configuration of the scaling platform (Testbed 2)

Rule configuration		
CPU underuse	CPU bottleneck	CPU boundary
#events up = 0 #events down = 5	#events up = 2 #events down = 3 amount: 225 shares	150 shares

scenario, used for PageRank, only one configuration is used. Furthermore, a baseline configuration is used for all the workloads, which runs them without any scaling, thus representing any Cloud-like instance with reserved resources.

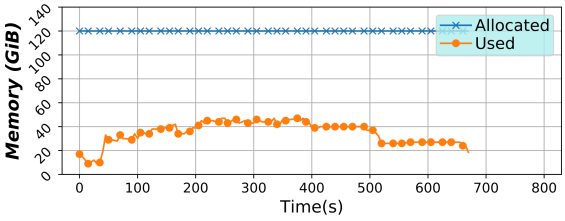
The second testbed aims to show the scalability as well as some specific flexibility features of the platform, and because of this there is just one platform configuration scenario common to all the workloads (detailed in Table 3.5). In addition, only the CPU is studied in this case. This configuration is considered to be in a middle point in terms of both benevolence and responsiveness.

Finally, the remaining configuration parameters, mainly those to tune the services, are the same across both testbeds: a polling frequency of 5 seconds for the *Scaler* and the *Structure/Database Snapshoters*, and a 10-second window delay and window duration for the *Guardian*.

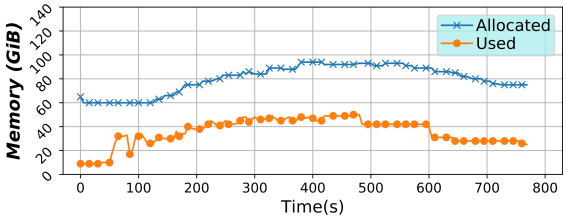
3.5.3. Real-time scaling results

The experiments presented in this section show the resource time series of workload executions for both the baseline and the serverless configurations, where dynamic resource scaling is applied by the platform in real time. The first testbed is used to execute them.

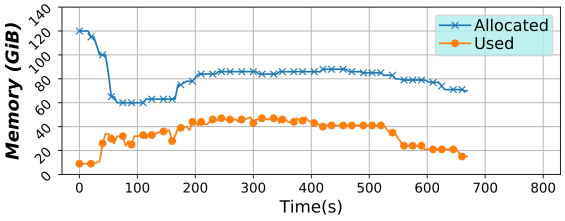
Figure 3.8 shows the aggregated, application-level memory usage for TeraSort. From top to bottom, the plots represent the baseline configuration (Figure 3.8a) and the four serverless configurations described in Table 3.4. Due to the high-level nature of the plots, which aggregate the memory usage of all containers, upper and lower limits cannot be shown as these only apply to individual containers. It is worth noting that although the allocated amount of memory varies between configurations, the used amount follows mostly the same pattern, showing that both metrics are actually independent. Regarding the allocated amount, it remains constant in the baseline, while the boundary parameter causes it to be closer to or farther from the used amount in the serverless configurations. The middle plots (Figures 3.8b and 3.8c) represent the configurations where the memory boundary per container is set to 2 GiB (see Table 3.4), thus having a lower 'benevolence', while the bottom ones (Figures 3.8d and 3.8e) have a boundary of 3 GiB.



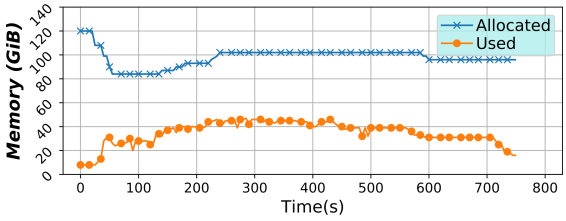
(a) Baseline



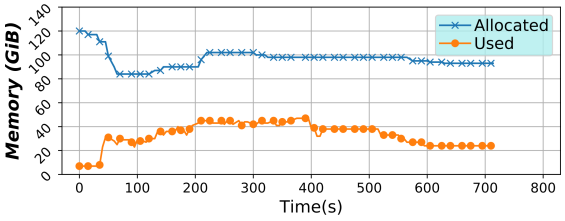
(b) cpu_mem



(c) CPU_mem

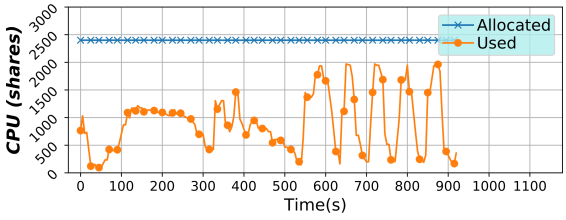


(d) cpu_MEM

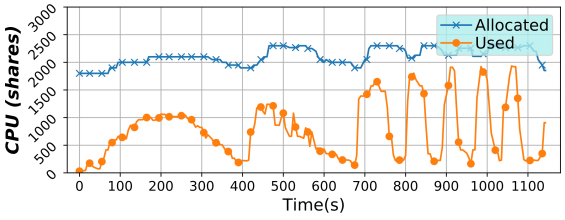


(e) CPU_MEM

Figure 3.8: Aggregated memory usage of TeraSort



(a) Baseline



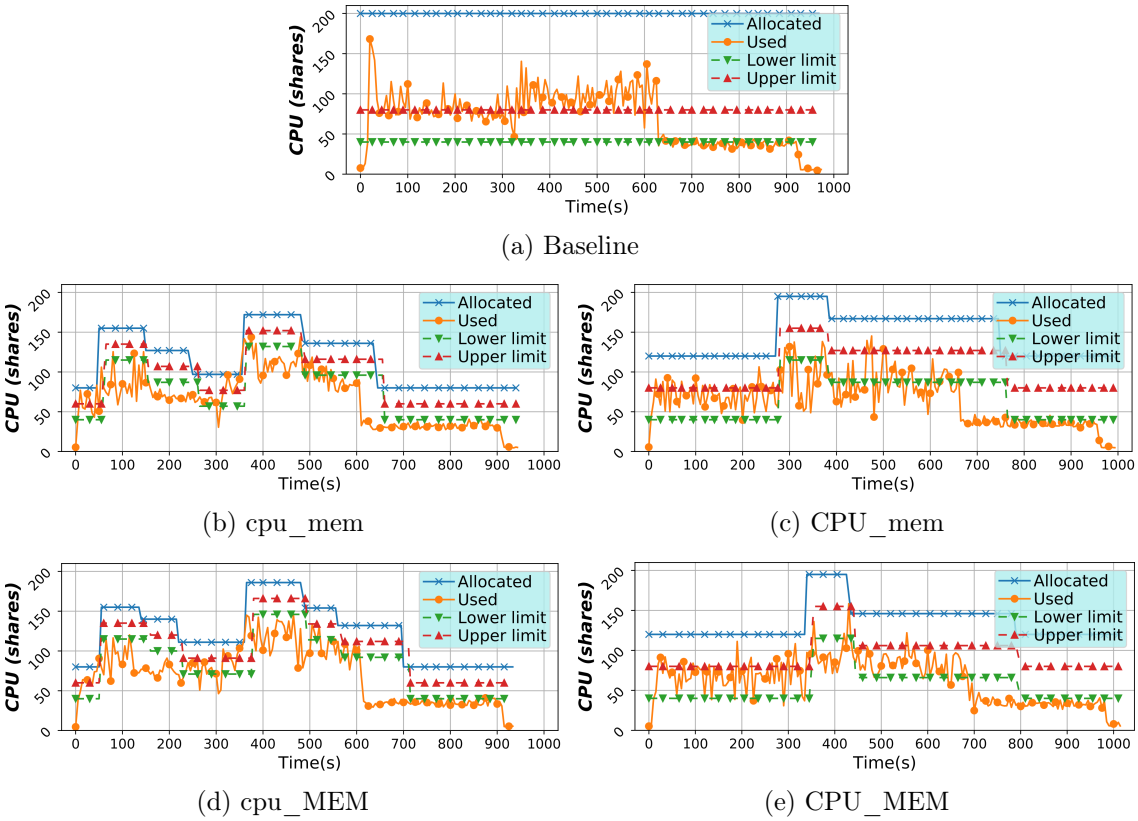
(b) CPU_MEM'

Figure 3.9: Aggregated CPU usage of PageRank

In Figure 3.9, the aggregated, application-level CPU usage is presented for PageRank. For this case only one serverless configuration is tested (right plot) along with the baseline (left plot). It can be observed that the pattern for the allocated amount roughly mimics the pattern of the used one, albeit with a slight delay. More precisely, this delay is the side effect to be tackled and modulated via the 'responsiveness' configuration discussed in Section 3.4.3. In addition, it is worth

noting how the CPU usage peaks, which correspond to iterative phases of the algorithm, have different values for both configurations. In the baseline, such peaks consistently reach around 2,000 shares (i.e., 20 cores). In the serverless configuration, they range from 1,600 to 2,000 shares. So, our platform adapts to the iterative nature of PageRank, as CPU peaks increase with each iteration.

Finally, Figure 3.10 presents the CPU usage of FixWindow for an individual container following the same configuration order as in Figure 3.8. Unlike previous application-level plots, it is now possible to show the upper and lower limits as they are only present when analyzing container resource usages. Moreover, these container-level plots allow to analyze the scaling operations that took place, why they were triggered, and how it is possible that the allocated resources may closely fit the used ones. For CPU, this is particularly the case with the *cpu_mem* and *cpu_MEM* configurations, as they set a lower boundary value. Regarding the plat-



(a) Baseline

(b) cpu_mem

(c) CPU_mem

(d) cpu_MEM

(e) CPU_MEM

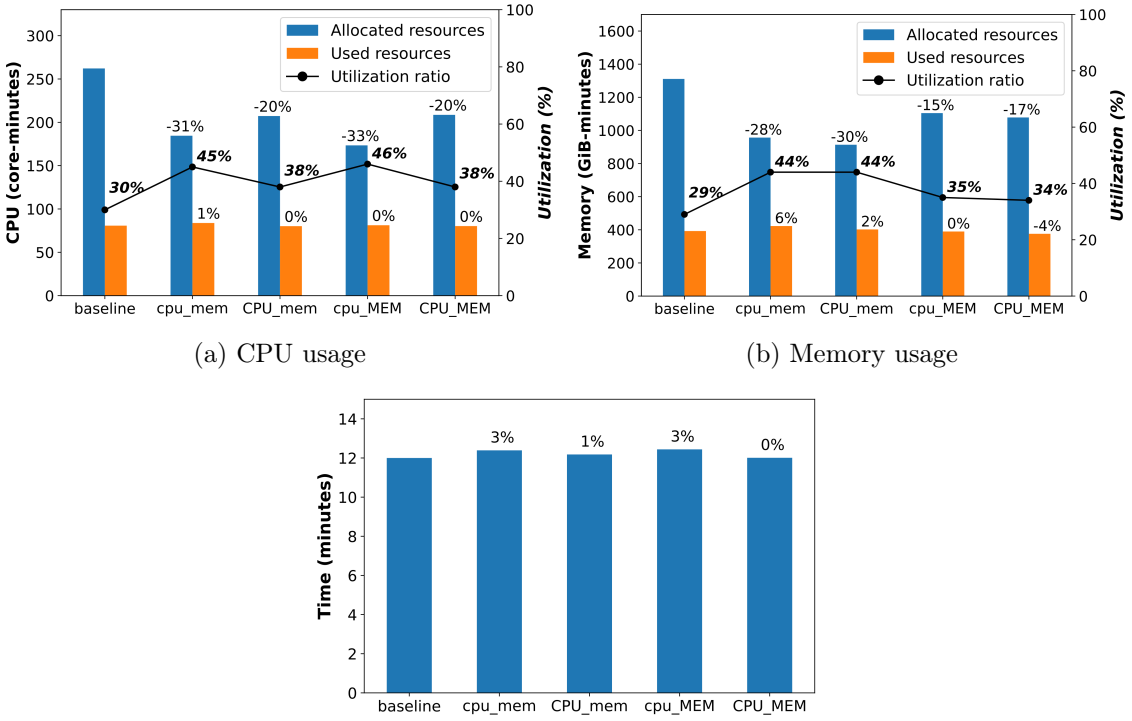
Figure 3.10: CPU usage and scaling of FixWindow

form behavior to adapt to the stream and considering the three 5-minute stages previously described in Table 3.2, it is easily identifiable that the four serverless configurations must perform a scaling-up operation around second 300, as the second stage processes a larger stream. Later, around second 600, a scaling-down operation is performed as a low amount of CPU is required, considering that the third stage starts with the smallest stream (see Table 3.2). In addition, it is interesting to note how the left-hand plots (Figures 3.10b and 3.10d), which have a lower CPU boundary, perform initial scaling-up operations around second 50. This is due to a cold-start effect, as all the executions begin with the lowest possible allocated amount, considering that the minimum amount is set to 40 CPU shares for all configurations. The fact that the right-hand plots (Figures 3.10c and 3.10e) do not show this behavior is because the allocated CPU amount was already large enough thanks to the higher boundary (see the effect of the boundary on the allocated amount in Figure 3.5).

3.5.4. Subsequent analysis results

The subsequent analysis presents the aggregated amounts for used and allocated resources (i.e., the area values shown in Figure 3.6), as well as the resource utilization ratio as previously defined in Section 3.4.2. These results are presented for each workload using the baseline and the serverless configurations. The execution times of the workloads are also included to account for the platform overhead. We show the average value of a minimum of 10 executions for each workload and configuration.

For the TeraSort workload in Figure 3.11, the CPU plot (see Figure 3.11a) shows how the used amount is constant across all the executions, whether any scaling is applied or not. However, the allocated amount varies from the baseline to each configuration. Taking into account the boundary parameter, for *cpu_mem* and *cpu_MEM* configurations, which have a lower CPU value, the allocated amount is reduced by around 32%, whereas for *CPU_mem* and *CPU_MEM*, where such value is higher, the allocated amount is reduced by 20%. The CPU utilization ratio increases from 30% in the baseline to 46% and 38% for the lower and higher boundary configurations, respectively (which means an improvement of 53% and 27%). When it comes to memory (see Figure 3.11b), the results are analogous.



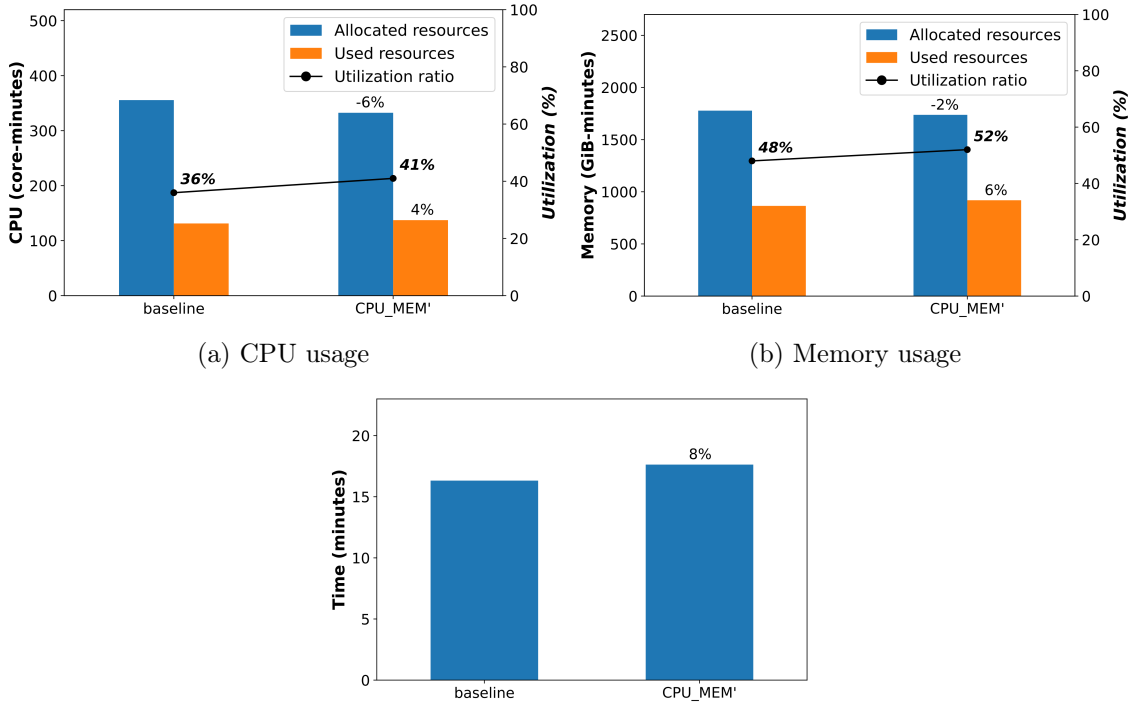
(a) CPU usage

(b) Memory usage

(c) Execution time

Figure 3.11: Resource usage and execution time of TeraSort

The used amount varies across the different configurations with a higher amount for *cpu_mem* and a lower amount for *CPU_MEM*. Nevertheless, this difference can be ignored as memory amount varies slightly even for different executions with the same configuration due to the activity of the garbage collector in these Java-based workloads. For the lower memory boundary configurations (*cpu_mem* and *CPU_mem*), the amount of allocated memory is around 29% lower than the baseline, whereas for the higher memory configurations (*cpu_MEM* and *CPU_MEM*) it is around 16%. In turn, the memory utilization ratio increases from 29% of the baseline to around 44% and 34% for the lower and higher boundary configurations, respectively (which means an improvement of 52% and 17%). Figure 3.11c shows, for each configuration, the execution time and the corresponding overhead compared to the baseline. It is interesting to note how the overhead changes slightly according to the CPU boundary configuration while for memory it seems to have no effect. For the lower CPU boundaries an overhead of only 3% is imposed, being negligible for configurations with high boundaries.



