BDWatchdog: real-time monitoring and profiling of Big Data applications and frameworks

Jonatan Enes*, Roberto R. Expósito, Juan Touriño

Computer Architecture Group, Universidade da Coruña, Campus de A Coruña, 15701 A Coruña, Spain

Abstract

Current Big Data applications are characterized by a heavy use of system resources (e.g., CPU, disk) generally distributed across a cluster. To effectively improve their performance there is a critical need for an accurate analysis of both Big Data workloads and frameworks. This means to fully understand how the system resources are being used in order to identify potential bottlenecks, from resource to code bottlenecks. This paper presents BDWatchdog, a novel framework that allows real-time and scalable analysis of Big Data applications by combining time series for resource monitorization and flame graphs for code profiling, focusing on the processes that make up the workload rather than the underlying instances on which they are executed. This shift from the traditional system-based monitorization to a process-based analysis is interesting for new paradigms such as software containers or serverless computing, where the focus is put on applications and not on instances. BDWatchdog has been evaluated on a Big Data cloud-based service deployed at the CESGA supercomputing center. The experimental results show that a process-based analysis allows for a more effective visualization and overall improves the understanding of Big Data workloads. BDWatchdog is publicly available at `http://bdwatchdog.dec.udc.es`.

Keywords: Big Data, monitoring, profiling, time series, flame graphs, process-based analysis

1. Introduction

In recent years, the use of Big Data frameworks and their resource demands are increasing at an astounding rate, both from the business world as well as from the research front. With the appearance of distributed processing frameworks such as Apache Hadoop [1] and subsequent evolutions like Apache Spark [2], Big Data workloads have evolved and adapted to the users' needs and use cases as well as to the underlying hardware infrastructure that supports them. Nevertheless, it is increasingly well established that Big Data workloads are characterized by a heavy use of different system resources (i.e., CPU, memory, disk and network) in one way or another, and thus in order to increase their efficiency several improvements are possible.

Much research has already been conducted on analyzing Big Data workloads and frameworks in order to either minimize their resource consumption and optimize them as a whole [3, 4, 5, 6] or to increase infrastructure consolidation and efficiency if cloud computing or hypervisor-based virtualization technologies are used [7, 8]. However, when it comes to a more fine grain level of system monitorization, there are no widespread solutions capable of targeting isolated processes (i.e., being a process an instance of a running application on an operating system), and in the end this is disregarded in favor of overall instance resource usage. Nevertheless, this step forward is especially important when considering emerging technologies like operating-system-level virtualization based on software containers (e.g., Docker [9]) or new compelling cloud-based models like serverless computing [10]. Both technologies use environments where applications are no

*Corresponding author. Tel.: +34 881 011 212; Fax: +34 981 167 160

Email addresses: `jonatan.enes@udc.es` (Jonatan Enes), `rreye@udc.es` (Roberto R. Expósito), `juan@udc.es` (Juan Touriño)

longer executed on fully provisioned and isolated instances, but rather on software containers (i.e., system-wide monitoring no longer applies). Furthermore, the efficiency and performance of Big Data frameworks is taken for granted or left aside as a developer's concern, treating them as black boxes. All the efforts to increase efficiency are usually pushed towards improving the environment where applications and frameworks are executed on, rather than on analyzing such frameworks for potential improvements.

This paper brings forward BDWatchdog, a novel framework to visually characterize the performance of Big Data workloads and frameworks, focusing the analysis on processes and individual applications rather than on entire instances. BDWatchdog mainly consists of two tools with specialized purposes: resource monitoring using time series and source code profiling with flame graphs, both being able to be used individually or combined, in real time and for any type of Big Data workload whether it is batch or streaming. On the one hand, using time series, system resources (e.g., CPU, disk) can be visually displayed over time using line plots, allowing for easy detection of local bottlenecks and patterns. Moreover, by properly storing and tagging the raw data it is also possible to later perform richer queries using selection and aggregation operations, allowing for example to plot the accumulated usage of a certain system resource by a single application executed across a cluster. With flame graphs [11], on the other hand, it is possible to visually identify source code bottlenecks or simply characterize the behaviour of a framework or a workload in terms of what parts of the code are most executed. This type of profiling is interesting because it uses a low-level kernel profiler rather than code instrumentation, so there is no need to alter neither the code nor the environment whatsoever. Furthermore, the visual approach based on flame graphs offers a better visualization experience, showing on the same graph code hot spots and allowing to mix operating system and Java Virtual Machine (JVM) stacks. Overall, this transparent profiling supporting the JVM makes BDWatchdog especially interesting to be used in Big Data scenarios, where JVM-based languages (e.g., Scala, Java) are predominant.

The overall deployed architecture and the tools developed are specifically designed to be scalable in order to work on a real Big Data scenario as well as to remain nonintrusive. The proposed framework has been evaluated using both batch and streaming workloads on a real Big Data infrastructure built using Docker containers and deployed at the Galicia Supercomputing Center (CESGA) [12].

The rest of this paper is organized as follows: Section 2 presents the concepts, terminology and describes the overall use case scenario. Section 3 describes the current technologies and their limitations, as well as the proposed solutions to overcome them. In Section 4, 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 5. Real-world use case scenarios where BDWatchdog is most interestingly applied are described in Section 6. Finally, Section 7 summarizes the results and reviews the main use cases.

2. Technical foundations

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. Big Data frameworks as groups of processes

Currently, Apache Hadoop [1] 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) [13] and HDFS (Hadoop Distributed File System) [14], 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 different JVMs. 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 [15] or Apache Spark [2] 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 [16].

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. This work 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.

2.2. Software containers and their monitorization

Until now we have referred to "instances" as the underlying minimal and independent infrastructure component used to host Big Data frameworks and applications. 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 [17, 18] 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 (e.g., AWS Lambda [19]), 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, this work 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, virtual machines and ultimately to software containers.

2.3. 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* [20] 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 [21, 22]. Currently, BDWatchdog only focuses on *perf* and its powerful profiling capabilities.

3. Related work

There are currently many solutions to provide resource usage metrics and/or application profiling from different fields such as High Performance Computing (HPC) or the cloud and the datacenter. For HPC applications there are several tools that can monitor and profile parallel programs, mainly those applications using the Message Passing Interface (MPI). These tools use sampling to provide monitorization of hardware counters and other events, and code instrumentation for profiling, all of this after the application finishes. Some examples of such tools are Extrae [23] and Score-P [24] for data collection, and Paraver [25] and Vampir [26] for visualization. However, although similar in design to our solution, most HPC tools are designed with paradigms such as MPI in mind. So these tools focus on improving performance by analyzing how the MPI routines are executed and how the communications occur between MPI processes. On the other hand, our solution focuses on operating system processes and their resource usage monitorization, as well as on JVM profiling of the applications, both in real time. In other scenarios like the cloud or datacenters, 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). Examples of such tools are given in the following subsections considering that Big Data architectures are typically deployed on scenarios closer to datacenter or cloud infrastructures than to HPC ones. As previously stated, 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 [27]. 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 [4], ALOJA [5] or MR-Advisor [6] 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.

3.1. Infrastructure monitoring

One of the first monitoring solutions that is still widely used is the Simple Network Management Protocol (SNMP) [28], 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 [29] or Zabbix [30] 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 [31] or Nagios [32] 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 datacenters 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 RDD, 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 [33], 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 [34] 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.

3.2. 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* [35] 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* [36], already considered in other works for very similar purposes [27], and the *netatop* module [37], 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.

3.3. 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 [38] or JProfiler [39], 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 [20] 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 [11] 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 or aggregated.

3.4. 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.

4. BDWatchdog: architecture design and implementation

The framework proposed in this paper is focused 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 and thus they are independently explained in Sections 4.1 and 4.2 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 4.3, 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.

4.1. BDWatchdog monitoring tool

To create a fully scalable monitoring system that can be used across many instances as well as allowing dynamically resizing 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 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

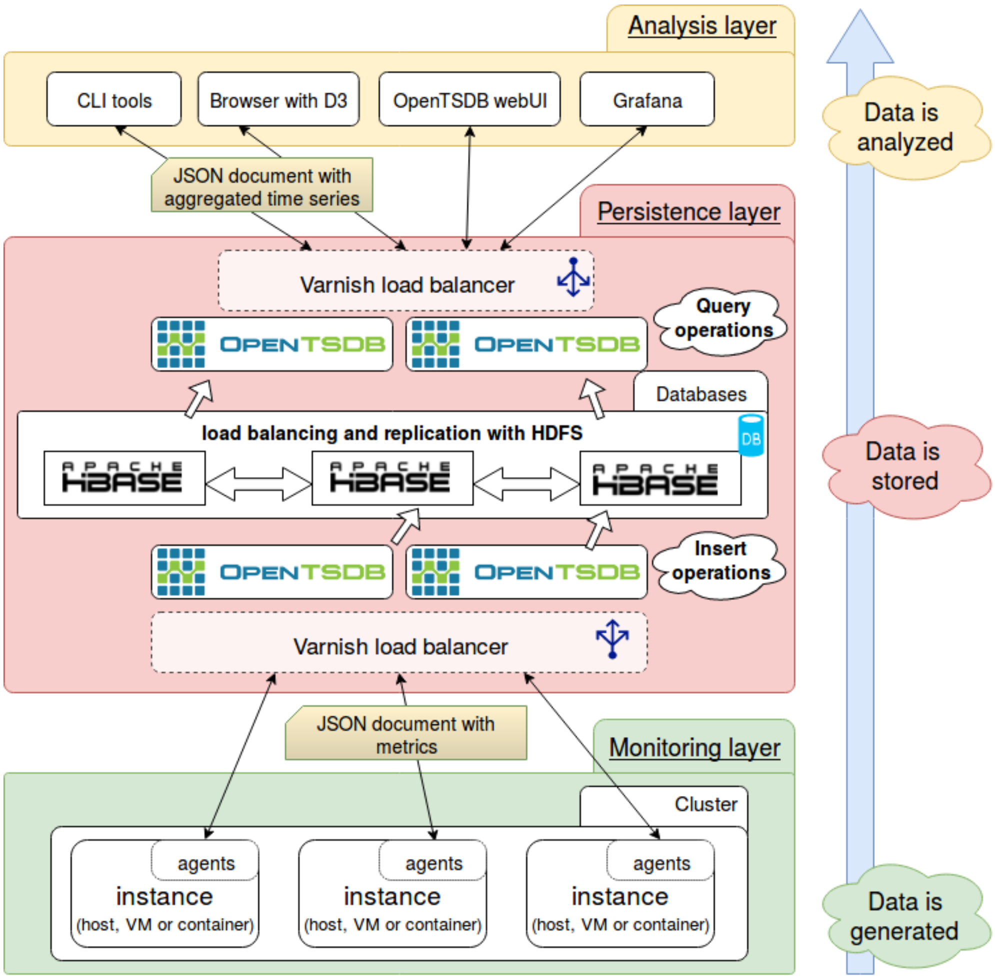


Figure 1: BDWatchdog monitoring architecture

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.

In order to understand how the monitoring data are created, stored and then analyzed, it is important to explain how these data are structured and defined in the first place, as described in Section 4.1.1. More details are then given regarding the implementation of the different, individual layers of the architecture in Sections 4.1.2, 4.1.3 and 4.1.4 for the monitoring, persistence and analysis layers, respectively.

4.1.1. 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 create a

```
{
        'metric':'proc.mem.virtual',
        'timestamp':'1491472173',
        'value':'133.15',
        'tags':{
                'host':'host3',
                'pid':'907',
                'command':'(python)'
        }
}
```

Listing 1: Definition of a monitoring metric point value as a JSON document in BDWatchdog

Table 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 (PID)											
		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

metric all these fields are mandatory (see Listing 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 [40], 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 1 for an example of several values for a metric in the form of a cube). All the system-wide and per-process metrics and tags supported by BDWatchdog, as well as their associations, are shown in Table 2.

However, instead of using a relational SQL database 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 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 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).

4.1.2. Time series monitoring layer

As mentioned in Section 3.2, 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. Nevertheless, these raw data generated by the agents have to be processed to fit the metric structure as defined in Section 4.1.1. So, a lightweight Python-based processing pipeline has been coupled to the agents (see Figure 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 PID to the actual Java main class that is running in that JVM (e.g., a NodeManager in the case of YARN).

Table 2: System-wide and per-process metrics with their associated tags supported by BDWatchdog

			Tags					
			host name	process name	PID	core id	disk	interface
system	cpu	sys.cpu.user	X			X		
		sys.cpu.kernel	X			X		
		sys.cpu.idle	X			X		
		sys.cpu.wait	X			X		
		sys.cpu.usage	X			X		
	mem	sys.mem.free	X					
		sys.mem.usage	X					
		sys.swap.free	X					
	disk	sys.disk.read.mb	X				X	
		sys.disk.read.ios	X				X	
		sys.disk.write.mb	X				X	
		sys.disk.write.ios	X				X	
		sys.disk.usage	X				X	
	net	sys.net.in.mb	X					X
		sys.net.out.mb	X					X
		sys.net.usage	X					X
	power	sys.power	X					
		sys.cpu.energy	X			X		
	temp	sys.cpu.temp	X			X		
process	cpu	proc.cpu.user	X	X	X			
		proc.cpu.kernel	X	X	X			
	mem	proc.mem.resident	X	X	X			
		proc.mem.virtual	X	X	X			
		proc.mem.swap	X	X	X			
	disk	proc.disk.reads.mb	X	X	X			
		proc.disk.writes.mb	X	X	X			
	net	proc.net.tcp.in.mb	X	X	X			
		proc.net.tcp.in.packets	X	X	X			
		proc.net.tcp.out.mb	X	X	X			
		proc.net.tcp.out.packets	X	X	X			

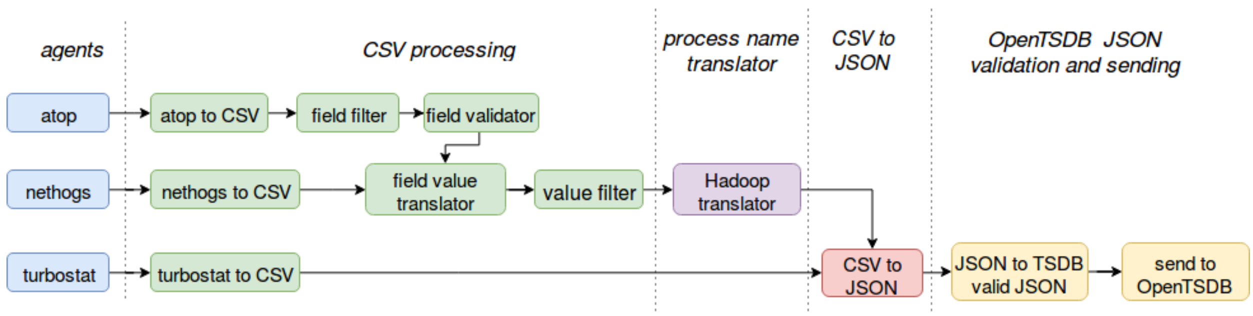


Figure 2: Processing pipeline of the monitoring tool

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. 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 Table 2). 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.