(a) CPU usage

(b) Memory usage

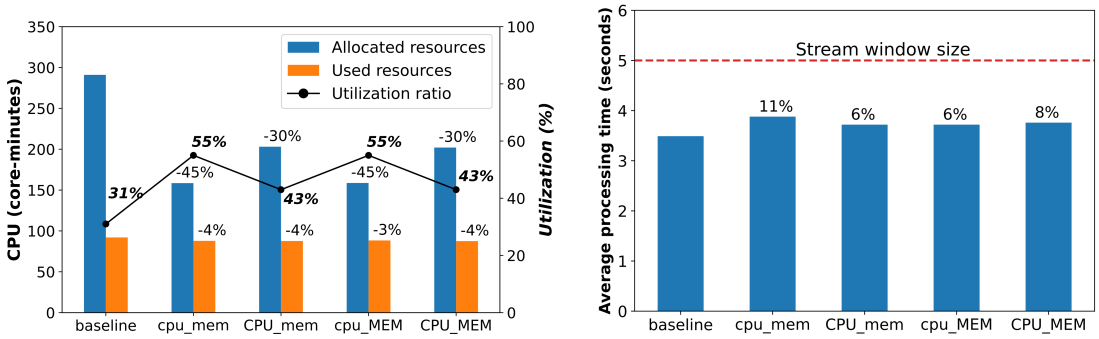
(c) Execution time

Figure 3.12: Resource usage and execution time of PageRank

Figure 3.12 shows the results for PageRank. Both CPU and memory (see Figures 3.12a and 3.12b) present similar used values with only a 4% and a 6% increase, respectively, and a slightly lower allocated amount (6% and 2%). The utilization ratio raises slightly from 36% to 41% for CPU and from 48% to 52% for memory. In this workload, the platform has more difficulties to achieve better results due to the previously commented oscillating behavior of the iterative phases of PageRank, having overall an execution time overhead of 8% (see Figure 3.12c). As with TeraSort, PageRank also shows to be more sensitive to CPU than to memory scaling, which requires to have a specific scenario with a more benevolent and responsive configuration (see Tables 3.3 and 3.4). Nevertheless, these experiments prove that in a worst-case scenario with an application that shows a highly variable resource usage pattern, our platform does not have a big impact on the workload and merely steers the serverless environment towards a traditional Cloud instance with reserved resources. If needed, the platform can also be dynamically configured to further minimize this overhead by increasing benevolence even more, as the rule configura-

tion can also be changed in real time to increase the amount of resources given to the application when scaling up.

Finally, the results for FixWindow are shown in Figure 3.13. The CPU usage (see Figure 3.13a) follows a similar behavior as TeraSort. While the used amount is almost equal for all the configurations (around 4% lower than the baseline), the allocated amount is greatly reduced. According to the boundary, when a lower CPU value is used (*cpu_mem* and *cpu_MEM* configurations) the allocated CPU is reduced by 45%, while with a higher boundary it is reduced by 30%. As a consequence, the utilization ratio is greatly increased from 31% of the baseline to 55% and 43% for the lower and higher boundary configurations, respectively (which means an improvement of 77% and 39%). The execution time overhead is more or less constant for all configurations, ranging from 6% to 11% (see Figure 3.13b). It is important to note that this execution time corresponds to the average processing time for a window and not to the workload duration, as the stream is endless. Thus, the stream processing does not suffer any penalty as long as the average processing time remains lower than the stream window size (see this parameter in Table 3.2 and Figure 3.13b). Overall, FixWindow represents a best-case scenario for our platform as, unlike PageRank, the resource usage patterns are more stable.



(a) CPU usage

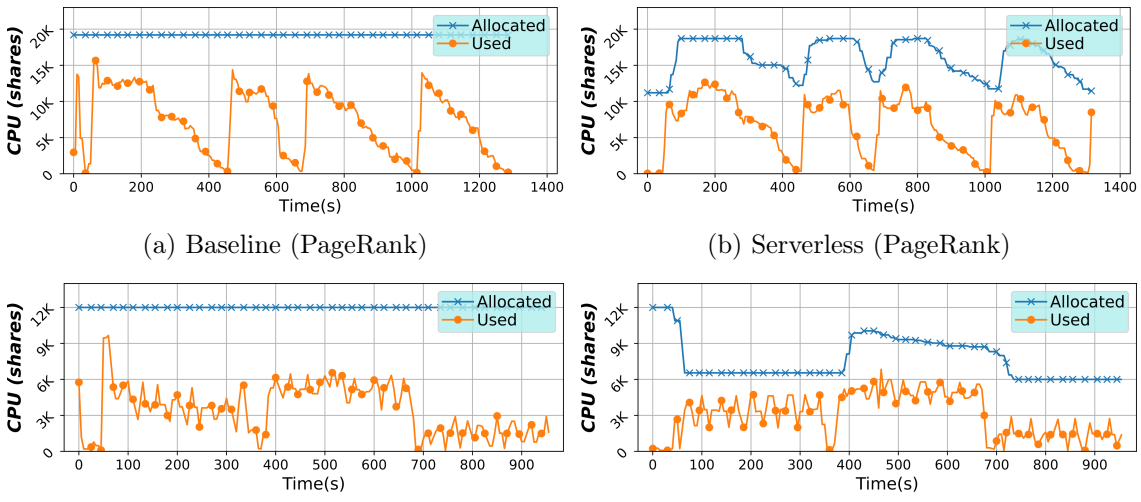
(b) Average processing time

Figure 3.13: CPU usage and average processing time of FixWindow

3.5.5. Scalability and hybrid experiments

Considering that this platform aims to support Big Data workloads or, overall, any application that requires a lot of containers to be deployed, it is important to test the scalability of the platform on a realistic scenario. In addition to scalability experiments based on increasing both the size of the cluster and the datasets, other experiments referred to as 'hybrid' are also presented. Hybrid experiments try to show the flexibility of the platform when it comes to dealing with unexpected resource patterns that arise when multiple workloads are executed, even in a concurrent way, as well as with the use case of the need to change the resource limits within the execution. All the experiments presented in this section use the second testbed described in Section 3.5.1, which increases the number of cores from 24 to 192 (8x). Furthermore, the size of the datasets also increases (e.g., 12x for PageRank, see Table 3.2). For each workload and experiment, the baseline and the serverless scenario with the configuration of Table 3.5 are presented.

Regarding scalability experiments, Figure 3.14 shows the aggregated, application-level CPU usage for PageRank (top graphs) and FixWindow (bottom graphs) using the 32-container cluster. It can be seen how on the serverless scenario the limit is scaled accordingly and follows closely the usage pattern. For these experiments the CPU utilization ratio increases from 37% to 41% (11% improvement) for PageRank



(a) Baseline (PageRank)

(b) Serverless (PageRank)

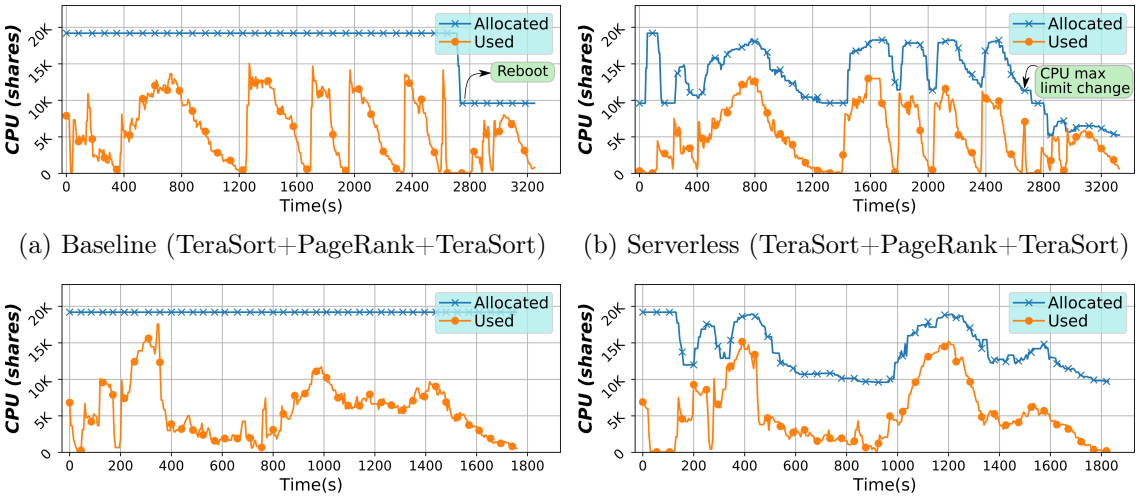
(c) Baseline (FixWindow)

(d) Serverless (FixWindow)

Figure 3.14: Aggregated CPU usage of PageRank (top) and FixWindow (bottom)

and from 28% to 39% (39% improvement) for FixWindow. The overhead in execution time is around 2% for PageRank, and 25% for the window average processing time of FixWindow. However, in this latter case the average processing time still keeps below the stream window size of 10 seconds, and thus it does not affect the stream processing overall.

Figure 3.15 presents the execution of two hybrid workloads, where TeraSort and PageRank are executed using different combinations. The workload of Figures 3.15a and 3.15b consists of executing TeraSort (300 GiB), followed by PageRank (60 million pages), and finally another TeraSort (130 GiB). Note that this workload uses two different CPU limits: a maximum of 19,200 shares (i.e., 32 containers with 6 cores each) for the first TeraSort and PageRank, while such limit is halved to 9,600 shares for the second TeraSort at approximately second 2,700. This change of resources is done dynamically, without any reboot, and with no inherent cost for our platform (see Figure 3.15b), only requiring a specific scaling operation and some minor changes in the structure documents to enforce the new limit. If compared with other solutions such as traditional Cloud instances (see Figure 3.15a) a reboot would be needed to change the instance resources, thus having an undetermined delay. For this workload the overhead imposed by our platform is only around 5%, while the CPU utilization increases from 32% to 38% (19% improvement). The hybrid workload of



(a) Baseline (TeraSort+PageRank+TeraSort)

(b) Serverless (TeraSort+PageRank+TeraSort)

(c) Baseline (two concurrent TeraSort)

(d) Serverless (two concurrent TeraSort)

Figure 3.15: Aggregated CPU usage of the hybrid workloads

Figures 3.15c and 3.15d consists of executing two TeraSort (200 GiB) concurrently with a 3-minute delay between both. This workload differs from the previous one in the fact that concurrency causes it to be significantly less deterministic, which serves to show how the platform adapts in real time to the unpredictable behavior of the underlying infrastructure. In this case, there is an execution overhead of only 4% (ignoring the reboot delay of the baseline) with a CPU utilization increase from 29% to 37% (28% improvement).

3.6. Conclusions

New paradigms for application execution are constantly appearing and evolving, always looking for more flexible and cost-effective ways of deploying and executing workloads. One of such novel paradigms is serverless computing, which evolved from the Cloud adoption in recent years. With this paradigm, the user no longer needs to provide resource specifications to run an application, which is billed only for the actual resources that were used rather than for the allocated ones. Unfortunately, the current most commonly used platforms and services for serverless computing enforce strict rules to execute applications or only support them as single programs or simple scripts.

This chapter has introduced a rescaling platform that implements an active management of system resources for container-based clusters, supporting by design different tuning policies. Choosing the serverless paradigm as the implemented policy, the platform provides users with a fully working virtual environment that is still really close to the more flexible, traditional virtual machine instances while relying on containers and providing serverless features such as a high resource utilization ratio. In order to apply the serverless paradigm to such containers, the platform dynamically and automatically scales their available resources according to their real usage, without any need for user intervention. Furthermore, resources are rescaled without having to reboot the containers or interrupt the application in any way due to the ability to limit resources with cgroups.

This platform has been experimentally assessed on two different infrastructures by running several Big Data workloads in order to demonstrate: 1) how they adapt

to a serverless environment, where the available resources dynamically change according to their real usage; and 2) the scalability and flexibility of the platform. Regarding the first point, the results have been presented in two ways: the real-time resource scaling behavior and the subsequent accounting of the resource usages. The real-time scaling has shown how the platform can be adapted to any application by using an appropriate configuration, although the degree of adaptation varies according to the resource usage pattern of each application. The subsequent resource accounting has proved why a serverless platform does not impose any penalty as long as it responds quickly enough to the application's needs. What is most important, this accounting also showed how the amount of used resources does not vary from experiments executed with a constant or reserved amount of resources (e.g., a virtual machine) to those with resource limits changed dynamically as in our platform. In turn, this implies that the resource utilization ratio can be improved from a baseline scenario of a non-serverless environment to one where the used and allocated resources are closer, while maintaining containers as the virtualization technology choice. This scenario is of great interest for those workloads with a stable resource usage as in the case of the FixWindow streaming application, where CPU utilization is improved by up to 77% with a window processing time overhead of only 6%. For a batch application like TeraSort, which presents a more variable use of resources than FixWindow, such improvement is up to around 53% for both CPU and memory utilization, with a negligible execution time overhead. Finally, regarding the scalability of the platform, it has been shown both from a design and theoretical point of view and experimentally how it can scale to encompass Big Data workloads without appreciating any impact on both the imposed overhead and the resource utilization ratio.

The framework that deploys the presented platform is available to download at `http://bdwatchdog.dec.udc.es/serverless`.

Chapter 4

Power budgeting and energy capping in container clusters

Considering that energy efficiency is currently one of the most studied research topics across computer systems for its importance as part of the movement towards a more environmentally sustainable future, the energy consumed by applications is becoming of prime importance for system administrators and service providers. However, most of the published literature focuses on the energy consumed as a whole by entire systems [113], individual physical hosts [86, 99] or, at the most, by virtual machines [68], as these are the most common means of virtualization. This chapter presents a novel scenario that helps move the research further when it comes to energy monitoring and management in large infrastructures [1, 39] such as Big Data clusters and, more specifically, in container clusters.

So far, Chapters 2 and 3 have described in detail two frameworks to implement fine-grain resource monitoring and a resource scaling platform for container-based virtual environments, respectively. More specifically, Chapter 3 focused on presenting the overall architecture of the rescaling platform and describing a resource tuning policy based on the serverless paradigm. However, it is also possible to support other interesting policies by extending the framework and/or integrating external tools. In this chapter, we present and analyze a scenario where the objective is to consider energy as a first-class accountable resource in the same way as CPU and memory. To fulfill this objective, a new tuning policy has been implemented for the rescaling

platform in order to manage energy as any other resource, and thus place energy caps and create power budgets that can be shared among applications or users across container clusters. In addition, both functionalities can be managed and changed in a dynamic way and in real time. This scenario is created by integrating and adapting the two frameworks described in previous chapters with a third-party energy measuring tool that provides fine-grain, software-based energy monitoring for containers.

The remainder of the chapter is organized as follows: Section 4.1 presents the underlying core technologies and concepts. Section 4.2 summarizes the related work. Section 4.3 describes how the scenario has been implemented to provide an energy-controlled environment relying on containers. Finally, experimental results are presented in Section 4.4 and concluding remarks are summarized in Section 4.5.

4.1. Background

This section first describes the category of energy study performed in this chapter, and then presents some existing technologies and the motivation behind their usage. It also defines other concepts that are later used throughout the chapter.

In our proposal, energy monitoring focuses on CPU exclusively and, more precisely, on individual containers. This decision is not only backed by technical limitations, but also by design choices. First, considering that we rely on software-level energy metrics, energy usages that are out of the actual server chassis (e.g., cooling, rack) are discarded to avoid any physical hardware deployment. Second, any energy study that uses software-level monitoring exclusively, particularly for virtualization technologies like containers [12, 23], is limited to CPU and memory energy metrics as these are typically provided by RAPL directly or inferred via lower-level processor information (e.g., the Linux *perf* tool). This fact leaves out devices such as disks and network cards. Finally, from the CPU and memory energy metrics that are available, only CPU is used as per design our platform scales CPU resources according to CPU usage, without taking into account memory. Nonetheless, note that the CPU accounts for the largest percentage of energy consumed for most applications, followed afterwards by memory and then, with a big difference, by disk

and network. This is further assessed by studies like [39], which shows that CPU cores have significantly higher energy usage than other devices under isolated high utilization, something that is increasingly clear as the number of cores rises.