4.1.3. 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 1, this layer mainly consists of: 1) a non-relational and scalable Hadoop database (Apache HBase [41]); 2) OpenTSDB [42, 43] 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 in Section 4.1.2.

4.1.4. 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) [44] 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 4.3. In addition, several scripts are provided to plot time series for a specific time interval as image files from a Command Line Interface (CLI).

4.2. BDWatchdog 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 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 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.

4.2.1. 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* [20], 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.

4.2.2. 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 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 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

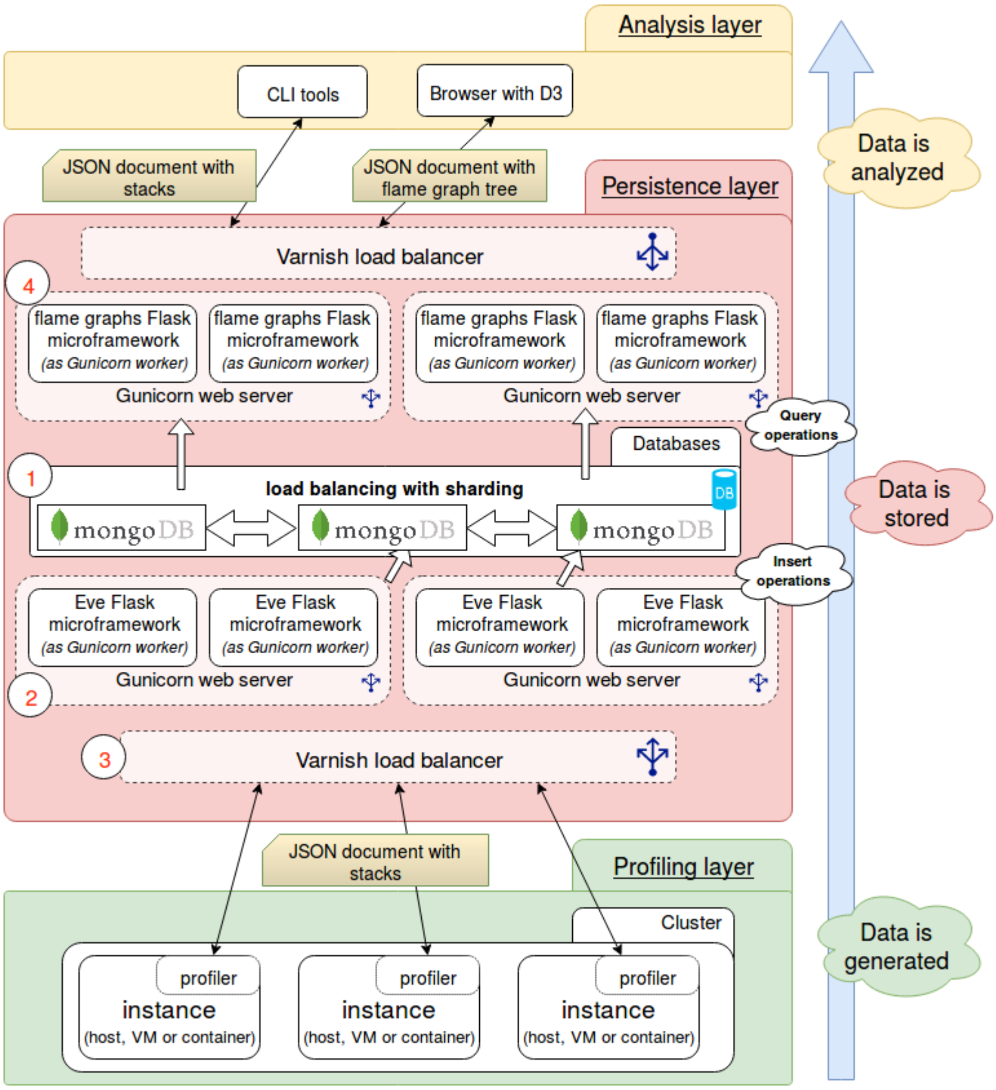


Figure 3: BDWatchdog profiling architecture

on-demand manner. In this way, later analyses 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.

4.2.3. Profiling data persistence layer

Using a similar architecture as the monitoring tool and taking scalability into consideration, MongoDB [45] has been chosen as the database to persist the profiling data (see label 1 in Figure 3). MongoDB is a popular document database that can be used to effectively store and retrieve a high number of JSON

```
{
‘timestamp’: 1495813324,
‘hostname’: ‘hadoop0’,
‘stack’: ‘java-18714;start_thread;java_start;JavaThread::run;
JavaThread::thread_main_inner;attach_listener_thread_entry;
JvmtiExport::load_agent_library;Agent_OnAttach;jvmti_GenerateEvents;
JvmtiCodeBlobEvents::generate_compiled_method_load_events;
JvmtiExport::post_compiled_method_load;cbCompiledMethodLoad;
generate_single_entry;sig_string;jvmti_GetMethodName;
Method::checked_resolve_jmethod_id’,
‘value’: 6
}
```

Listing 2: Definition of a profiled stack as a JSON document in BDWatchdog

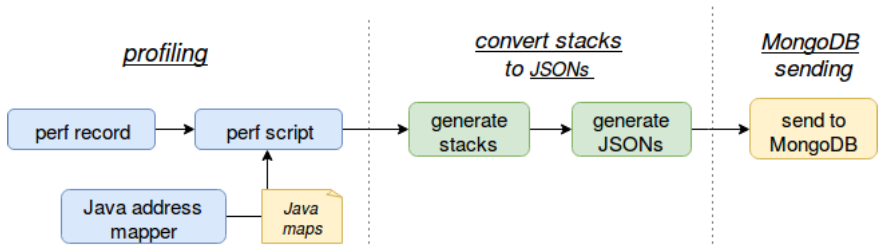


Figure 4: Processing pipeline to send profiling data

documents, like the ones used to store profiled stacks (see Listing 2 for an example). In addition, it provides sharding capabilities [46], 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 [47] 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 [48] 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 [49] 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.

4.2.4. 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 [50] 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

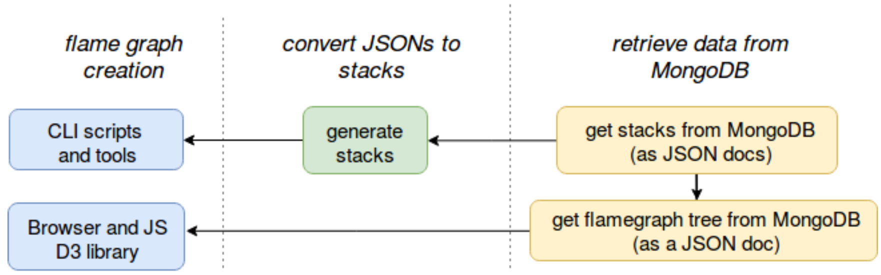


Figure 5: Processing pipeline to get profiling data

that generated and sent the data (see Figure 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 4.3.

4.3. BDWatchdog 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 6.

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.

5. Impact of BDWatchdog on Big Data workloads

This work 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). 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 in Section 4.1.2. 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 nonintrusive. 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 5.1 details the experimental configuration, while the results of the experiments are analyzed in Section 5.2.

5.1. Experimental configuration and software settings

The experiments have been conducted on a real Big Data platform [51] deployed at the CESGA supercomputing center [12]. This platform provides a cloud-based PaaS service to execute on-demand Hadoop virtual clusters. However, unlike traditional PaaS or IaaS services [17, 18] 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 [52]. Both batch and streaming workloads have been selected (see Table 3 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.

5.2. Experimental results

Figure 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 3). 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

Table 3: 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

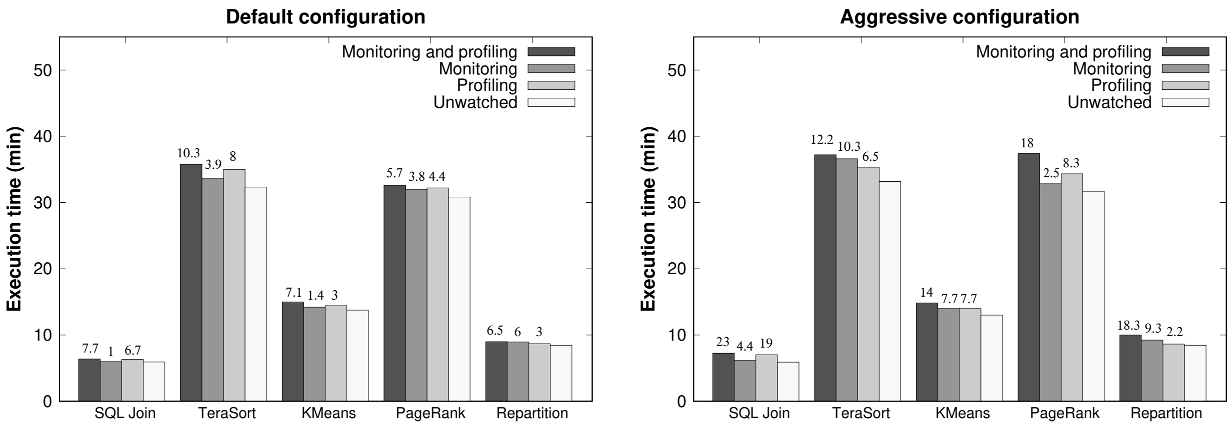


Figure 6: Impact study for different configurations and workloads (the overhead is presented as a percentage over each bar)

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.

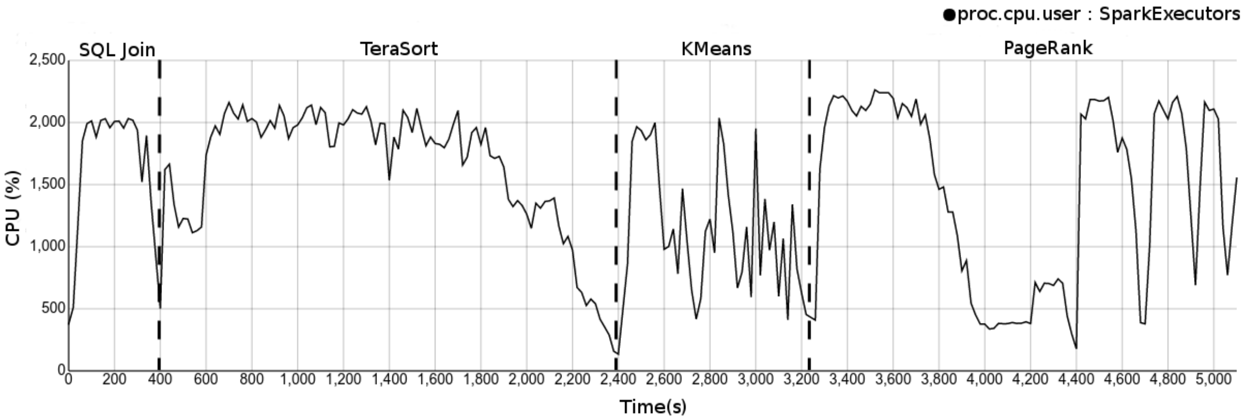


Figure 7: Aggregated CPU usage for Spark executors running batch workloads

6. BDWatchdog 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 behaviour, and resource bottlenecks that may hinder application performance. Examples of both use cases are described in Section 6.1. 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 6.2. 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 6.3. Finally, all the graphs presented in this section make use of the time-stamping feature (mentioned at the beginning of Section 4) 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.

6.1. Identification of resource usage patterns and bottlenecks

BDWatchdog allows to perform filtering and aggregation operations on a certain system resource for specific processes. For instance, the user can aggregate the resource usage of all the Spark executors running on the cluster, or the executors running on a single container. As an example, Figure 7 presents an overall view of the batch experiments shown on a single graph, aggregating the CPU usage of 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 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 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 are identified in the left graph, which means a maximum resource utilization slightly above 80%, considering the maximum value being 2,400 from adding