The motivation behind using containers has already been laid out in the previous chapter, putting much of the focus on their feature of real-time changeable resource limits thanks to the cgroups file system. Our platform relies on this feature to achieve the objective of our energy scenario, that is, being able to set an energy limit on a user or an application by using a form of CPU throttling to control the energy consumed by the containers at any time. Such throttling is implemented via a precise, fine-grain control of the CPU shares that containers have at any moment, taking advantage of the ability to do so with cgroups. However, some limitations have to be taken into account when deploying the user's applications using containers: 1) energy can only be limited for those applications (or parts of) that are containerized; 2) containers have to be private and user attached, otherwise it is not possible to differentiate and account for user's energy usage; and 3) applications should ideally be agnostic of the number of cores presented to them and have some degree of flexibility in this regard, considering that the number of cores varies along time.

Finally, some concepts regarding energy require to be defined as they will be extensively used later on. First, the term *power budget* is used to express a given power limit that should not be surpassed, which can be applied to the entire platform, users and applications (as a set of containers). Second, the concept of *energy proportionality* [18, 75] is used to explain the behavior of current CPUs when it comes to the energy consumed compared to the work being performed. Such behavior can be expressed as how the relationship between both variables evolves according to the CPU load. On an ideal CPU, such relationship would be constant and linear, that is, the energy consumed by the CPU in idle state is zero, half the maximum power at half load and the maximum when usage is the highest. Unfortunately, real CPUs behave quite differently, causing an inescapable impact on the experiments, as further discussed in Section 4.4. With the currently available hardware, not only the energy consumed is not zero when the CPU is idle, thus having an *idle energy consumption*, but also its evolution according to the load is logarithmic instead of linear [94]. So, we define the *energy efficiency ratio* as the amount of CPU shares

used according to the energy consumed in a given time window.

4.2. Related work

Previous works have studied different energy-related topics on Cloud environments, using virtual machines or containers. These topics include energy monitoring, management or ultimately capping on systems, from individual applications on single processors to entire workflows across data centers.

Regarding energy monitoring, a few tools can report on energy metrics from running containers, such as DEEP-mon [23] and PowerAPI [32, 46]. Both solutions rely on low-level operating system utilities (e.g., *perf*) that enable precise control over events triggered by individual processes or threads. These events, in conjunction with other system metrics, are strongly correlated and thus can be used to formulate a power estimation of a container. In addition, these solutions impose a near-negligible overhead. Related to this topic, it is interesting to point out a study of the energy consumption overhead that container-based virtualization imposes on the applications [91]. In this work, the authors run several workloads in and out of Docker containers, concluding that they pose a small overhead both in runtime and energy consumption, the latter being mostly related to the former.

There are also several solutions to manage the energy consumed using different techniques, such as scheduling policies, Dynamic Voltage and Frequency Scaling (DVFS) technologies, or those based on container/virtual machine migration. Regarding virtual machines, Paya and Marinescu [86] present several options, such as migrating and changing the state of hosts or even shutting them down, all in order to maintain a Cloud system within an area close to being energy proportional. Piraghaj et al. [88] introduce containers into their study, albeit grouped within virtual machines. The authors aim to increase the energy efficiency of a system through overall consolidation of containers on the hosts, thus avoiding scenarios of either underutilization or overloading, and migrating virtual machines, if necessary. Recent works focus entirely on groups of containers and try to increase their energy efficiency as a whole. To do so, Lin et al. [73] look for the most efficient configuration of containers, cores and hosts, as well as core frequency and voltage. This work also

proves that the energy consumption of a workload is highly dependent, among other factors, on the number of cores (i.e., CPU shares) and the overall container configuration on the entire host. Regarding this topic, it is also interesting to mention other works that study how DVFS and its scaling of the CPU frequency and voltage [63] can affect the workloads executed on containers [73], virtual machines [38] and bare hosts [78], or how it may be used for scheduling decisions [99] or for specific processing paradigms such as streaming [33].

There are other works that present frameworks and platforms with a similar objective to the scenario proposed in this chapter, which are worth describing in more detail:

- iBrownout [113] is a platform that, with the idea of a control brownout in mind, deactivates optional parts of services or applications when there is a need to lower the energy consumed according to an imposed limit, or in a high load situation. Furthermore, iBrownout is able to apply scalability policies when it comes to the number of hosts, putting idle hosts to sleep if necessary to achieve a high container consolidation overall. Similarly to our solution, iBrownout also defines a power budget, users and applications (referred to as services), which in turn are composed of microservices running on containers. Unlike our solution, iBrownout uses horizontal scaling combined with guided decision making to alter the execution of services. Nevertheless, our platform does not necessarily require an a priori knowledge about the applications executed on it and the containers are scaled dynamically in terms of CPU.

- DockerCap [12] is a software-level power capping framework for Docker containers. This framework is similar to our platform in that it can enforce an energy limit to be respected. To do so, DockerCap implements a feedback loop that relies on the cgroups file system to restrict the container CPU quota and thus its energy consumption. Nonetheless, our work goes further by considering not only individual containers, but also significantly larger sets of them across several hosts, while also targeting much more complex workloads such as Big Data applications and higher-level abstractions such as multi-container applications and users. We also present the energy overheads that inevitably arise due to the longer execution times when restrictions are applied, giving some insight into how these overheads vary according to the underlying CPUs'

energy efficiency and proportionality. This is further exposed with the larger range of power caps we use (DockerCap goes from 20W to 40W, while we go up to 1,500W).

- Nornir is described in [34, 35] as a self-adapting framework capable of enforcing energy constraints dynamically by using mechanisms such as DVFS or core reallocation. Most interestingly, this work uses a feedback loop similar to the one used for DockerCap and our platform, backing it as a good strategy for real-time energy enforcement. However, Nornir focuses on parallel applications running on shared-memory systems, leaving out any virtualization technology.

To summarize and put the proposed platform in context with all the previous related work, it has to be noted that our main objective is to define a power budget that can be enforced on users, applications and containers. To do so, vertical scaling of CPU resources is used with the decisions being taken through policies and thresholds.

4.3. Scenario design and implementation

Our main goal is to provide users with a container-based platform where energy consumption can be monitored and, to a certain limit, enforced. The overall architecture of the proposed platform is described in Section 4.3.1. The control of the energy consumed by the containers via CPU throttling is thoroughly described in Sections 4.3.2 and 4.3.3.

4.3.1. Architecture overview

Considering the container as the basic infrastructure unit that can be managed within the platform, the following entities are defined to create richer and more flexible scenarios: *containers*, *applications* and *users*.

So, the proposed platform is organized as a three-level hierarchy, as depicted in Figure 4.1, showing the aforementioned entities from bottom to top. At the lowest level, multiple containers can be grouped to form an application. At the intermediate

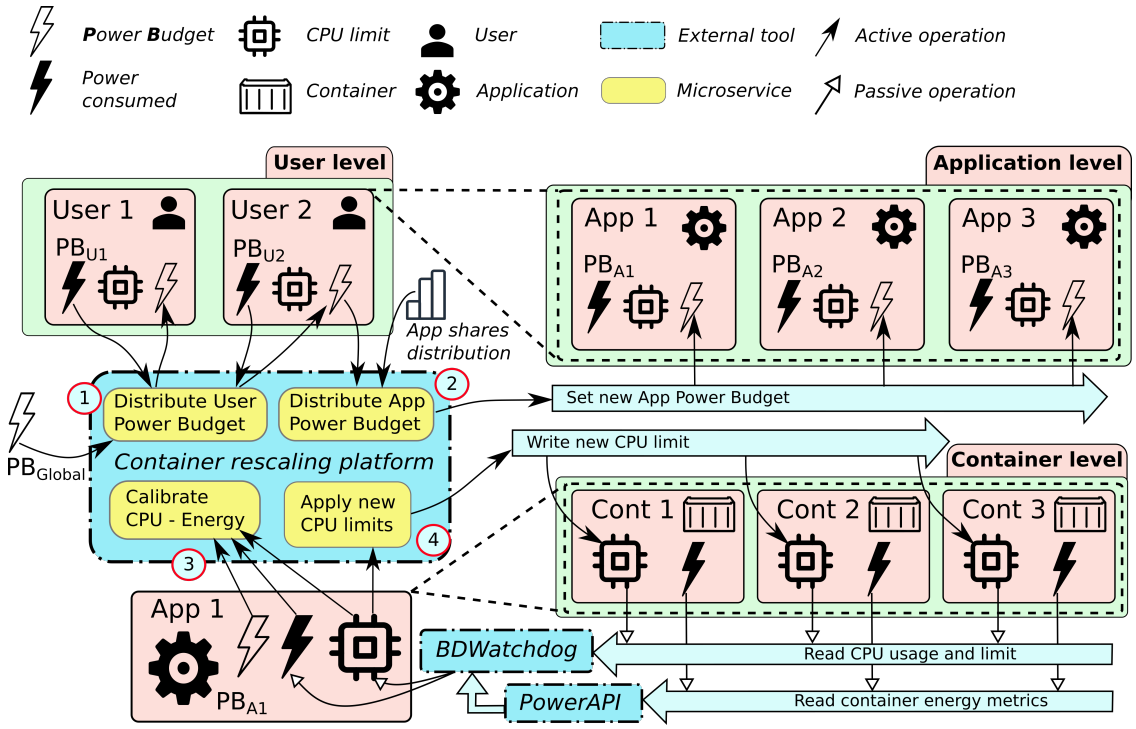


Figure 4.1: High-level overview of the platform architecture

level, an application acts as an abstract structure that adds up all the containers' CPU and energy metrics and, likewise, allows to set a power budget that has to be respected by all the grouped containers as a whole. Users would be placed at the highest level, with applications attached to them. Similar to applications, each user has an accounting for the grouped energy and CPU and a power budget, which is enforced down to the user's applications through their own budgets. Note that containers do not have any specific energy limit in our platform, as power control is only considered at the user and application levels. As containers are treated as black boxes that are part of a more complex structure (i.e., an application) there is no point in limiting the energy of specific containers.

With the above hierarchy laid out, the procedure to propagate a global power budget down to the CPU limits of the containers can be described, from top to bottom, in four steps (numbered in Figure 4.1): 1) the global power budget is dynamically divided among the users according to their usage, based on the aggregated energy consumption; 2) at the user level, the power budget is statically distributed among all its applications according to a ratio configured by the user, thus setting a

power budget for each application; 3) at the application level, the power budget is compared to the actual energy that is being consumed at the moment to determine if any operation is needed to be carried out. If that is the case, a request is created with the number of CPU shares to be increased or decreased; and 4) if such request is created, it is applied across all the containers by adapting their CPU limits using cgroups. The first two steps defined are fairly simple, as they merely distribute a power budget dynamically (step 1) or statically according to a set of ratios (step 2). However, steps 3 and 4 are more complex and are further described in Sections 4.3.2 and 4.3.3, respectively. All these steps, referred to as active operations, are carried out in order as needed (see Figure 4.1).

Regarding the external tools chosen to implement this platform, several had to be adapted for the task in hand. For the core feature of scaling the containers' CPU, the platform described in Chapter 3 was extended by implementing an alternative resource tuning policy to work with energy caps and power budgets, as well as to support applications and users in such rescaling policy instead of just containers like with the serverless policy. To provide the new policy, several new microservices were implemented (see them in Figure 4.1). For resource monitoring, two different tools were used for measuring CPU and energy metrics. On the one hand, PowerAPI [32, 46] was used for energy monitoring as it is capable of reporting container energy usage in real time with high accuracy. In this case, PowerAPI was also extended from supporting only Docker containers to include LXC. On the other hand, BDWatchdog was used for monitoring CPU utilization, aggregating both the user and kernel usage per container into a single metric measurement. The energy metrics reported by PowerAPI were also stored as time series within BDWatchdog in order to be retrieved in the same manner as the CPU metrics. In contrast to the aforementioned active operations, both monitoring tools carry out the passive operations, that is, they are configured to continuously poll the containers' CPU and energy usages with a given frequency.

4.3.2. Power budgeting policy

The time series for CPU usage and energy consumption of applications play a key role when making the decision to change the CPU limits during step 3 of Figure 4.1.

A simple policy that only considers energy can be used: if the energy consumed exceeds the power budget, a request to scale down the CPU limits is generated; otherwise, if it is below the cap, maybe the CPU limits need to be increased as the application could be suffering a CPU bottleneck. However, this policy can easily cause instability as the CPU patterns of Big Data applications can be unpredictable at times considering that they generally go through multiple processing stages that demand different amount of resources. To mitigate this issue, the CPU usage of the application is also taken into account along with energy. Hence, we define a richer and more robust policy considering both CPU usage and energy consumption, as shown in Table 4.1.

To illustrate the behavior of the platform when deciding if an action has to be taken, we use Figure 4.2 together with Table 4.1. This figure reports on the

Table 4.1: Policy decision criteria

State	CPU Usage	Energy Consumption	Decision
1	-	In power budget	Keep CPU
2	-	Above power budget	Decrease CPU
3	Low/Medium	Under power budget	Keep CPU
4	High/Bottleneck	Under power budget	Increase CPU

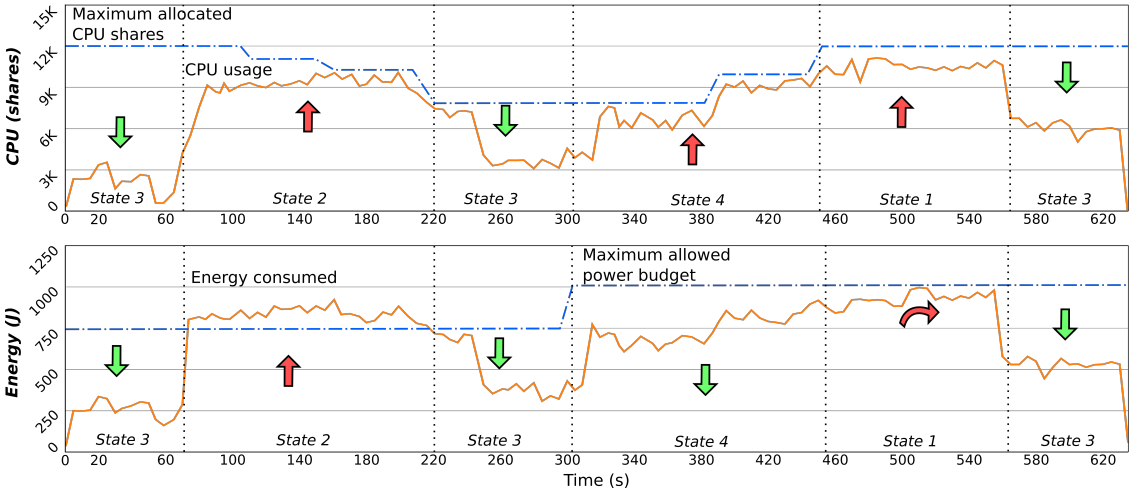


Figure 4.2: Time series for the CPU usage (top) and energy consumption (bottom) of the application

time series for the aggregated CPU usage and energy consumption of a synthetic workload used as a representative example. At the beginning of the execution, CPU and energy remain low from second 0 to around 70, so there is no need for any action (State 3 in Table 4.1). Then, from second 70 to around 220, there is a sharp increase in CPU usage and, in turn, in energy consumed. This sudden CPU spike causes the energy consumed to be above the initial power budget (750 J/s), thus prompting the platform to act and scale down the CPU shares to reduce energy consumption (State 2). This action decreases CPU shares from 12,000 (i.e., 120 cores) to about 8,000, which causes energy to be under the cap from second 220 to around 300. Note that, around second 220 to 240, the CPU limit is the exact one to cause energy consumption to be just below the cap, which can be seen as a perfectly controlled and stable state. Both CPU and energy remain low from second 250 to 300, so no changes are needed and the platform goes back to State 3, with the difference that the CPU limit is now more realistic according to the power budget. At second 300, the budget is increased from 750 J/s to 1,000 J/s while at the same time the CPU usage rises significantly until it becomes a bottleneck, causing the platform to transition to State 4. This state (the opposite of State 2) implies that the application is using close to all of its CPU shares but, at the same time, is not fully using its power budget. To act accordingly, the CPU limit is progressively raised up to 12,000 shares until the energy consumed is near but below the budget (around second 460). From second 460 to around 560, the platform enters State 1, where, although the CPU is near a bottleneck, energy consumption is really close to the limit and thus no action is taken. Finally, CPU usage and energy consumption decrease after second 560, so the platform switches to State 3, but keeping the CPU limit at 12,000 shares.