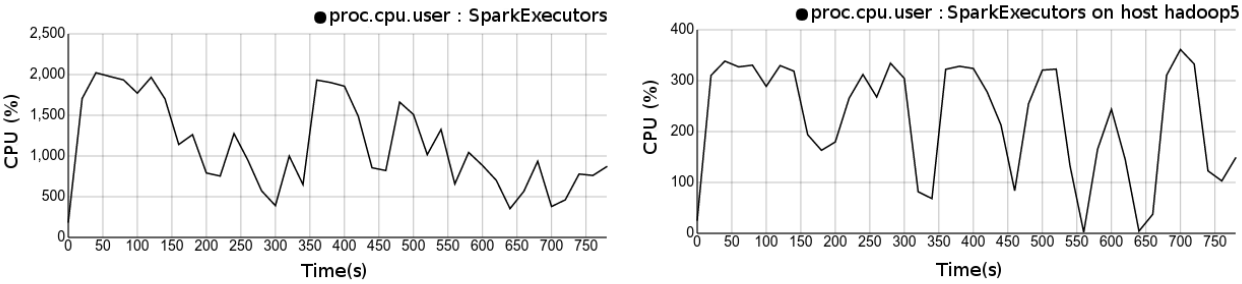
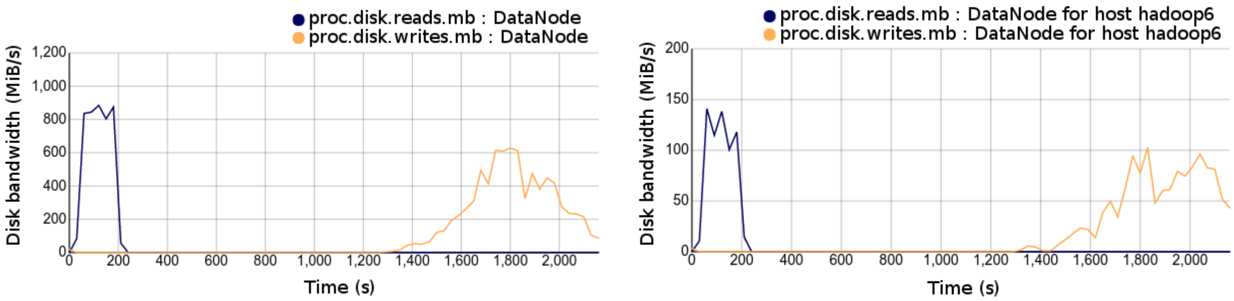
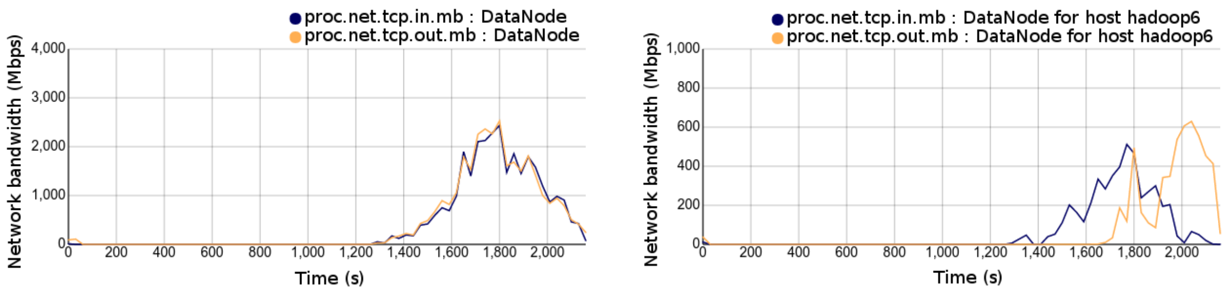


Figure 8: Aggregated CPU usage for Spark executors running KMeans



(a) Disk bandwidth



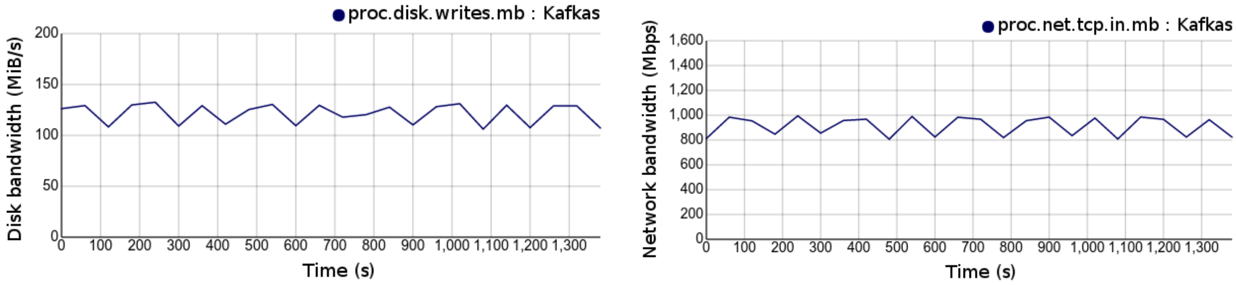
(b) Network bandwidth

Figure 9: Aggregated disk and network bandwidth for DataNode processes running TeraSort

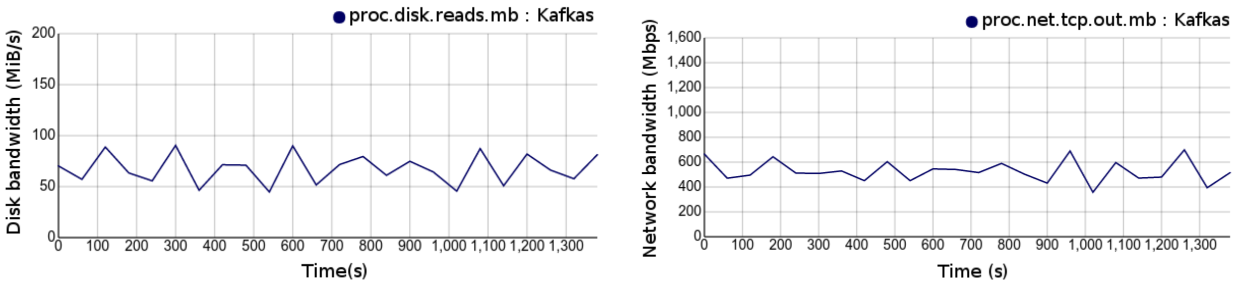
up 6 containers with 4 cores each and 100% per core. On the right graph it is possible to spot local peaks of above 300 for the container *hadoop5*, which represents a utilization of 75%. This container would be an example of an under-performing container, however the difference is close to the aggregate and falls within an expected deviation.

Figure 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 9(a) 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



(a) Disk and network bandwidth for incoming data



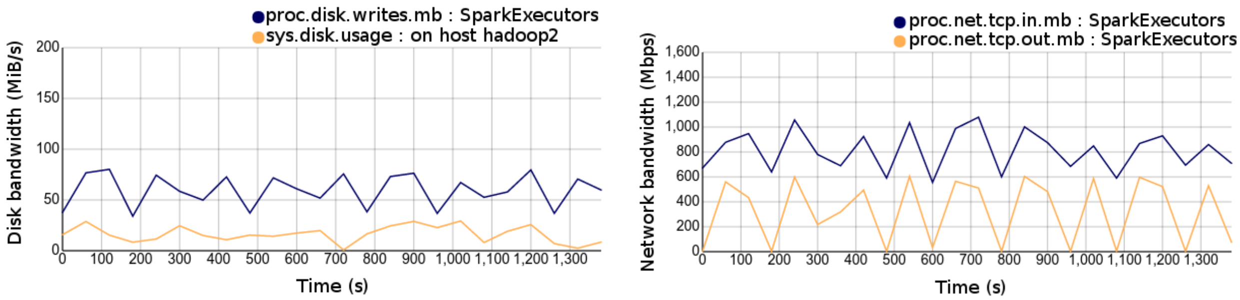
(b) Disk and network bandwidth for outgoing data

Figure 10: Aggregated resource usage for the Kafka broker while running Repartition

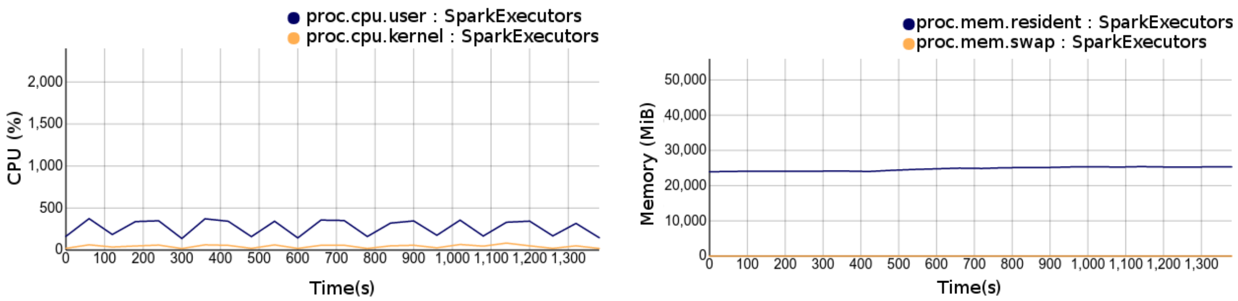
bottlenecks could appear for DataNodes. Both graphs show that the initial phase lasts for about 200 seconds with near constant values, which could mean that a bottleneck is taking place when retrieving the data, but further analysis is required to confirm this. In Figure 9(b), the incoming and outgoing network traffic is displayed. It is easily identifiable the final write phase as mentioned for Figure 9(a) 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 behaviour 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 10 depicts the data movement for the streaming Repartition workload as handled by the Kafka cluster. Figure 10(a) 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 (1000 Mbps $\approx$ 125 MiB/s). Similarly, Figure 10(b) 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 11, where the stream data processing is shown in the Spark executors. Figure 11(a) rules out any disk or network bottleneck. In the left graph, disk write bandwidth is



(a) Disk and network bandwidth



(b) CPU and memory usage

Figure 11: Aggregated resource usage for Spark executors running Repartition

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. Moreover, the right graph shows network aggregate bandwidths far from the network limit (i.e., 10 Gbps). Figure 11(b) 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.

6.2. Spotting code bottlenecks

As mentioned in Section 4.2.4, BDWatchdog allows to plot interactive flame graphs in SVG format that can be visualized 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 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 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 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.

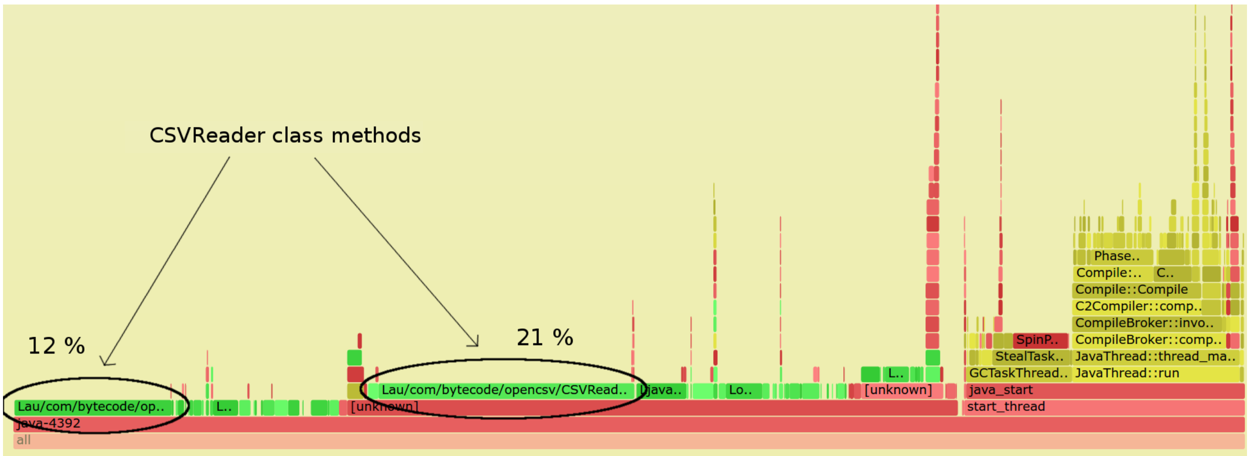


Figure 12: Flame graph for the SQL Join profiling

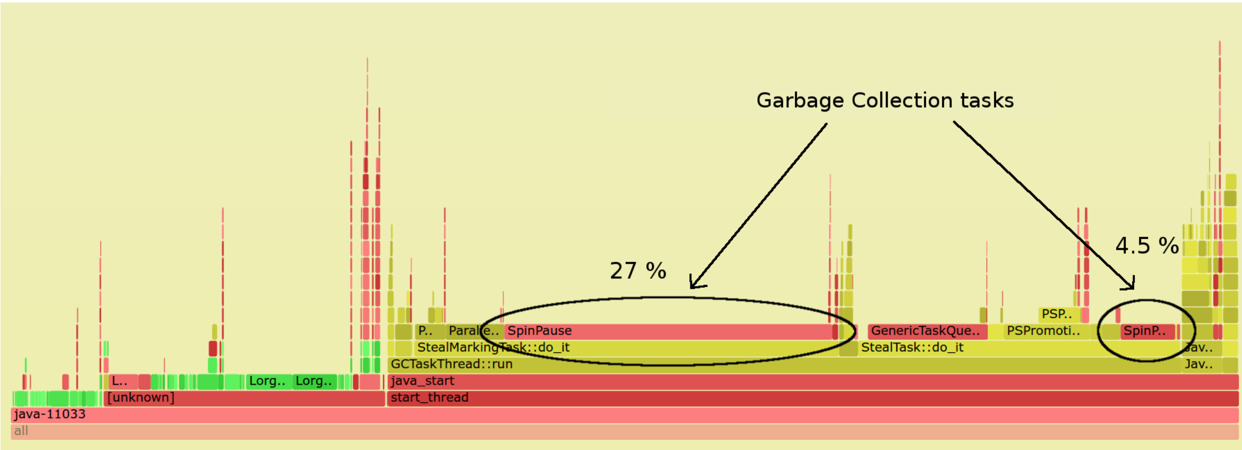


Figure 13: Flame graph for the TeraSort profiling

6.3. 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. In addition, this can 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 14 and 15 present the analysis of a PageRank execution during one of the Spark stages that is performed: flatMap. Figure 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 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 tools confirms that although this flatMap stage presents a high CPU utilization, a large percentage of the execution time is not being spent on actual data processing. In this scenario, further analysis could assess if such percentage is reasonable or if it is the result of non-optimal configuration instead. Moreover, the Spark classes identified as the most executed can also

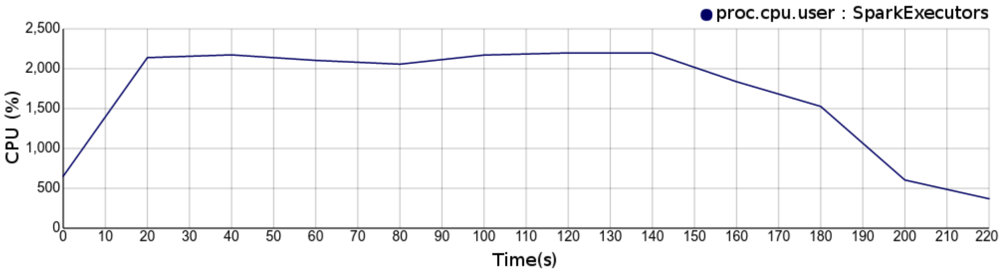


Figure 14: Aggregated CPU usage for Spark executors during a flatMap phase of PageRank