4.3.3. CPU scaling policy

A request to scale up/down the CPU shares of an application, which groups multiple containers, must be translated to individual requests for each container during step 4 of Figure 4.1. Considering that such request can only be to either decrease or increase the CPU limit to lower or raise energy consumption, respectively, the policy to translate such application requests to container requests can be determined as follows. On the one hand, if the request is to increase the CPU limit, the policy raises the limit to those containers that have a lower CPU utilization

(i.e., the containers whose CPU usage is the farthest from its limit). This decision relies heavily on the fact that the applications executed generally have similar CPU patterns across all the containers (i.e., the reason behind their grouping), specially in Big Data scenarios. On the other hand, if the request is to decrease the CPU limit, the policy reduces the limit to those containers with the highest CPU usage, considering that this is the fastest way of decreasing the overall energy consumed by the application.

4.4. Big Data use cases

To analyze the efficiency of our energy control policy, we consider a realistic Big Data setup described as follows. Three experiments that represent Big Data use cases from different domains are evaluated to prove the feasibility and effectiveness of the proposed platform: 1) static power budgeting using a machine learning workload; 2) application-level budgeting running a genome analysis pipeline; and 3) user-level budgeting using a streaming application. It is worth noting that although the deployed applications use Big Data technologies, they do not rely heavily on I/O but rather on in-memory processing and thus they are mainly CPU-bound. The characteristics of the hardware used and the configuration of the platform are detailed in Section 4.4.1, whereas the experiments are described in Section 4.4.2. Finally, Section 4.4.3 discusses the obtained results.

4.4.1. Hardware & platform configuration

To deploy the experimental testbeds, which consist of a set of containers, several nodes from the Grid'5000 infrastructure [54] have been used, referred to from now on as 'hosts'. The experiments are carried out on LXC-based container clusters deployed using up to 8 physical hosts. As described in Table 4.2, two types of hosts with different hardware characteristics are used for the containers. Each LXC container runs Ubuntu 18.04 LTS and Java JDK 1.8.0_212.

Regarding the tools, BDWatchdog and PowerAPI are deployed using a monitoring polling frequency of 5 seconds, while the container rescaling platform is used with

Table 4.2: Host hardware configuration

	Host type 1	**Host type 2**
CPU	2x Intel Xeon Gold 6126 @2.60GHz [24 cores]	2x Intel Xeon E5-2680 v4 @2.40GHz [28 cores]
Memory	192 GiB DDR4	768 GiB DDR4
Disks	2x 480 GiB SSD SATA Intel (OS) 4x 4 TiB HDD SAS (data)	2x 400 GiB SSD SAS Toshiba (OS) 2x 4 TiB HDD SAS (data)
Network	2x10 Gbps Ethernet	2x10 Gbps Ethernet

Table 4.3: Experimental configuration

	Testbed 1	**Testbed 2**	**Testbed 3**
#Containers	32	16	16
CPU limit (shares)	600	700	600
Memory	45 GiB	156 GiB	45 GiB
#Hosts and type	8 type 1	4 type 2	4 type 1

the new policy for this scenario to scale CPU resources using energy as input metric. In addition, the platform is configured to work with 40-second time windows.

4.4.2. Experimental configuration

The experiments are designed with the aim of showing representative use cases of the platform with different objectives to prove. Table 4.3 lays out the differences in terms of the hardware used for each experiment, mainly the number of containers used and the amount of resources. Four containers per physical host are used for all the experiments, as this is a good ratio to distribute the pool of host resources across a set of containers to avoid either a group of containers with few but fat instances, or many but thin instances. Each experiment is detailed below.

Static power budgeting

This experiment serves to prove that the platform is able to effectively throttle the energy consumed by the applications in real time and with no previous informa-

tion about the workloads, which are treated as black boxes just like the containers. As shown in Table 4.3, this experiment is deployed on a 32-container cluster across 8 hosts (Testbed 1). These containers are grouped to create a Hadoop 2.9.0 [101] cluster where Big Data workloads can be executed. The Big Data Evaluator (BDEv) tool [107] is used to set up Hadoop and to run the workloads in an automatic way.

In this experiment, a single user deploys one application running KMeans, an iterative clustering technique, using Spark 2.3.0 [114]. The input dataset contains 20 million samples and the algorithm performs up to 10 iterations using 30 clusters. This workload was chosen due to its iterative behavior and variable CPU and energy patterns, with periods of high and low load.

Application-level power budgeting

The second experiment aims to show how energy can be managed and shared among different applications as any other reserved and finite resource such as CPU or memory. More specifically, a certain energy amount is allocated to a single user and then shifted across multiple applications according to a time-varying demand. This shifting can be seen as a power budget that is adapted across the applications as the execution evolves, acting as some sort of sliding power window. This experiment uses 16 containers deployed on 4 hosts (see Testbed 2 in Table 4.3).

The experiment executed by the user is PathSeq [69], a bioinformatics pipeline for the identification and discovery of microorganisms on human tissue or biological samples. This pipeline detects and accounts for the present microorganisms through several processes that first remove any human genome sequence and later match the remaining ones (i.e., non-human sequences) with those of multiple microorganisms, such as bacteria, amoeba or fungi. All tasks of the pipeline are implemented on three separate applications that are executed on the platform, as shown in Figure 4.3. These applications are: 1) Auxiliary (2 containers), which creates all the required files that are used later on (e.g., the dictionary for the different bacteria datasets); 2) Preprocessing (4 containers), which aligns the human samples to a reference genome and then filters out any human sequence; and 3) Processing (10 containers), which aligns the remaining sequences to those of the microorganisms in search of matches, that are later counted for and presented through a final report.

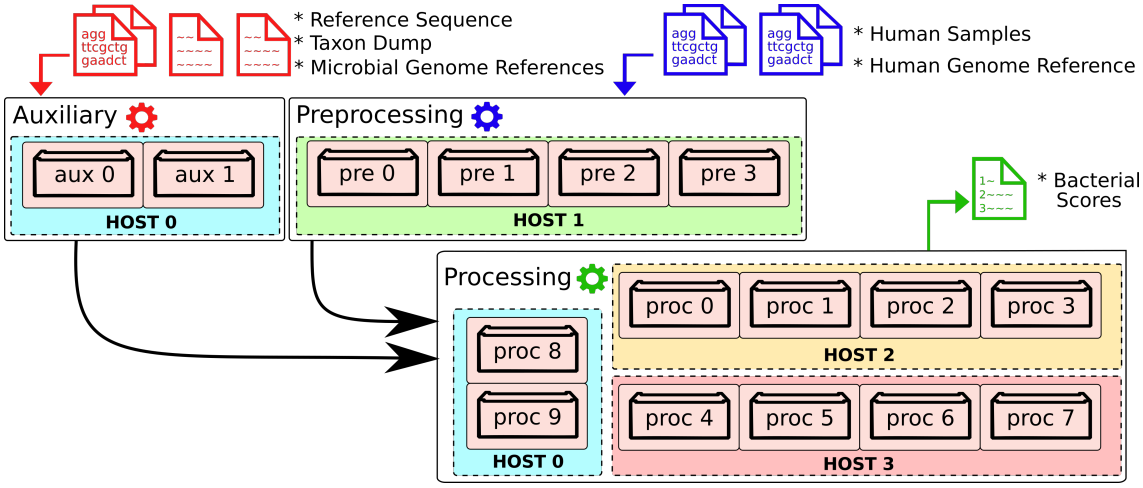


Figure 4.3: Overall container layout of the PathSeq pipeline

The PathSeq pipeline is executed using 12 microbial genome reference files as input to look for matches on a human sample file of around 6 GiB. This file consists of stool samples from healthy individuals obtained from the European Nucleotide Archive (ENA) [72], while the other files (e.g., microbial references) were downloaded from the National Center for Biotechnology Information (NCBI) website [83]. More specifically, the microbe files are processed in parallel by the Auxiliary application (6 files per container), while the human sample is aligned by the Preprocessing application. Next, the Processing application matches this sample against the resulting microbe files. As the Auxiliary and Preprocessing applications are not dependent, they are executed in parallel, albeit the Preprocessing stage is generally longer. To run this pipeline, GATK [22] is used since it allows Spark to be integrated with a Hadoop cluster to parallelize the operations executed during the Preprocessing and Processing stages [109].

User-level dynamic power budgeting

The last experiment shows how a global power budget for the entire infrastructure can be dynamically divided among users according to their needs. The experiment considers two different users with separate and dedicated environments running a streaming application. Each environment consists of: 1) a data generator from the HiBench 7.0 suite [61]; 2) a message broker based on Kafka 2.1.0 [50]; and 3) a

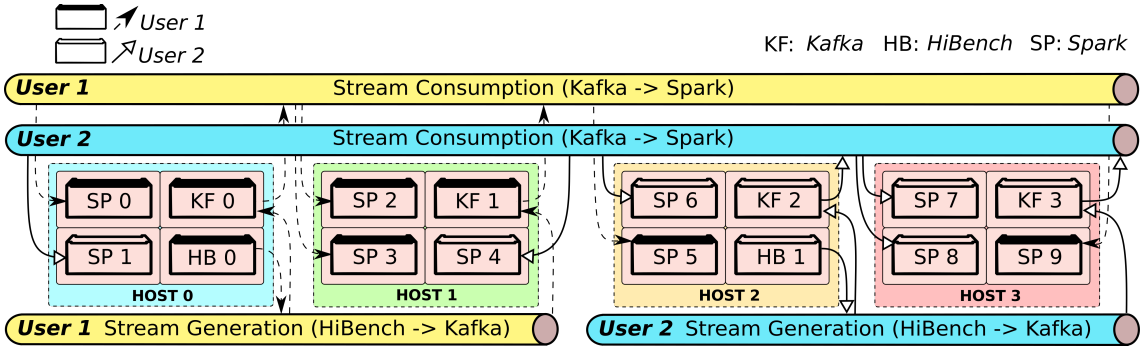


Figure 4.4: Overall container layout of the streaming application

Spark 2.3.0 cluster that runs the FixWindow streaming workload, also obtained from HiBench.

As shown in Table 4.3, 16 containers are used in this experiment (see Testbed 3), deployed on 4 hosts following the layout described in Figure 4.4. For each user, a total of 8 containers are used with 1 HiBench data generator, 2 Kafka brokers and 5 slave nodes for Spark. To simulate varying loads for the users, the streaming application is configured with three stream sizes that can be changed at any time for any user, named large (171 MiB/s), medium (76 MiB/s) and small (38 MiB/s). Finally, the streams are processed in 10-second windows.

Note that although for the first two experiments specific power caps (previously acquired knowledge) are applied, the third one presents a scenario where energy is dynamically distributed. This last scenario could also be used in order to estimate power caps if a static cap is desirable.

4.4.3. Experimental results

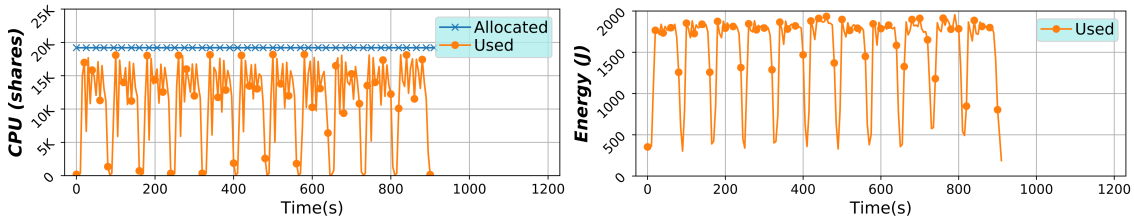
The results for the three experiments are presented in the next subsections. For the first and second experiments, we compare a scenario where each experiment is executed without any power budgeting or interference overall from the platform, referred to as *baseline*, with a scenario where capping is applied, named as *capped*. All the plots show the aggregated CPU usage (shares) and energy consumption (Joules) during the execution of the experiments, generated by performing an integral with the trapezoidal rule, which provides good accuracy as the time series have discrete

values with a frequency of 5 seconds.

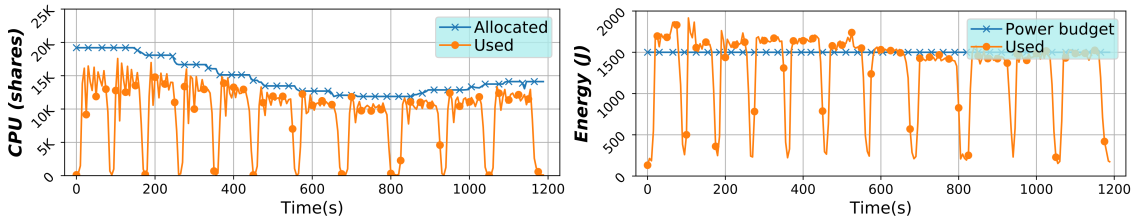
Five metrics are used to compare both scenarios and expose the strong points as well as the drawbacks of the platform: 1) runtime (in minutes); 2) total amount of energy consumed (J); 3) average energy consumed per second (J/s); 4) total amount of CPU usage (core-minutes), calculated as the aggregated number of cores that the experiment used times its runtime; and 5) the energy efficiency ratio as defined at the end of Section 4.1 (core-minutes/J). Although the plots in the following subsections only show the most representative execution for each experiment, the metrics provided are obtained from the average of a minimum of 5 executions.

Static power budgeting

Figure 4.5 shows CPU usage (left plots) and energy consumption (right plots) for KMeans. It can be seen how after setting a limit of 1,500 J/s for the capped scenario, energy consumption is brought down under control after enough time passes (around second 600, see right plot in Figure 4.5b). The power budget set to this workload aims to bring the peaks of energy consumption of up to 1,800 J/s in the baseline scenario down to under 1,500. The time until energy is finally controlled is needed because the workload began the execution with full CPU resources available.



(a) Baseline scenario



(b) Capped scenario

Figure 4.5: CPU usage and energy consumption for KMeans

It is interesting to note that the CPU limit reached at second 700 (around 12,500 shares) appears to be lower than the one strictly required. This in turn causes energy consumption to drop below the maximum between seconds 700 and 800 (8th iteration of the algorithm). During this iteration, the platform switches to State 4 (see Table 4.1), where the CPU usage has a bottleneck and simultaneously the energy consumed is below the cap. The platform tackles this issue and transitions to the really stable State 1 by slightly increasing the CPU resources, staying in that state for the remaining iterations.

As a result of the limit imposed, the average energy consumed is reduced from 1,450 J/s for the baseline scenario to 1,320 J/s (-9%) for the capped one. Regarding total consumption, it is 1,356 and 1,638 kJ (+21%), with runtimes of 15 and 20 minutes (+33%), respectively. The total CPU usage remains the same for both scenarios (1,783 core-minutes). The explanation for these metrics lies in the fact that, while the workload is not affected (i.e., it does the same operations), its runtime obviously increases due to the CPU restrictions. In turn, the longer runtime increases the total energy consumption due to the lack of energy proportionality of the CPU, as mentioned at the end of Section 4.1, which heavily penalizes longer runtimes due to the idle energy consumption.

Application-level power budgeting

Figure 4.6 presents CPU usage and energy consumption for the three applications that compose the PathSeq pipeline in the baseline scenario. Although the first two applications begin simultaneously, as mentioned in Section 4.4.2, it can be observed that Auxiliary ends much sooner. Immediately after Preprocessing, the Processing application starts, having a runtime of around 20 minutes. The metrics for each application and for the pipeline as a whole are provided in Table 4.4. While Auxiliary and Preprocessing are close in terms of average energy consumption for the baseline scenario (92 and 114 J/s, respectively), Processing goes up to 322 J/s. This difference is important to properly distribute a limited power budget across the applications without resulting in severe imbalances.

To configure the platform and specify the power budget, we take into account the following points for Auxiliary and Preprocessing: 1) they are executed simultaneously;

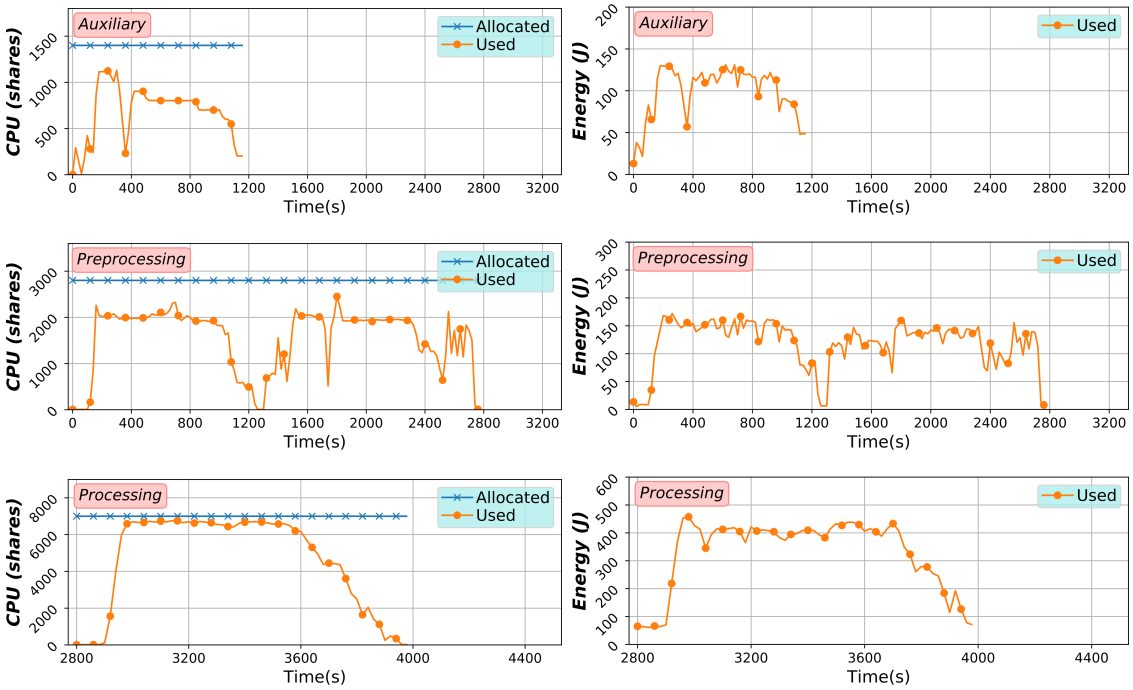


Figure 4.6: CPU usage and energy consumption for PathSeq (baseline scenario)

Table 4.4: Metrics for the PathSeq pipeline

	Auxiliary		**Preprocessing**		**Processing**		**Total**	
	Baseline	*Capped*	*Baseline*	*Capped*	*Baseline*	*Capped*	*Baseline*	*Capped*
Total Energy (kJ)	108	129 (+19%)	321	319 (-1%)	416	471 (+13%)	845	919 (+9%)
Average Energy (J/s)	92	82 (-11%)	114	104 (-9%)	322	273 (-15%)	205	192 (-6%)
Total CPU usage (core-minutes)	132	135 (+2%)	721	716 (-1%)	903	904 (0%)	1,756	1,755 (0%)
Runtime (minutes)	20	26 (+30%)	47	51 (+9%)	22	29 (+32%)	69	80 (+16%)
Energy efficiency (core-minutes/kJ)	1.22	1.05 (-14%)	2.25	2.24 (0%)	2.17	1.92 (-12%)	2.08	1.91 (-8%)

2) both require more or less the same average energy per second as mentioned before; and 3) Auxiliary finishes sooner. With this information in hand, the power budget is defined as follows: 1) while Auxiliary is running, 270 J/s are split as 90/110/70 J/s for Auxiliary, Preprocessing and Processing, respectively; 2) as soon as Auxiliary finishes and while Preprocessing is still running, the same 270 J/s are split as 10/190/70 J/s; and 3) once Processing begins its execution, 350 J/s are split as 10/20/320 J/s. As can be seen, this budget splits the energy at the beginning so that both Auxiliary and

Preprocessing are balanced. Then, it shifts the energy freed by Auxiliary so that Preprocessing can use it, and finally it moves most energy to Processing. Considering that the last stage is significantly more energy demanding, the power budget is increased from 270 to 350 J/s to further show the dynamic capabilities of the platform. It is important to remark that increasing the power budget and changing the shares for energy distribution are actions performed dynamically and in real time by the platform, without requiring any stopping or modifying the applications, which are overall unaware of these changes. Regarding the energy allocated to applications that are idle (e.g., Processing during the first two stages), it is taken into account that even the idle containers consume a minimum amount of energy in order to establish a realistic power budget.

Figure 4.7 reports on the results of the energy-capped pipeline using this budget. As can be observed, the Auxiliary and Preprocessing applications effectively split the power budget until Auxiliary finishes at second 1,500. At this time, Preprocessing finishes its first task (the alignment). At this point, the energy allocated to Auxiliary is shifted to Preprocessing so that its second task (the filter) can use a higher power

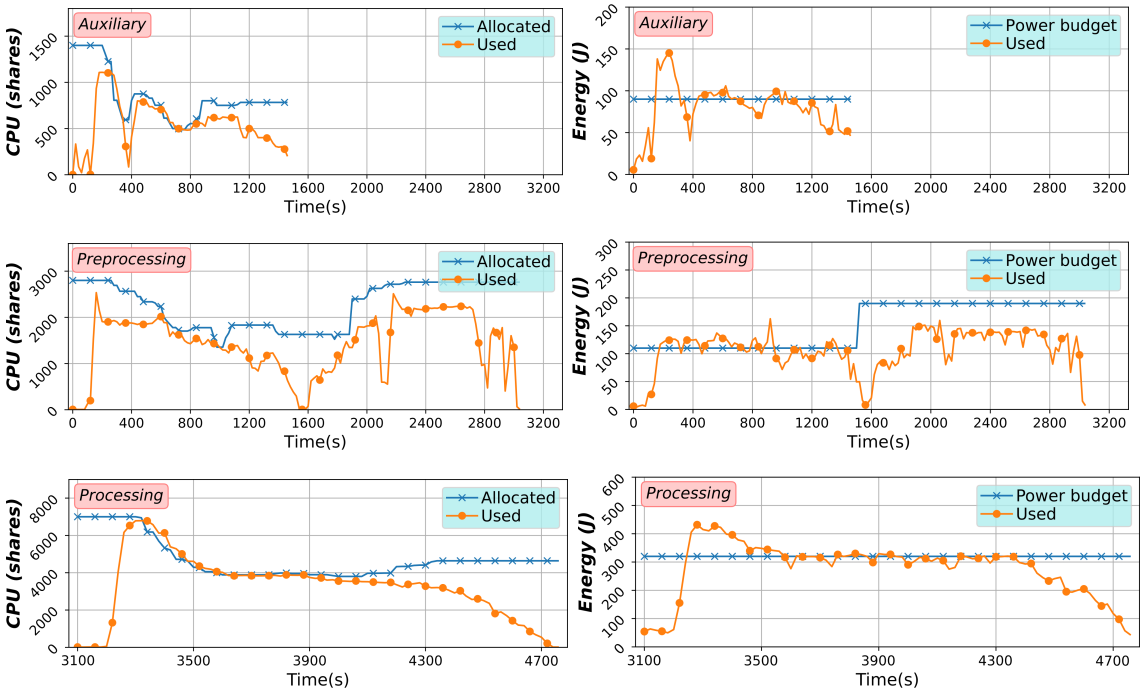


Figure 4.7: CPU usage and energy consumption for PathSeq (capped scenario)

budget increased from 110 to 190 J/s. Once the cap is raised by the platform, the Preprocessing application suddenly transitions from a stable State 1 to State 4 (see Table 4.1), where the CPU has a bottleneck while the energy consumed is lower than the new power budget. This situation is tackled via a scaling up operation on the CPU shares allocated to the containers, carried out from around second 1,800 to 2,300. Once Preprocessing finishes, Processing starts and the platform moves nearly all the available budget (320 J/s) towards this application.

The metrics for the capped scenario are also shown in Table 4.4. For the first two stages of the pipeline, the capping decreases the average energy consumed for Auxiliary and Preprocessing by 11% and 9%, respectively. However, Auxiliary is clearly more affected by the capping as, being a significantly shorter application running on a smaller set of containers, it is quickly brought under the limit, where it spends most of its execution. On the other hand, the limit is applied to Preprocessing, but it is only affected for a shorter amount of time, as its second task benefits from the higher cap. This difference between both applications explains why for the first one there is a runtime overhead of 30%, while for the second one it is only 9% and also having no overhead in the total energy consumption. Regarding Processing, the power budget is clearly enforced as the average energy consumed is reduced by 15%. This in turn causes a runtime overhead of 32%. When it comes to the total amount of CPU usage, it has little variation between both scenarios for all pipeline stages, as the CPU shares consumed are the same overall.

Finally, it is interesting to analyze the energy efficiency ratio as a guideline to quantify the amount of CPU that could be profited from an amount of energy consumed in a given time period. If the values for such metric are compared across the different applications (see Table 4.4), it is easy to see that the values for Preprocessing and Processing are considerably higher when compared to Auxiliary. That is, such applications are executed in a more energy-efficient way. Taking the baseline as an example, Preprocessing used 2.25 core-minutes/kJ, while Auxiliary only 1.22. This difference arises from the energy proportionality of the CPU as previously discussed, and also by the fact that the higher the CPU usage is on the physical host, the higher the energy efficiency. Note that Preprocessing runs on 4 containers deployed on Host 1 (see Figure 4.3), whereas Auxiliary runs on 2 containers within Host 0, the other 2 sibling containers belonging to Processing (idle when Auxiliary is running).

User-level dynamic power budgeting

Figure 4.8 shows CPU usage and energy consumption for the streaming application executed by two different users. For this experiment, three stages are created to emulate different scenarios, named in order of execution (see Table 4.5): 1) Balance, where User 1 incrementally scales up its stream (3 streams, small-medium-large, see Section 4.4.2) while User 2 does the same in reverse (large-medium-small); 2) Contention, where both users process a large stream to create a contention scenario; and 3) Efficiency, where User 1 processes 3 consecutive large streams while User 2 first remains idle and then processes two streams of small and large size, respectively.

The objective of these stages is different. On the one hand, Balance and Contention aim to show how a global power budget for the entire infrastructure can be shared and balanced among users and their applications. The difference between both lies

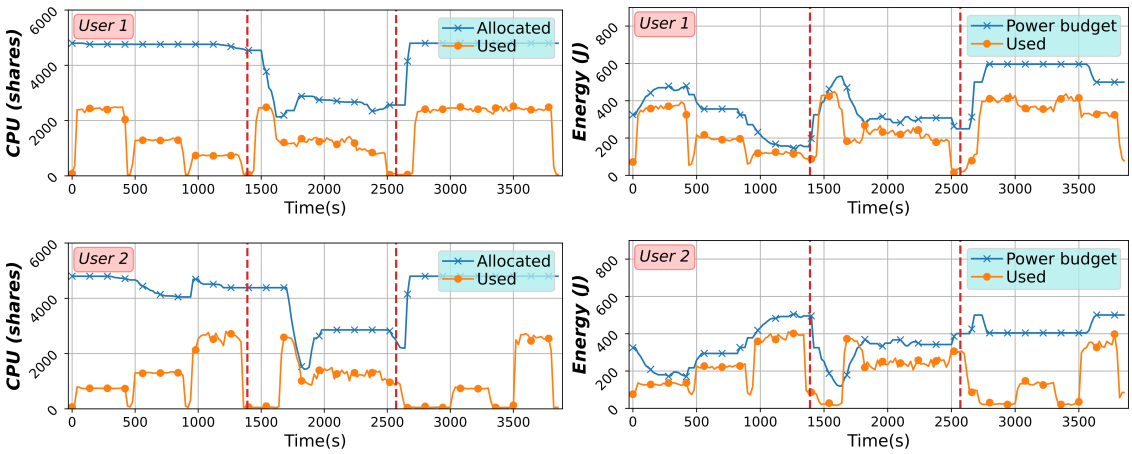


Figure 4.8: CPU usage and energy consumption for streaming (stages Balance, Contention and Efficiency split by dashed lines)

Table 4.5: Average energy consumption (J/s)

	Balance			**Contention**	**Efficiency**		
	stream				stream		
	1	2	3		1	2	3
User 1	321	185	117	210	386	324	306
User 2	124	204	323	215	28	119	317

in the fact that, while for the Balance stage the budget is enough to accommodate both users, as their CPU and energy requirements are inversely correlated over time, in the Contention stage both users have simultaneously high resource requirements and the budget has to be split. In this latter case, as the budget is not enough to process two large streams, a scale down process has to be carried out. On the other hand, the Efficiency stage serves to further analyze the variation of the energy efficiency according to the load of the underlying physical host, which unfortunately affects the users even if their processing environments are virtually separated.

As can be seen in Figure 4.8, the Balance stage lasts from the beginning to around second 1,400, with each stream running for 450 seconds. An initial power budget of 650 J/s is shared between both users according to their requirements, shifting a budget of around 475 J/s from User 1 in the first stream to User 2 at the end of the stage. After Balance finishes, the Contention stage begins lasting from second 1,400 to around 2,600. In this stage, the same budget (650 J/s) is similarly split between users. However, because this budget is insufficient to cater for both users, their streaming applications are scaled down from an expected average energy consumption of around 320 J/s when processing a large stream (see first stream of User 1 for the Balance stage in Table 4.5) down to 210 J/s. Finally, the Efficiency stage begins at around second 2,600 and the global power budget is increased from 650 J/s to 1,000 J/s (in this stage the budget enforcement is not the focus). It can be seen in Table 4.5 how the streaming application of User 2 lowers the average energy consumption of User 1: from 386 J/s (User 2 is idle) to 324 (-16%) and 306 (-21%) when User 2 is processing the small and large streams, respectively. This behavior can be explained by the fact that the higher the underlying host utilization, the higher the energy efficiency (i.e., less amount of energy for the same processing requirements). Using the energy efficiency ratio, the values obtained for the Efficiency stage are 1.02, 1.13 (+11%) and 1.24 (+22%) core-minutes/kJ for the idle, small and large streams, respectively. This further proves how the most efficient scenario is the one where the underlying hosts have the CPU usage as high as possible, in this case, when both users are processing a large stream.

4.5. Conclusions

In this chapter, energy has been presented as another system resource alongside CPU that can be likewise shared and split across users, applications (as a set of containers) and the containers themselves. Moreover, it can be capped at the software level as any other resource to ensure that energy consumption is kept below a certain limit. To implement this concept, we have deployed a new scenario created from the combination and extension of the two frameworks described in Chapters 2 and 3 to support energy metrics and implement a new tuning policy, respectively. In this scenario, the CPU shares of containers can be scaled down or up in order to either reduce the energy or allow it to raise. To prove that such energy management can be effectively carried out for Big Data applications as a representative real-world environment, three experiments have been assessed to expose multiple use cases.

The experimental results have demonstrated that it is possible to transparently enforce a certain power limit without incurring any overhead in terms of the amount of CPU required to complete a particular task. Such limit is configurable and can be adapted to the specific needs of users and applications. Furthermore, our platform can manage a power budget that is dynamically distributed, either across multiple applications of the same user as in the case of a bioinformatics pipeline, or between several users as shown with a streaming application. Finally, any energy-related study using multiple CPUs and hosts has to deal with issues tied to the lack of energy proportionality of current CPU microarchitectures. This chapter has also provided some insight into such issues and the impact they have in real experiments.

The platform is publicly available at `http://bdwatchdog.dec.udc.es/energy`.

Chapter 5

Conclusions and future work

Next, we summarize the main contributions of the Thesis and provide some insight on future research directions.

Conclusions

Currently, Big Data applications are increasing in size and becoming much more complex in terms of both the resources they need and the underlying infrastructures most commonly used to deploy them, with a clear evolution. On the one hand, the initial MapReduce paradigm was implemented through batch processing that relied heavily on disk and network, being deployed mainly on commodity clusters. On the other hand, the current state-of-the-art processing engines (e.g., Spark, Flink) also support stream and in-memory processing and involve a higher and more heterogeneous resource consumption, from disk and network to memory and CPU, and, in turn, energy. Furthermore, these frameworks are often deployed on virtual clusters and Cloud platforms. This Thesis has explored such evolution to define and propose novel ways for resource analysis and tuning of Big Data workloads when they are deployed on container-based virtual clusters. Considering this as a twofold objective regarding resources, with passive analysis as the first and active tuning as the second one, this Thesis has presented in detail two frameworks that address each of the objectives independently as well as an additional scenario to specifically manage

energy consumption. This latter scenario comes as a result of the combination and extension of the two previous frameworks with other required third-party tool.

The first proposed framework, BDWatchdog, has been specifically designed to provide fine-grain resource monitoring of Big Data applications as time series, as well as real-time and transparent profiling through stack sampling. On the one hand, the monitoring functionality is able to work at the process level, allowing to continuously collect metrics in a close to real-time manner, being in turn able to serve them afterwards by using filters and aggregations. It is interesting to note how this process-based monitoring allows to inherently support new virtualization technologies such as containers. On the other hand, the profiling functionality enables to analyze further the applications, thus gaining deeper insight into how they are using the system resources provided to them. Several use cases have been studied through extensive experimentation with BDWatchdog, demonstrating how these functionalities combined together can give a more detailed picture to users about application's behavior, even in real time. Furthermore, these experiments have shown that this framework is capable of scaling with the application's size while also imposing a low overhead.

The second proposal of this Thesis builds upon BDWatchdog to create a rescaling framework for containers where resources are not only monitored, but rather acted upon to set limits according to different tuning policies or use cases. One of such policies, as described in Chapter 3, depicts a serverless environment where CPU and memory resources are scaled according to the real usage of the applications by properly modifying the underlying containers' limits. This framework has proved that it is possible to build a serverless platform with flexible virtualization technologies such as containers and, in addition, that Big Data workloads can be successfully executed on such platform with a manageable overhead. Aside from providing the user with the advantages of the serverless paradigm, another main benefit of the proposed framework lies on the increased resource utilization. As an example, a 77% increase has been obtained when running a streaming workload, considering that is well adapted to the serverless paradigm as its resource usage is rather stable and does not present a high variability over time.