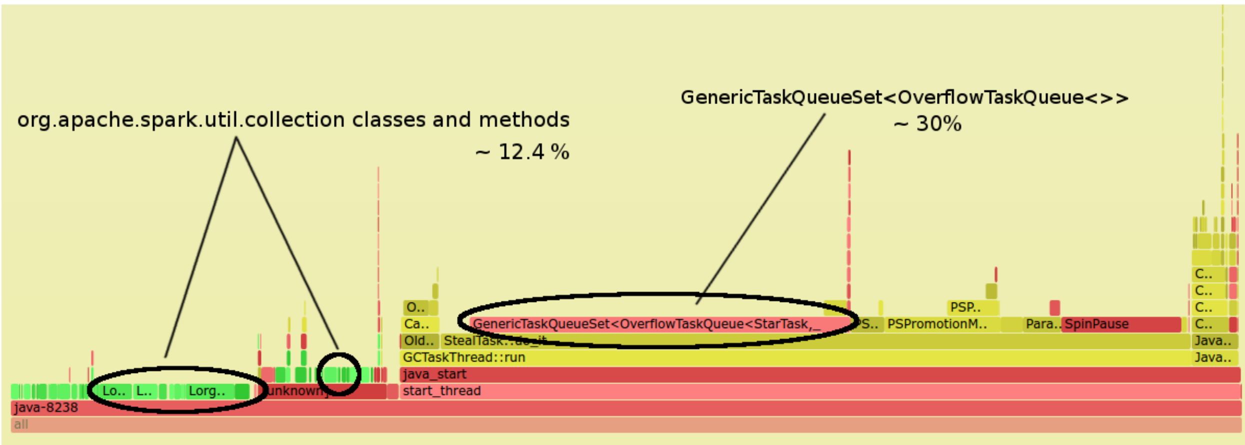


Figure 15: Flame graph for a Spark executor during a flatMap phase of PageRank

be the target of further analysis for possible source code optimizations. Overall, the combination of both tools allows to analyze the behaviour of specific stages and assert that they are both resource and code efficient.

7. Conclusions

Big Data applications are nowadays very demanding in terms of resources. In order to fully understand their behaviour and to assess their efficiency to later improve them, more in-depth analysis tools are required. Traditional monitoring solutions focus on retrieving aggregated system-wide metrics. However, this not only introduces noise because unrelated processes may be included in the end result, but also it does not allow to account for individual processes. This process-focused scenario is of great interest as most of the current Big Data frameworks typically involve just a few processes that are nevertheless very resource-demanding and thus highly susceptible of suffering bottlenecks not only on disk but also on CPU, memory or network. In addition, most of the available monitoring solutions are used on bare-metal hosts or hypervisor-based virtualization, not being designed to support newer types of virtualization. An example of the latter are the increasingly popular software containers, which share an operating system kernel to be executed, and although a system-wide view is still available, it is not representative of the container and its processes.

A similar problem arises when it comes to analyzing application performance in terms of the source code, and doing so in real time while applications are deployed on a cluster. The available profiling tools rely on being used in a more on-demand scenario, usually in development or testing environments, even requiring to alter the code with instrumentation. Moreover, if interpreted languages like Java are used, profiling data may be missing as the Java profiler is unable to account for calls outside the JVM or its internal management routines (e.g., software locks, garbage collection).

In this paper we proposed BDWatchdog, a solution for real-time analysis of Big Data frameworks and workloads that combines per-process resource monitoring and low-level profiling. For accurate resource accounting of applications 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. Moreover, this profiling solution is able to mix system and JVM profiling data, providing a broad picture of the code executed. Both monitoring and profiling tools have been designed and tested to run in real time, be scalable and work with real-world Big Data workloads and clusters, while at the same time seek for the trade-off between being accurate and remain nonintrusive.

The use cases exposed have shown that BDWatchdog allows to analyze an application or framework in terms of resource consumption and code executed, with a unified view, even when they are deployed on a cluster. Using resource analysis, this framework allows to directly spot bottlenecks, usage patterns or account for resource utilization. Some of these scenarios can be further analyzed to expose ways to improve application efficiency with rescaling or reconfiguration. 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). BDWatchdog is publicly available at `http://bdwatchdog.dec.udc.es`.

Acknowledgements

This work was supported by the Ministry of Economy, Industry and Competitiveness of Spain (Project TIN2016-75845-P, AEI/FEDER, EU), and by the FPU Program of the Ministry of Education (grant FPU15/03381). We gratefully acknowledge CESGA for providing access to the Big Data PaaS service, and also sincerely thank Dr Javier López Cacheiro for his technical support to perform the experiments.

References

[1] The Apache Software Foundation, Apache Hadoop, `http://hadoop.apache.org/`, Last visited: September 2017.

[2] M. Zaharia, et al., Apache Spark: a unified engine for Big Data processing, Communications of the ACM 59 (11) (2016) 56–65.

[3] D. Wu, A. Gokhale, A self-tuning system based on application profiling and performance analysis for optimizing Hadoop MapReduce cluster configuration, in: 20th IEEE International Conference on High Performance Computing, HiPC'13, Bangalore, India, 2013, pp. 89–98.

[4] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, S. Babu, Starfish: a self-tuning system for Big Data analytics, in: 5th Biennial Conference on Innovative Data Systems Research, CIDR'11, Asilomar, CA, USA, 2011, pp. 261–272.

[5] N. Poggi, et al., ALOJA: a systematic study of Hadoop deployment variables to enable automated characterization of cost-effectiveness, in: 2014 IEEE International Conference on Big Data, IEEE Big Data 2014, Washington DC, USA, 2014, pp. 905–913.

[6] M. Wasi-Ur-Rahman, N. S. Islam, X. Lu, D. Shankar, D. K. Panda, MR-Advisor: a comprehensive tuning tool for advising HPC users to accelerate MapReduce applications on supercomputers, in: 28th International Symposium on Computer Architecture and High Performance Computing, SBAC-PAD'16, Los Angeles, CA, USA, 2016, pp. 198–205.

[7] A. Corradi, M. Fanelli, L. Foschini, VM consolidation: a real case based on OpenStack cloud, Future Generation Computer Systems 32 (2014) 118–127.

[8] W. Yan, C. Li, S. Du, X. Mao, An optimization algorithm for heterogeneous Hadoop clusters based on dynamic load balancing, in: 17th International Conference on Parallel and Distributed Computing, Applications and Technologies, PDCAT'16, Guangzhou, China, 2016, pp. 250–255.

[9] D. Merkel, Docker: lightweight Linux containers for consistent development and deployment, Linux Journal (239) (2014) 76–91.

[10] I. Baldini, et al., Serverless computing: current trends and open problems, arXiv:1706.03178 (2017).

[11] B. Gregg, The flame graph, Communications of the ACM 59 (6) (2016) 48–57.

[12] CESGA Supercomputing Center, `http://www.cesga.es/`, Last visited: September 2017.

[13] V. K. Vavilapalli, et al., Apache Hadoop YARN: Yet Another Resource Negotiator, in: 4th Annual Symposium on Cloud Computing, SOCC'13, Santa Clara, CA, USA, 2013, pp. 5:1–5:16.

[14] K. Shvachko, H. Kuang, S. Radia, R. Chansler, The Hadoop distributed file system, in: 26th IEEE Symposium on Mass Storage Systems and Technologies, MSST'10, Incline Village, NV, USA, 2010, pp. 1–10.

[15] J. Dean, S. Ghemawat, MapReduce: simplified data processing on large clusters, Communications of the ACM 51 (1) (2008) 107–113.

[16] N. Garg, Apache Kafka, Packt Publishing Ltd, 2013.

[17] Amazon EMR, `https://aws.amazon.com/emr/`, Last visited: September 2017.

[18] Azure HDInsight, `https://azure.microsoft.com/services/hdinsight/`, Last visited: September 2017.

[19] Amazon Web Services Lambda, `https://aws.amazon.com/lambda/`, Last visited: September 2017.

[20] V. M. Weaver, Linux perf_event features and overhead, in: 2nd International Workshop on Performance Analysis of Workload Optimized Systems, FastPath'13, Austin, TX, USA, 2013.

[21] G. Ren, E. Tune, T. Moseley, Y. Shi, S. Rus, R. Hundt, Google-wide profiling: a continuous profiling infrastructure for data centers, IEEE Micro 30 (4) (2010) 65–79.

[22] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G. Y. Wei, D. Brooks, Profiling a warehouse-scale computer, in: 42nd ACM/IEEE Annual International Symposium on Computer Architecture, ISCA'15, Portland, OR, USA, 2015, pp. 158–169.

[23] M. Wagner, G. Llort, E. Mercadal, J. Giménez, J. Labarta, Performance analysis of parallel Python applications, in: International Conference on Computational Science, ICCS 2017, Zurich, Switzerland, 2017, pp. 2171–2179.

[24] A. Knüpfer et al., Score-P: a joint performance measurement run-time infrastructure for Periscope, Scalasca, TAU, and Vampir, in: 5th International Workshop on Parallel Tools for High Performance Computing, Dresden, Germany, 2011, pp. 79–91.

[25] V. Pillet, J. Labarta, T. Cortes, S. Girona, PARAVER: a tool to visualize and analyze parallel code, in: 18th World occam and Transputer User Group Technical Meeting, WoTUG-18, Manchester, United Kingdom, 1995, pp. 17–31.

[26] H. Brunst, M. Winkler, W. E. Nagel, H.-C. Hoppe, Performance optimization for large scale computing: the scalable VAMPIR approach, in: International Conference on Computational Science, ICCS 2001, San Francisco, CA, USA, 2001, pp. 751–760.

[27] E. Kuehn, M. Fischer, C. Jung, A. Petzold, A. Streit, Monitoring data streams at process level in scientific Big Data batch clusters, in: IEEE/ACM International Symposium on Big Data Computing, BDC'14, London, United Kingdom, 2014, pp. 90–95.

[28] J. Case, M. Fedor, M. Schoffstall, J. Davin, A Simple Network Management Protocol (SNMP), Internet Requests for Comments 1157 (1990) http://www.rfc–editor.org/info/rfc1157, Last visited: September 2017.

[29] H. Liu, Y. Liu, J. Zheng, The application of Cacti in the campus network traffic monitoring, Computer & Telecommunication 4 (2008) 4.

[30] P. Tader, Server monitoring with Zabbix, Linux Journal (195) (2010) 72–78.

[31] M. L. Massie, B. N. Chun, D. E. Culler, The Ganglia distributed monitoring system: design, implementation, and experience, Parallel Computing 30 (7) (2004) 817–840.

[32] E. Imamagic, D. Dobrenic, Grid infrastructure monitoring system based on Nagios, in: Workshop on Grid Monitoring, GMW'07, Monterey Bay, CA, USA, 2007, pp. 23–28.

[33] Amazon CloudWatch, `https://aws.amazon.com/cloudwatch/`, Last visited: September 2017.

[34] Amazon Web Services (AWS), `https://aws.amazon.com/`, Last visited: September 2017.

[35] atop: performance monitor for Linux, `https://www.atoptool.nl/index.php`, Last visited: September 2017.

[36] nethogs: Linux 'net top' tool, `https://github.com/raboof/nethogs`, Last visited: September 2017.

[37] netatop: module for network per-process statistics, `https://www.atoptool.nl/netatop.php`, Last visited: September 2017.

[38] VisualVM website, `https://visualvm.github.io/`, Last visited: September 2017.

[39] E.-Y. Cho, JProfiler: code coverage analysis tool for OMP project, Tech. rep., CMU 17-654 & 17-754 (2006).

[40] S. Chaudhuri, U. Dayal, An overview of data warehousing and OLAP technology, ACM SIGMOD Record 26 (1) (1997) 65–74.

[41] The Apache Software Foundation, HBase: a distributed database for large datasets, `https://hbase.apache.org/`, Last visited: September 2017.

[42] B. Sigoure, OpenTSDB: a scalable, distributed time series database, in: O'Reilly Open Source Convention, OSCON'11, Portland, OR, USA, 2011.

[43] T. W. Wlodarczyk, Overview of time series storage and processing in a cloud environment, in: 4th IEEE International Conference on Cloud Computing Technology and Science, CloudCom'12, Taipei, Taiwan, 2012, pp. 625–628.

[44] M. Bostock, V. Ogievetsky, J. Heer, D3: Data-Driven Documents, IEEE Transactions on Visualization and Computer Graphics 17 (12) (2011) 2301–2309.

[45] P. Membrey, E. Plugge, T. Hawkins, The definitive guide to MongoDB: the noSQL database for cloud and desktop computing, Springer, 2010.

[46] E. Dede, M. Govindaraju, D. Gunter, R. S. Canon, L. Ramakrishnan, Performance evaluation of a MongoDB and Hadoop platform for scientific data analysis, in: 4th ACM Workshop on Scientific Cloud Computing, ScienceCloud'13, New York, NY, USA, 2013, pp. 13–20.

[47] Eve: a Python REST API framework, `http://python-eve.org/`, Last visited: September 2017.

[48] M. Grinberg, Flask web development: developing web applications with Python, O'Reilly Media, Inc., 2014.

[49] Gunicorn: a Python WSGI HTTP server for UNIX, `http://gunicorn.org/`, Last visited: September 2017.

[50] Flame graph project, `https://github.com/brendangregg/FlameGraph`, Last visited: September 2017.

[51] CESGA Big Data PaaS service, http://bigdata.cesga.es/, Last visited: September 2017.
[52] S. Huang, J. Huang, J. Dai, T. Xie, B. Huang, The HiBench benchmark suite: characterization of the MapReduce-based data analysis, in: 26th IEEE International Conference on Data Engineering Workshops, ICDEW'26, Long Beach, CA, USA, 2010, pp. 41–51.