Finally, this Thesis has presented an extended scenario which combines both frameworks with a third-party tool (PowerAPI) in order to allow to manage energy

on container clusters. By implementing an alternative tuning policy for the rescaling framework, energy can be treated as another resource along CPU, allowing to implement the features of energy capping and power budgeting. This scenario has shown that these features allow to deploy interesting use cases for Big Data. On the one hand, energy capping enables to effectively control the energy consumed by an application or user without actively causing any overhead on the CPU resources they require. On the other hand, power budgeting allows to share and distribute an energy quota across applications and/or users, either as desired or to dynamically adapt to the varying energy demand.

Future work

The frameworks presented in this Thesis have been designed to be easily extended or adapted, as well as to be able to include new or alternative base tools for their functionality. In fact, the scenario described in Chapter 4 is a clear example of such flexibility, by adapting and integrating the two previously proposed frameworks with PowerAPI to explore a new use case focused on energy management. However, it is still possible to extend them even further.

For BDWatchdog, it would be of great interest to improve its profiling capabilities so that other resources, such as disk, memory and network, can also be analyzed. These features may be implemented following the same design as for the CPU profiling, that is, by sampling low-level performance metrics that are used to later create flame graphs. Such profiling has been proved to be feasible thanks to novel technologies like eBPF and the BCC toolkit [21]. Furthermore, the resource time series provided by BDWatchdog are useful for other use cases such as pattern-based job classification or anomaly detection, using in both cases such time series as input for machine learning algorithms.

Regarding the resource scaling framework, it has to be noted that currently only vertical scaling is supported, that is, only the resources of the containers that host a certain application are scaled in terms of several tuning policies such as the serverless paradigm. Although this feature is able to deal with highly varying demands as it grants very fine-grain resource tuning, it would be interesting to also

support horizontal scaling. Relying on this feature, the framework would be able to act on extreme scenarios with either very low or very high resource demands by shutting down or spinning up instances, respectively. Furthermore, other resources such as disk and network may be scaled as well. This is supported inherently by the proposed framework as those resources can also be managed via appropriate mechanisms such as cgroups for disk and traffic control for network. Regarding the overhead imposed, it could be mitigated even more if both the framework and the applications interchanged information to aid in the scaling policy, considering that as of now the framework treats the containers as black boxes.

Finally, regarding energy capping and power budgets, some interesting use cases can be explored with minor extensions and adaptations of the current platform. For instance, the concept of power budget may be extended to other resources to create CPU or memory budgets. With these budgets, the base resource would be split and shared among the user's applications or the application's containers. The concept of restricting the CPU shares to cap energy consumption can also be slightly modified by changing energy for temperature, thus creating a scenario where the goal is to control temperature so that a configurable threshold is not exceeded. This scenario is useful as a software counterpart of the existing CPU throttling implemented in current hardware, which is generally less flexible and only used once the temperature has already risen considerably.

Bibliography

[1] Francisco Almeida, Marcos D. Assunção, Jorge Barbosa, Vicente Blanco, Ivona Brandic, Georges Da Costa, Manuel F. Dolz, Anne C. Elster, Mateusz Jarus, Helen D. Karatza, Laurent Lefèvre, Ilias Mavridis, Ariel Oleksiak, Anne-Cécile Orgerie, and Jean-Marc Pierson. Energy monitoring as an essential building block towards sustainable ultrascale systems. *Sustainable Computing: Informatics and Systems*, 17:27–42, 2018.

[2] Amazon Auto Scaling service. `https://aws.amazon.com/autoscaling/`. Last visited: July 2020.

[3] Amazon CloudWatch service. `https://aws.amazon.com/cloudwatch/`. Last visited: July 2020.

[4] Amazon EC2 service. `https://aws.amazon.com/ec2/`. Last visited: July 2020.

[5] Amazon ECS service. `https://aws.amazon.com/ecs/`. Last visited: July 2020.

[6] Amazon EMR. `https://aws.amazon.com/emr/`. Last visited: July 2020.

[7] Amazon Fargate service. `https://aws.amazon.com/fargate/`. Last visited: July 2020.

[8] Amazon Lambda service. `https://aws.amazon.com/lambda/`. Last visited: July 2020.

[9] Amazon Web Services (AWS). `https://aws.amazon.com/`. Last visited: July 2020.

[10] J. Chris Anderson, Jan Lehnardt, and Noah Slater. *CouchDB: The Definitive Guide: Time to Relax*. O'Reilly Media, Inc., 2010.

[11] Michael Anderson, Shaden Smith, Narayanan Sundaram, Mihai Capotă, Zheguang Zhao, Subramanya Dulloor, Nadathur Satish, and Theodore L. Willke. Bridging the gap between HPC and Big Data frameworks. *Proceedings of the VLDB Endowment*, 10(8):901–912, 2017.

[12] Amedeo Asnaghi, Matteo Ferroni, and Marco D. Santambrogio. DockerCap: a software-level power capping orchestrator for Docker containers. In *Proceedings of the 14th IEEE International Conference on Embedded and Ubiquitous Computing (EUC 2016)*, pages 90–97, Paris, France, 2016.

[13] atop: performance monitor for Linux. `https://www.atoptool.nl/index.php`. Last visited: July 2020.

[14] Azure HDInsight. `https://azure.microsoft.com/services/hdinsight/`. Last visited: July 2020.

[15] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, and Philippe Suter. Serverless computing: current trends and open problems. In *Research Advances in Cloud Computing*, pages 1–20. Springer, 2017.

[16] Ioana Baldini, Paul Castro, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, and Philippe Suter. Cloud-native, event-based programming for mobile applications. In *Proceedings of the International Conference on Mobile Software Engineering and Systems (MOBILESoft'16)*, pages 287–288, Austin, TX, USA, 2016.

[17] Manu Bansal, Eyal Cidon, Arjun Balasingam, Aditya Gudipati, Christos Kozyrakis, and Sachin Katti. Trevor: automatic configuration and scaling of stream processing pipelines. arXiv:1812.09442, 2018.

[18] Luiz André Barroso and Urs Hölzle. The case for energy-proportional computing. *IEEE Computer*, 40(12):33–37, 2007.

[19] David Bernstein. Containers and cloud: from LXC to Docker to Kubernetes. *IEEE Cloud Computing*, 1(3):81–84, 2014.

[20] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. D3: Data-Driven Documents. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2301–2309, 2011.

[21] BPF Compiler Collection (BCC). `https://github.com/iovisor/bcc`. Last visited: July 2020.

[22] Broad Institute. GATK: Genome Analysis Toolkit. `https://software.broadinstitute.org/gatk`. Last visited: July 2020.

[23] Rolando Brondolin, Tommaso Sardelli, and Marco D. Santambrogio. DEEPmon: dynamic and energy efficient power monitoring for container-based infrastructures. In *Proceedings of the 32nd IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW 2018)*, pages 676–684, Vancouver, BC, Canada, 2018.

[24] Rajkumar Buyya, Satish Narayana Srirama, Giuliano Casale, Rodrigo Calheiros, Yogesh Simmhan, Blesson Varghese, Erol Gelenbe, Bahman Javadi, Luis Miguel Vaquero, Marco A. S. Netto, Adel Nadjaran Toosi, Maria Alejandra Rodriguez, Ignacio M. Llorente, Sabrina De Capitani Di Vimercati, Pierangela Samarati, Dejan Milojicic, Carlos Varela, Rami Bahsoon, Marcos Dias De Assuncao, Omer Rana, Wanlei Zhou, Hai Jin, Wolfgang Gentzsch, Albert Y. Zomaya, and Haiying Shen. A manifesto for future generation cloud computing: research directions for the next decade. *ACM Computer Surveys*, 51(5):105:1–105:38, 2018.

[25] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache Flink: stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 38(4):28–38, 2015.

[26] Emiliano Casalicchio and Stefano Iannucci. The state-of-the-art in container technologies: application, orchestration and security. *Concurrency and Computation: Practice and Experience*, Article e5668, 2020.

[27] Jeffrey D. Case, Mark S. Fedor, Martin L. Schoffstall, and James R. Davin. RFC1157: Simple Network Management Protocol (SNMP). `https://www.rfc-editor.org/info/rfc1157`, 1990. Last visited: July 2020.

[28] CESGA Big Data PaaS service. `http://bigdata.cesga.es`. Last visited: July 2020.

[29] CESGA Supercomputing Center. `http://www.cesga.es/`. Last visited: July 2020.

[30] Surajit Chaudhuri and Umeshwar Dayal. An overview of data warehousing and OLAP technology. *ACM SIGMOD Record*, 26(1):65–74, 1997.

[31] Eun-Young Cho. JProfiler: code coverage analysis tool for OMP project. Technical report, CMU 17-654 & 17-754, 2006.

[32] Maxime Colmant, Mascha Kurpicz, Pascal Felber, Loïc Huertas, Romain Rouvoy, and Anita Sobe. Process-level power estimation in VM-based systems. In *Proceedings of the Tenth European Conference on Computer Systems (EuroSys'15)*, pages 14:1–14:14, Bordeaux, France, 2015.

[33] Tiziano De Matteis and Gabriele Mencagli. Keep calm and react with foresight: strategies for low-latency and energy-efficient elastic data stream processing. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'16)*, pages 13:1–13:12, Barcelona, Spain, 2016.

[34] Daniele De Sensi, Tiziano De Matteis, and Marco Danelutto. Simplifying self-adaptive and power-aware computing with Nornir. *Future Generation Computer Systems*, 87:136–151, 2018.

[35] Daniele De Sensi, Massimo Torquati, and Marco Danelutto. A reconfiguration algorithm for power-aware parallel applications. *ACM Transactions on Architecture and Code Optimization*, 13(4):43:1–43:25, 2016.

[36] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[37] Elif Dede, Madhusudhan Govindaraju, Daniel Gunter, Richard Shane Canon, and Lavanya Ramakrishnan. Performance evaluation of a MongoDB and Hadoop platform for scientific data analysis. In *Proceedings of the 4th ACM Workshop on Scientific Cloud Computing (ScienceCloud'13)*, pages 13–20, New York, NY, USA, 2013.

[38] Youwei Ding, Xiaolin Qin, Liang Liu, and Taochun Wang. Energy efficient scheduling of virtual machines in cloud with deadline constraint. *Future Generation Computer Systems*, 50:62–74, 2015.

[39] Mohammed El Mehdi Diouri, Manuel F. Dolz, Olivier Glück, Laurent Lefèvre, Pedro Alonso, Sandra Catalán, Rafael Mayo, and Enrique S. Quintana-Ortí. Assessing power monitoring approaches for energy and power analysis of computers. *Sustainable Computing: Informatics and Systems*, 4(2):68–82, 2014.

[40] Yanqing Duan, John S. Edwards, and Yogesh K. Dwivedi. Artificial intelligence for decision making in the era of Big Data – evolution, challenges and research agenda. *International Journal of Information Management*, 48:63–71, 2019.

[41] Jonatan Enes, Javier López Cacheiro, Roberto R. Expósito, and Juan Touriño. Big Data-oriented PaaS architecture with disk-as-a-resource capability and container-based virtualization. *Journal of Grid Computing*, 16(4):587–605, 2018.

[42] Jonatan Enes, Roberto R. Expósito, and Juan Touriño. BDWatchdog: real-time monitoring and profiling of Big Data applications and frameworks. *Future Generation Computer Systems*, 87:420–437, 2018.

[43] Jonatan Enes, Roberto R. Expósito, and Juan Touriño. Real-time resource scaling platform for Big Data workloads on serverless environments. *Future Generation Computer Systems*, 105:361–379, 2020.

[44] Jonatan Enes, Guillaume Fieni, Roberto R. Expósito, Romain Rouvoy, and Juan Touriño. Power budgeting of Big Data applications in container-based clusters. 2020 (Submitted for publication).

[45] Eve: a Python REST API framework. `http://python-eve.org/`. Last visited: July 2020.

[46] Guillaume Fieni, Romain Rouvoy, and Lionel Seinturier. SmartWatts: self-calibrating software-defined power meter for containers. In *Proceedings of the 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGrid'20)*, pages 479–488, Melbourne, Australia, 2020.

[47] Flame graph project. `https://github.com/brendangregg/FlameGraph`. Last visited: July 2020.

[48] Avrilia Floratou, Ashvin Agrawal, Bill Graham, Sriram Rao, and Karthik Ramasamy. Dhalion: self-regulating stream processing in Heron. *Proceedings of the VLDB Endowment*, 10(12):1825–1836, 2017.

[49] James Gardner. The Web Server Gateway Interface (WSGI). In *The Definitive Guide to Pylons*, pages 369–388. Springer, 2009.

[50] Nishant Garg. *Apache Kafka*. Packt Publishing Ltd, 2013.

[51] Google Cloud Functions. `https://cloud.google.com/functions/`. Last visited: July 2020.

[52] Google Compute Engine (GCE). `https://cloud.google.com/compute/`. Last visited: July 2020.

[53] Brendan Gregg. The flame graph. *Communications of the ACM*, 59(6):48–57, 2016.

[54] Grid'5000 testbed for experiment-driven research. `https://www.grid5000.fr`. Last visited: July 2020.

[55] Miguel Grinberg. *Flask Web Development: Developing Web Applications with Python*. O'Reilly Media, Inc., 2014.

[56] Gunicorn: a Python WSGI HTTP server for UNIX. `http://gunicorn.org/`. Last visited: July 2020.

[57] Rui Han, Li Guo, Moustafa M. Ghanem, and Yike Guo. Lightweight resource scaling for cloud applications. In *Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid'12)*, pages 644–651, Ottawa, ON, Canada, 2012.

[58] Rui Han, Li Guo, Yike Guo, and Sijin He. A deployment platform for dynamically scaling applications in the cloud. In *Proceedings of the 3rd International Conference on Cloud Computing Technology and Science (CloudCom'11)*, pages 506–510, Athens, Greece, 2011.

[59] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Serverless computation with OpenLambda. In *Proceedings of the 8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud'16)*, pages 33–39, Denver, CO, USA, 2016.

[60] Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma Bilgen Cetin, and Shivnath Babu. Starfish: a self-tuning system for Big Data analytics. In *Proceedings of the 5th Biennial Conference on Innovative Data Systems Research (CIDR'11)*, pages 261–272, Asilomar, CA, USA, 2011.

[61] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. The HiBench benchmark suite: characterization of the MapReduce-based data analysis. In *Proceedings of the 26th IEEE International Conference on Data Engineering Workshops (ICDEW'26)*, pages 41–51, Long Beach, CA, USA, 2010.

[62] Tim Hulsen, Saumya S. Jamuar, Alan R. Moody, Jason H. Karnes, Orsolya Varga, Stine Hedensted, Roberto Spreafico, David A. Hafler, and Eoin F. McKinney. From Big Data to precision medicine. *Frontiers in Medicine*, 6:34:1–34:14, 2019.

[63] Shadi Ibrahim, Tien-Dat Phan, Alexandra Carpen-Amarie, Houssem-Eddine Chihoub, Diana Moise, and Gabriel Antoniu. Governing energy consumption in Hadoop through CPU frequency scaling: an analysis. *Future Generation Computer Systems*, 54:219–232, 2016.

[64] Emir Imamagic and Dobrisa Dobrenic. Grid infrastructure monitoring system based on Nagios. In *Proceedings of the Workshop on Grid Monitoring (GMW'07)*, pages 23–28, Monterey Bay, CA, USA, 2007.

[65] Dingde Jiang, Yuqing Wang, Zhihan Lv, Sheng Qi, and Surjit Singh. Big Data analysis-based network behavior insight of cellular networks for industry 4.0 applications. *IEEE Transactions on Industrial Informatics*, 16(2):1310–1320, 2020.

[66] JsonLogic. `http://jsonlogic.com/`. Last visited: July 2020.

[67] Svilen Kanev, Juan P. Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. In *Proceedings of the 42nd ACM/IEEE International Symposium on Computer Architecture (ISCA'15)*, pages 158–169, Portland, OR, USA, 2015.

[68] Nakku Kim, Jungwook Cho, and Euiseong Seo. Energy-credit scheduler: an energy-aware virtual machine scheduler for cloud systems. *Future Generation Computer Systems*, 32:128–137, 2014.

[69] Aleksandar D. Kostic, Akinyemi I. Ojesina, Chandra Sekhar Pedamallu, Joonil Jung, Roel G.W. Verhaak, Gad Getz, and Matthew Meyerson. PathSeq: software to identify or discover microbes by deep sequencing of human tissue. *Nature Biotechnology*, 29(5):393–396, 2011.

[70] Eileen Kuehn, Max Fischer, Christopher Jung, Andreas Petzold, and Achim Streit. Monitoring data streams at process level in scientific Big Data batch clusters. In *Proceedings of the IEEE/ACM International Symposium on Big Data Computing (BDC'14)*, pages 90–95, London, United Kingdom, 2014.

[71] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. Twitter Heron: stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD'15)*, pages 239–250, Melbourne, Australia, 2015.

[72] Rasko Leinonen, Ruth Akhtar, Ewan Birney, Lawrence Bower, Ana Cerdeno-Tárraga, Ying Cheng, Iain Cleland, Nadeem Faruque, Neil Goodgame, Richard Gibson, Gemma Hoad, Mikyung Jang, Nima Pakseresht, Sheila Plaister, Rajesh Radhakrishnan, Kethi Reddy, Siamak Sobhany, Petra Ten Hoopen, Robert Vaughan, Vadim Zalunin, and Guy Cochrane. The European Nucleotide Archive. *Nucleic Acids Research*, 39:D28–D31, 2011.

[73] Ching-Chi Lin, Jian-Jia Chen, Pangfeng Liu, and Jan-Jan Wu. Energy-efficient core allocation and deployment for container-based virtualization. In *Proceedings of the 24th IEEE International Conference on Parallel and Distributed Systems (ICPADS 2018)*, pages 93–101, Sentosa, Singapore, 2018.

[74] Haiyan Liu, Ying Liu, and Jing Zheng. The application of Cacti in the campus network traffic monitoring. *Computer & Telecommunication*, 4:4, 2008.

[75] David Lo, Liqun Cheng, Rama Govindaraju, Luiz André Barroso, and Christos Kozyrakis. Towards energy proportionality for large-scale latency-critical workloads. In *Proceedings of the 41st ACM/IEEE International Symposium on Computer Architecture (ISCA'14)*, pages 301–312, Minneapolis, MN, USA, 2014.

[76] LXD - the Linux Container Daemon. `https://ubuntu.com/server/docs/containers-lxd`. Last visited: July 2020.

[77] Maciej Malawski, Adam Gajek, Adam Zima, Bartosz Balis, and Kamil Figiela. Serverless execution of scientific workflows: experiments with HyperFlow, AWS Lambda and Google Cloud Functions. *Future Generation Computer Systems*, 110:502–514, 2020.

[78] Stathis Maroulis, Nikos Zacheilas, and Vana Kalogeraki. ExpREsS: energy efficient scheduling of mixed stream and batch processing workloads. In *Proceedings of the 14th IEEE International Conference on Autonomic Computing (ICAC 2017)*, pages 27–32, Columbus, OH, USA, 2017.

[79] Matthew L. Massie, Brent N. Chun, and David E. Culler. The Ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817–840, 2004.

[80] Peter Membrey, Eelco Plugge, and Tim Hawkins. *The Definitive Guide to MongoDB: The NoSQL Database for Cloud and Desktop Computing*. Apress, 2010.

[81] Dirk Merkel. Docker: lightweight Linux containers for consistent development and deployment. *Linux Journal*, 239:76–91, 2014.

[82] Microsoft's Azure Functions service. `https://azure.microsoft.com/en-us/services/functions/`. Last visited: July 2020.

[83] NCBI. National Center for Biotechnology Information. `https://www.ncbi.nlm.nih.gov`. Last visited: July 2020.

[84] netatop: module for network per-process statistics. `https://www.atoptool.nl/netatop.php`. Last visited: July 2020.

[85] nethogs: Linux 'net top' tool. `https://github.com/raboof/nethogs`. Last visited: July 2020.

[86] Ashkan Paya and Dan C. Marinescu. Energy-aware load balancing and application scaling for the cloud ecosystem. *IEEE Transactions on Cloud Computing*, 5(1):15–27, 2015.

[87] Alfonso Pérez, Germán Moltó, Miguel Caballer, and Amanda Calatrava. Serverless computing for container-based architectures. *Future Generation Computer Systems*, 83:50–59, 2018.

[88] Sareh Fotuhi Piraghaj, Amir Vahid Dastjerdi, Rodrigo N. Calheiros, and Rajkumar Buyya. A framework and algorithm for energy efficient container consolidation in cloud data centers. In *Proceedings of the 2015 IEEE International Conference on Data Science and Data Intensive Systems (DSDIS 2015)*, pages 368–375, Sydney, Australia, 2015.

[89] Nicolas Poggi, David Carrera, Aaron Call, Sergio Mendoza, Yolanda Becerra, Jordi Torres, Eduard Ayguadé, Fabrizio Gagliardi, Jesús Labarta, Rob Reinauer, et al. ALOJA: a systematic study of Hadoop deployment variables to enable automated characterization of cost-effectiveness. In *Proceedings of the 2014 IEEE International Conference on Big Data (IEEE BigData 2014)*, pages 905–913, Washington DC, USA, 2014.

[90] Gang Ren, Eric Tune, Tipp Moseley, Yixin Shi, Silvius Rus, and Robert Hundt. Google-wide profiling: a continuous profiling infrastructure for data centers. *IEEE Micro*, 30(4):65–79, 2010.

[91] Eddie Antonio Santos, Carson McLean, Christopher Solinas, and Abram Hindle. How does Docker affect energy consumption? Evaluating workloads in and out of Docker containers. *Journal of Systems and Software*, 146:14–25, 2018.

[92] Neil Savage. Going serverless. *Communications of the ACM*, 61(2):15–16, 2018.

[93] Omar Sefraoui, Mohammed Aissaoui, and Mohsine Eleuldj. OpenStack: toward an open-source solution for cloud computing. *International Journal of Computer Applications*, 55(3):38–42, 2012.

[94] Rathijit Sen and David A. Wood. Energy-proportional computing: a new definition. *IEEE Computer*, 50(8):26–33, 2017.

[95] Juwei Shi, Yunjie Qiu, Umar Farooq Minhas, Limei Jiao, Chen Wang, Berthold Reinwald, and Fatma Özcan. Clash of the titans: MapReduce vs. Spark for large scale data analytics. *Proceedings of the VLDB Endowment*, 8(13):2110–2121, 2015.

[96] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop distributed file system. In *Proceedings of the 26th IEEE Symposium on Mass Storage Systems and Technologies (MSST'10)*, pages 1–10, Incline Village, NV, USA, 2010.

[97] Benoît Sigoure. OpenTSDB: a scalable, distributed time series database. In *O'Reilly Open Source Convention (OSCON'11)*, Portland, OR, USA, 2011.

[98] Simar Preet Singh, Anand Nayyar, Rajesh Kumar, and Anju Sharma. Fog computing: from architecture to edge computing and Big Data processing. *The Journal of Supercomputing*, 75(4):2070–2105, 2019.

[99] Georgios L. Stavrinides and Helen D. Karatza. An energy-efficient, QoS-aware and cost-effective scheduling approach for real-time workflow applications in

cloud computing systems utilizing DVFS and approximate computations. *Future Generation Computer Systems*, 96:216–226, 2019.

[100] Paul Tader. Server monitoring with Zabbix. *Linux Journal*, 195:72–78, 2010.

[101] The Apache Software Foundation. Apache Hadoop. `http://hadoop.apache.org/`. Last visited: July 2020.

[102] The Apache Software Foundation. HBase: a distributed database for large datasets. `https://hbase.apache.org/`. Last visited: July 2020.

[103] The Apache Software Foundation. OpenWhisk: Open source serverless Cloud platform. `https://openwhisk.apache.org/`. Last visited: July 2020.

[104] Erwin van Eyk, Alexandru Iosup, Simon Seif, and Markus Thömmes. The SPEC cloud group's research vision on FaaS and serverless architectures. In *Proceedings of the 2nd International Workshop on Serverless Computing (WoSC '17)*, pages 1–4, Las Vegas, NV, USA, 2017.

[105] Blesson Varghese and Rajkumar Buyya. Next generation cloud computing: new trends and research directions. *Future Generation Computer Systems*, 79:849–861, 2018.

[106] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing (SOCC'13)*, pages 5:1–5:16, Santa Clara, CA, USA, 2013.

[107] Jorge Veiga, Jonatan Enes, Roberto R. Expósito, and Juan Touriño. BDEv 3.0: energy efficiency and microarchitectural characterization of Big Data processing frameworks. *Future Generation Computer Systems*, 86:565–581, 2018.

[108] VisualVM website. `https://visualvm.github.io/`. Last visited: July 2020.

[109] Mark A. Walker, Chandra Sekhar Pedamallu, Akinyemi I. Ojesina, Susan Bullman, Ted Sharpe, Christopher W. Whelan, and Matthew Meyerson. GATK

PathSeq: a customizable computational tool for the discovery and identification of microbial sequences in libraries from eukaryotic hosts. *Bioinformatics*, 34(24):4287–4289, 2018.

[110] Md. Wasi-Ur-Rahman, Nusrat Sharmin Islam, Xiaoyi Lu, Dipti Shankar, and Dhabaleswar K. Panda. MR-Advisor: a comprehensive tuning tool for advising HPC users to accelerate MapReduce applications on supercomputers. In *Proceedings of the 28th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'16)*, pages 198–205, Los Angeles, CA, USA, 2016.

[111] Vincent M. Weaver. Linux perf_event features and overhead. In *Proceedings of the 2nd International Workshop on Performance Analysis of Workload Optimized Systems (FastPath'13)*, Austin, TX, USA, 2013.

[112] Tomasz Wiktor Wlodarczyk. Overview of time series storage and processing in a cloud environment. In *Proceedings of the 4th IEEE International Conference on Cloud Computing Technology and Science (CloudCom'12)*, pages 625–628, Taipei, Taiwan, 2012.

[113] Minxian Xu, Adel Nadjaran Toosi, and Rajkumar Buyya. iBrownout: an integrated approach for managing energy and brownout in container-based clouds. *IEEE Transactions on Sustainable Computing*, 4(1):53–66, 2018.

[114] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache Spark: a unified engine for Big Data processing. *Communications of the ACM*, 59(11):56–65, 2016.

Appendix A

Resumen extendido en castellano

Introducción

El análisis mediante Big Data ha estado en constante crecimiento y evolución en los últimos años hasta el punto de que ya no se le puede considerar como una sola área de conocimiento, sino como un conjunto de paradigmas, entornos y herramientas que se pueden aplicar a otras muchas áreas donde las Tecnologías de la Información también están presentes (v.gr., medicina, economía, industria, defensa) [62, 65]. Además, a medida que evolucionó, el Big Data también se integró en diferentes e importantes disciplinas, desde las ya asentadas Computación de Altas Prestaciones [11] e Inteligencia Artificial [40], a otras áreas más novedosas como la computación 'edge' o 'fog' [98], proporcionando en todo momento sus virtudes como la escalabilidad masiva o la alta capacidad para procesamiento de datos. Sin embargo, el Big Data no quedó intacto tras transitar por todas estas áreas, sino que evolucionó de forma significativa en términos tanto de los paradigmas sobre los que se asentaba como de la infraestructura subyacente sobre la que se desplegaba.

Considerando MapReduce [36] como el primer paradigma de procesamiento Big Data, implementado exitosamente por proyectos de código abierto como Apache Hadoop [101], los entornos de procesamiento actuales como Apache Spark [114] y Apache Flink [25] son considerablemente más versátiles. Esta evolución marcó un cambio importante en lo que respecta a los recursos típicamente utilizados por estos

entornos. Mientras que el paradigma MapReduce se basaba principalmente en el disco y la red, dado que los datos tenían que computarse en lotes y transmitirse a través del clúster, los nuevos entornos de procesamiento añaden la CPU y la memoria al conjunto de recursos utilizados de forma intensiva, considerando que ahora los datos se pueden procesar de forma más eficiente con estrategias 'in-memory', especialmente en cargas de trabajo iterativas [95]. Esto a su vez motiva que el consumo energético sea ahora también un aspecto importante a tener en cuenta, como ya lo es en otras áreas con trabajos con alta demanda de CPU como la Computación de Altas Prestaciones. Sin embargo, con estos nuevos entornos surgen también un mayor rango de aplicaciones y casos de uso posibles, lo que a su vez incrementa el grado de heterogeneidad en los recursos utilizados. Por ejemplo, un clúster Big Data puede estar ejecutando un algoritmo de aprendizaje máquina intensivo en CPU y, a continuación, un trabajo de procesamiento 'stream' intensivo en red, o incluso los dos al mismo tiempo. En lo que respecta a la infraestructura, las aplicaciones Big Data han dejado atrás a los clusters 'commodity', muy utilizados inicialmente, en favor de aproximaciones más flexibles. Actualmente, la mayoría de los usuarios despliegan sus aplicaciones usando plataformas consolidadas como el Cloud y tecnologías ya maduras como la virtualización, combinación con la cual se ofrecen despliegues más rápidos, bajo demanda y con la posibilidad de una facturación según el uso real (pago por uso). Adicionalmente, es posible beneficiarse de cualquier mejora que aparezca para estas tecnologías (v.gr., computación 'serverless', computación 'edge'), ofreciendo de esta forma un grado adicional de flexibilidad y, en algunos casos, de eficiencia.

Por desgracia, si algo caracteriza a las aplicaciones Big Data es el potencial gran tamaño de la infraestructura que necesitan, siendo esta escalable de forma directamente proporcional al tamaño de los datos a procesar y/o al límite temporal deseado. Si los despliegues se hacen en una plataforma Cloud puede ser necesario un gran número de instancias virtuales. A medida que aumenta el tamaño de la infraestructura, gestionar su monitorización puede resultar cada vez más difícil, hasta el punto de tener que sacrificar el detalle en favor de un resumen general, o directamente también puede ser difícil la gestión activa de los recursos utilizados. Esto último puede ocasionar mayores costes debido a la infrautilización de los recursos.

Esta Tesis explora y presenta varias ideas relacionadas que, conjuntamente, bus-

can mejorar la gestión de las aplicaciones Big Data en lo que respecta al uso de los recursos y al perfilado del código, todo ello enfocado específicamente en el soporte de entornos virtuales basados en contenedores dada su gran popularidad [26]. Las ideas principales se basan en premisas respaldadas por tecnologías y paradigmas que, aunque bien asentadas y probadas, no han sido extensamente implementadas para Big Data o para entornos basados en contenedores. Las principales contribuciones de esta Tesis se han propuesto mediante el desarrollo de dos entornos disponibles públicamente, y que además se han probado empíricamente con varios experimentos y escenarios representativos. Lo que es más importante, ambos entornos se han diseñado basándose en los siguientes conceptos nucleares: 1) ser lo más simple posible en lo que respecta a sus interfaces de usuario; 2) ser capaces de trabajar en tiempo real, sin tener que reiniciar ninguna aplicación y/o herramienta; y 3) ser lo menos intrusivo posible, buscando tener bajo impacto y en general trabajar de forma transparente con los entornos de procesamiento Big Data y sus aplicaciones.

El primer entorno propuesto, BDWatchdog, se centra en ofrecer un análisis exhaustivo de las aplicaciones mediante una combinación de la monitorización de los recursos y el perfilado del código. En lo que respecta a la monitorización, se usan series temporales enfocadas a procesos individuales, en vez de al sistema completo. Esta monitorización de grano fino, en combinación con operaciones de filtrado y agregación soportadas por las bases de datos de series temporales, ofrece múltiples opciones para analizar una tarea en particular, desde las variaciones en el uso de recursos entre diversos nodos (v.gr., maestro y esclavos), a la contabilidad de un mismo proceso ejecutándose en todo el clúster (v.gr., ejecutores Spark), o incluso para aislar componentes específicos de una tarea (v.gr., brokers y consumidores en una aplicación 'streaming'). En cuanto al perfilado del código, BDWatchdog implementa un muestreo a bajo nivel de la pila directamente desde el kernel del sistema operativo, sin necesidad de instrumentar, interrumpir o alterar la aplicación en ningún momento. Adicionalmente, este tipo de perfilado se realiza de forma transparente a las aplicaciones, lo que a su vez permite que pueda iniciarse o detenerse en cualquier momento, ofreciendo además todos los datos recogidos de forma inmediata y en tiempo real. La combinación de ambas funcionalidades convierte a BDWatchdog en una piedra angular para otras herramientas de más alto nivel que requieran información del uso de recursos por parte de las aplicaciones de forma precisa y en tiempo real, sin importar además la infraestructura subyacente en la que se despliegan.

El segundo entorno propuesto tiene como objetivo proporcionar un reescalado automático y en tiempo real de las aplicaciones Big Data. Concretamente, este entorno explora cómo ejecutar dichas aplicaciones en clusters de contenedores mediante diferentes políticas de ajuste de los recursos asignados. Nuestra aproximación implementa una política de escalado acorde al uso real de los recursos en términos de CPU y memoria, creando de esta forma un entorno serverless. Las plataformas serverless actuales están diseñadas principalmente para ejecutar scripts simples que son agnósticos de la infraestructura subyacente, mientras que al mismo tiempo poseen el potencial de escalar su demanda de recursos según sus necesidades. De esta forma, en un escenario serverless el usuario no tiene que especificar ninguna configuración inicial de recursos, garantizando de esta manera un cierto grado de auto-configuración y auto-gestión. Adicionalmente, dichos recursos se facturan teniendo en cuenta solo aquellos que realmente se han utilizado, y no los recursos asignados como ocurre en los entornos Cloud tradicionales como los de Infraestructura como Servicio (IaaS). Por desgracia, las plataformas serverless existentes aplican reglas estrictas en lo que se refiere al tipo de aplicaciones soportadas, lo que a menudo excluye cualquier tarea que requiera de entornos más complejos como los de las instancias virtuales (i.e., un sistema operativo, librerías de terceros y ficheros de configuración). A fin de preservar la flexibilidad y los beneficios del paradigma serverless, el entorno propuesto en esta Tesis proporciona una plataforma que despliega contenedores que se comportan como instancias tradicionales, pero que al mismo tiempo siguen una política de asignación de recursos acorde a la filosofía serverless. Para lograr este objetivo, nuestro entorno se construye sobre BDWatchdog, usando sus capacidades de monitorización de grano fino para poder implementar una gestión activa de los recursos asignados a un conjunto de contenedores utilizados para ejecutar una tarea Big Data en particular.

Finalmente, esta Tesis presenta un escenario práctico donde ambos entornos trabajan conjuntamente para gestionar la energía como si se tratase de otro recurso del sistema igual que la CPU y la memoria. Para proporcionar esta novedosa gestión energética, se crea una nueva política de ajuste para el entorno de reescalado a fin de implementar el concepto de límite energético o presupuesto de potencia. La idea de presupuesto de potencia se puede definir como un límite energético que puede imponerse sobre una entidad y que se aplica a lo largo del tiempo. Más concretamente, en este último escenario se han modelado diferentes usuarios y sus aplicaciones

aplicándoles presupuestos de potencia variables, considerando las aplicaciones como grupos de contenedores. Para poder implementar la limitación energética a los contenedores, se define una regulación de la CPU combinada con políticas configurables. Es también interesante destacar que esta gestión energética se ha implementado completamente a nivel de software, desde la medición de la energía consumida a la limitación de la misma.

Objetivos y Metodología de Trabajo

Los principales objetivos de esta Tesis, así como sus subobjetivos, se listan a continuación.

1. Diseño e implementación de un entorno para el análisis exhaustivo de cargas de trabajo Big Data.
    - Soporte eficiente para entornos basados en contenedores
    - Monitorización de grano fino y a nivel de proceso
    - Perfilado de código a bajo nivel usando diagramas de llamas ('flame graphs')
    - Bajo impacto en el rendimiento y soporte para funcionar en tiempo real
2. Desarrollo de un entorno de reescalado de recursos para aplicaciones Big Data desplegadas en clusters de contenedores.
    - Reescalado de recursos de forma automática y en tiempo real
    - Alto grado de respuesta y adaptación a fin de minimizar el impacto en el rendimiento
    - El diseño debe permitir el soporte de múltiples políticas de reescalado
3. Implementación de una política específica para el entorno de reescalado que proporcione funcionalidades de limitación energética y presupuesto de potencia.
    - El presupuesto debe soportar límites energéticos estáticos y dinámicos

- El presupuesto se tiene que poder aplicar tanto a usuarios como a aplicaciones

Los dos primeros objetivos se abordaron mediante el desarrollo de dos entornos diseñados específicamente para implementar todas las funcionalidades requeridas por cada subobjetivo. Por un lado, el primer entorno se centra en un punto de vista pasivo donde los recursos del sistema y de las aplicaciones se analizan para extraer información útil y lo más detallada posible, siempre con el objetivo de trabajar en tiempo real y con el mínimo impacto posible en el rendimiento. Para este objetivo, primero fue necesario adquirir un sólido conocimiento previo sobre cómo desplegar clusters virtuales basados en contenedores sobre los que poder ejecutar aplicaciones Big Data de forma adecuada y eficiente. Para este propósito, se desarrolló una arquitectura de Plataforma como Servicio (PaaS). Por otro lado, el segundo entorno actúa de una forma más activa al gestionar y modificar de forma dinámica el contexto en el que se ejecutan las aplicaciones Big Data. En este caso hay una necesidad de analizar cómo reaccionan las aplicaciones cuando su contexto de ejecución subyacente está siendo constantemente modificado en términos de los recursos disponibles (v.gr., CPU, memoria). En general, ambos entornos se desarrollaron con una metodología ligeramente ágil donde las funcionalidades interesantes eran descritas, diseñadas, implementadas y mejoradas sobre la marcha, con el fin de obtener bancos de pruebas experimentales para ser posteriormente evaluados. Esta metodología fue posible en parte gracias a que todas las tecnologías subyacentes utilizadas, como las bases de datos de series temporales o NoSQL, y los lenguajes de programación de alto nivel como Python, permiten en general un prototipado rápido. Finalmente, para el tercer objetivo, los dos entornos previamente mencionados se combinaron y extendieron para integrarse con una herramienta externa a fin de poder trabajar con la energía como otro recurso adicional.

Cabe resaltar que los tres objetivos concluyeron en sus últimas etapas en herramientas utilizables disponibles públicamente. Además, estas herramientas se han evaluado extensamente con la ejecución de cargas de trabajo Big Data representativas y desplegadas en clusters de contenedores, con el objetivo de obtener todos los resultados experimentales presentados en esta Tesis.

Conclusiones

Actualmente, las aplicaciones Big Data están creciendo en tamaño y complejidad, tanto en términos de los recursos que necesitan como de la infraestructura subyacente más comúnmente utilizada para su despliegue, marcando una clara evolución. Por un lado, el paradigma MapReduce se implementó usando el procesamiento por lotes, el cual dependía fuertemente de recursos como el disco y la red, desplegándose principalmente en clusters 'commodity'. Por otro lado, los motores de procesamiento del estado del arte (v.gr., Spark, Flink) soportan también procesamiento 'in-memory' y 'stream', involucrando para ello a un mayor número de recursos y más heterogéneo, desde el disco y la red hasta la memoria, la CPU y, por ende, la energía. Además, estos entornos Big Data más flexibles se despliegan a menudo en clusters virtuales y plataformas Cloud. Esta Tesis ha explorado esta evolución para definir y proponer nuevas maneras para el análisis de recursos y el ajuste de cargas de trabajo Big Data cuando estas se despliegan en clusters virtuales basados en contenedores. Considerando esto como un doble objetivo en lo que respecta a los recursos, con el análisis pasivo como primer objetivo y el ajuste activo como segundo, esta Tesis ha presentado en detalle dos entornos que abordan cada objetivo de forma independiente, así como un escenario adicional que analiza de forma específica la gestión del consumo energético. Este último escenario es resultado de la combinación y extensión de los dos entornos previos con otra herramienta externa.

El primer entorno propuesto, BDWatchdog, se ha diseñado específicamente para proporcionar una monitorización de grano fino de los recursos para aplicaciones Big Data mediante series temporales, así como un perfilado del código de forma transparente mediante el muestreo de la pila de llamadas del programa. Por un lado, la funcionalidad de monitorización es capaz de trabajar a nivel de proceso, permitiendo así una recogida de métricas continua y próxima al tiempo real, siendo capaz de ofrecerlas inmediatamente después usando filtros y agregaciones. Es interesante mencionar que esta monitorización a nivel de proceso permite un soporte inherente de nuevas tecnologías de virtualización como los contenedores. Por otro lado, la funcionalidad de perfilado permite extender el análisis de las aplicaciones, adquiriendo de esta manera una visión más profunda sobre cómo dichas aplicaciones están usando los recursos asignados. Se han estudiado varios casos de uso a través de una extensa experimentación con BDWatchdog, demostrando cómo la combinación

de estas funcionalidades puede ofrecer un informe detallado a los usuarios sobre el comportamiento de su aplicación, incluso en tiempo real. Adicionalmente, los experimentos han demostrado que este entorno es capaz de escalar con el tamaño de la aplicación al mismo tiempo que supone un bajo impacto en su rendimiento.

La segunda propuesta de esta Tesis se asienta sobre BDWatchdog para crear un entorno de reescalado para contenedores donde los recursos no solo se monitorizan sino que también se modifican para fijar límites acorde a diferentes políticas de ajuste o diferentes casos de uso. Una de estas políticas ofrece un entorno serverless donde la CPU y la memoria se reescalan de acuerdo al uso real de las aplicaciones, modificando adecuadamente los límites de los recursos de sus contenedores subyacentes. Este entorno ha demostrado que es posible construir una plataforma serverless con tecnologías de virtualización más flexibles como los contenedores y que, además, es posible ejecutar aplicaciones Big Data en dicha plataforma con un impacto asumible. Además de ofrecer al usuario las ventajas del paradigma serverless, otro gran beneficio del entorno de reescalado propuesto es un mayor grado de utilización de los recursos. Por ejemplo, se ha obtenido un incremento del 77 % en la ejecución de una tarea de procesamiento 'stream', considerando que es fácilmente adaptable al paradigma serverless gracias a su relativa estabilidad en el uso de recursos y a que no presenta una gran variabilidad a lo largo del tiempo.

Finalmente, esta Tesis ha presentado un escenario adicional que combina los dos entornos previos con una herramienta de terceros (PowerAPI) a fin de poder gestionar la energía en clusters de contenedores. Al implementar una política de ajuste alternativa para el entorno de reescalado, la energía puede gestionarse como otro recurso del sistema de igual forma que la CPU, permitiendo así implementar funcionalidades de limitación energética y presupuesto de potencia. Este escenario ha mostrado que estas funcionalidades permiten desplegar casos de uso interesantes para Big Data. Por un lado, la limitación de energía permite controlar de forma efectiva la energía consumida por una aplicación o usuario sin causar apenas impacto en los recursos de CPU que requiere. Por otro lado, el presupuesto de potencia permite compartir y distribuir una cuota de energía entre aplicaciones y/o usuarios, ya sea según unos límites deseados o para adaptarse dinámicamente a demandas energéticas variables.

Trabajo Futuro

Los entornos presentados en esta Tesis se han diseñado para poder extenderse o adaptarse fácilmente, además de para poder incorporar herramientas nuevas o alternativas para su funcionalidad. De hecho, el escenario energético descrito es un claro ejemplo de dicha flexibilidad al adaptar e integrar los dos entornos previamente propuestos con la herramienta PowerAPI para explorar un nuevo caso de uso enfocado en gestión energética. No obstante, aún es posible extenderlos más.

Para BDWatchdog sería de gran interés mejorar sus capacidades de perfilado para que otros recursos como el disco, la memoria o la red también pudieran analizarse. Estas funcionalidades podrían seguir el mismo diseño utilizado para el perfilado de la CPU, es decir, el muestreo a bajo nivel de métricas de rendimiento que luego se usarían para crear diagramas de llamas. La viabilidad de este perfilado se ha demostrado gracias a nuevas tecnologías como eBPF y el conjunto de scripts de BCC [21]. Adicionalmente, las series temporales de recursos proporcionadas por BDWatchdog son útiles para otros casos de uso como la clasificación de trabajos en base a sus patrones de consumo o para la detección de anomalías, usando en ambos casos las series temporales como entrada de datos para algoritmos de aprendizaje máquina.

Sobre el entorno de reescalado de recursos, cabe mencionar que actualmente solo se permite el escalado vertical, es decir, solo los recursos de los contenedores que alojan una aplicación se escalan en base a diversas políticas de ajuste como la del paradigma serverless. Aunque esta funcionalidad es capaz de lidiar con escenarios de alta variabilidad al permitir un ajuste de grano fino, sería interesante el soporte de un escalado horizontal. Con esta funcionalidad, el entorno sería capaz de actuar en casos extremos donde hubiera un uso de recursos especialmente bajo o alto, ya fuera retirando o añadiendo instancias, respectivamente. Además, otros recursos como el disco y la red también pueden escalarse. El entorno propuesto lo soporta de forma inherente, ya que dichos recursos también pueden gestionarse con mecanismos apropiados como cgroups para el disco y 'Linux Traffic Control' para la red. Finalmente, el impacto causado en el rendimiento podría mitigarse todavía más si el entorno y las aplicaciones intercambiaran información para ayudar en la política de escalado, considerando que actualmente el entorno gestiona los contenedores como

cajas negras.

Por último, en lo que respecta a limitación energética y presupuesto de potencia, algunos casos de uso interesantes podrían explorarse con pequeñas extensiones y adaptaciones de la actual plataforma. Por ejemplo, el concepto de presupuesto de potencia podría extenderse a otros recursos para así crear presupuestos de CPU o de memoria. Con estos presupuestos, el recurso base se particionaría y compartiría entre las aplicaciones de un usuario o entre los contenedores de una aplicación. El concepto de restricción de la CPU para limitar el consumo energético podría modificarse ligeramente cambiando energía por temperatura, creando de esta manera un nuevo escenario en el cual el objetivo es controlar la temperatura para que esta no exceda un límite definido. Este escenario es útil como una alternativa software a la limitación de CPU ya existente en la mayoría del hardware actual, la cual es menos flexible en general y su uso se restringe a escenarios en los que la temperatura ya ha subido de forma considerable sobrepasando un cierto umbral.

Principales Contribuciones

Las contribuciones originales derivadas de esta Tesis son las siguientes:

- Desarrollo de una arquitectura PaaS para el despliegue de aplicaciones Big Data en clusters de contenedores. Estos contenedores ofrecen un entorno completamente funcional que imita una máquina virtual, beneficiándose no obstante de despliegues más rápidos y de una menor penalización en la virtualización de recursos [41].

- Diseño e implementación del entorno BDWatchdog para el análisis exhaustivo de aplicaciones Big Data usando la monitorización de recursos y el perfilado del código fuente, ambas en tiempo real [42].

- Desarrollo de un entorno para proporcionar un contexto 'serverless' para contenedores. En este contexto los recursos se escalan automáticamente de forma acorde a su uso real [43].

- Despliegue y evaluación de un escenario específico para la gestión de límites

energéticos y presupuestos de potencia, tanto para usuarios como para aplicaciones [44].

Software desarrollado

Se han desarrollado las siguientes herramientas software en la Tesis, estando públicamente disponibles:

- BDWatchdog: entorno para la monitorización y el perfilado de aplicaciones Big Data en tiempo real. Disponible en `http://bdwatchdog.dec.udc.es`.
- Plataforma para el escalado de recursos en clusters de contenedores en tiempo real siguiendo el paradigma serverless. Disponible en `http://bdwatchdog.dec.udc.es/serverless`.
- Especificaciones e información adicional sobre el escenario experimental de limitación energética. Disponible en `http://bdwatchdog.dec.udc.es/energy`.

Software registrado

Se han registrado dos productos software en el Registro de la Propiedad Intelectual como resultado de esta Tesis:

- Jonatan Enes, Roberto R. Expósito, and Juan Touriño. BDWatchdog: herramienta para la monitorización y perfilado de trabajos Big Data en tiempo real, Octubre 2019. Número de asiento registral: 03/2020/180. Entidad titular: Universidade da Coruña. País de prioridad: España. En explotación por Torus Software Solutions S.L. a través del contrato INV11419 desde el 01/11/2019.
- Jonatan Enes, Roberto R. Expósito, and Juan Touriño. Plataforma de escalado de recursos en tiempo real para contenedores en entornos serverless, Julio 2020. Número de asiento registral: pendiente. Entidad titular: Universidade da Coruña. País de prioridad: España.

Publicaciones de la Tesis

Artículos en revistas

- Jonatan Enes, Javier López Cacheiro, Roberto R. Expósito, and Juan Touriño. Big Data-oriented PaaS architecture with disk-as-a-resource capability and container-based virtualization. *Journal of Grid Computing*, 16(4):587–605, 2018. JCR Q1.
- Jorge Veiga, Jonatan Enes, Roberto R. Expósito, and Juan Touriño. BDEv 3.0: energy efficiency and microarchitectural characterization of Big Data processing frameworks. *Future Generation Computer Systems*, 86:565–581, 2018. JCR Q1 (primer decil).
- Jonatan Enes, Roberto R. Expósito, and Juan Touriño. BDWatchdog: real-time monitoring and profiling of Big Data applications and frameworks. *Future Generation Computer Systems*, 87:420–437, 2018. JCR Q1 (primer decil).
- Jonatan Enes, Roberto R. Expósito, and Juan Touriño. Real-time resource scaling platform for Big Data workloads on serverless environments. *Future Generation Computer Systems*, 105:361–379, 2020. JCR Q1 (primer decil).
- Jonatan Enes, Roberto R. Expósito, Javier López Cacheiro, and Juan Touriño. Anomaly detection in HPC systems through job classification using time series and machine learning. 2020. (Enviado para publicación en revista).

Congresos internacionales

- Jonatan Enes, Guillaume Fieni, Roberto R. Expósito, Romain Rouvoy, and Juan Touriño. Power budgeting of Big Data applications in container-based clusters. 2020. (Enviado para publicación en congreso).

Otras publicaciones menores

- Jonatan Enes, Roberto R. Expósito, and Juan Touriño. Towards smart and automatic optimization for Big Data: real-time application monitoring and profiling. In *3rd NESUS Winter School & PhD Symposium*. Zagreb, Croatia, 2018.