Vertical Elasticity on Marathon and Chronos Mesos frameworks

Sergio López-Huguet[a,*], Igor Natanael[b], Andrey Brito[b], Ignacio Blanquer[a]

[a]*Instituto de Instrumentación para Imagen Molecular (I3M) Centro mixto CSIC - Universitat Politècnica de València, Camino de Vera s/n, 46022, Valencia*
[b]*Laboratório de Sistemas Distribuídos, Universidade Federal de Campina Grande, R. Aprígio Veloso, 882 - Universitário, Campina Grande - PB, 58429-900, Brasil*

Abstract

Marathon and Chronos are two popular Mesos frameworks that are widely used for deploying fault-tolerant services and periodic batch jobs. Marathon and Chronos provide by design mechanisms for horizontal elasticity, scaling up and down the number of job and service instances. Horizontal elasticity is appropriate when the problems that are solved are inherently parallel. However, when the problem cannot benefit from an increase of the amount of resources, vertical elasticity must be considered. This work implements on top of Marathon and Chronos Mesos frameworks, a mechanism to vary the resources associated to an executor dynamically, according to its progress and considering specific Quality of Service (QoS). The mechanism developed provides a wrapper executable and a service that takes the decision of increasing or decreasing the resources allocated to different Chronos iterations or a long-living Marathon application. The mechanism makes use of checkpointing techniques to preserve the execution of Marathon applications and leverages OpenStack Monasca for the monitoring. [1]

Keywords: Quality of service, Cloud Computing, Vertical elasticity, Mesos frameworks

1. Introduction

Apache Mesos [1] is a large-scale Resource Management System (RMS) that can partition and assign the resources of computing infrastructure (CPU, memory, I/O, disk and special resources) across several job schedulers, namely frameworks. Apache Mesos is not used to execute applications, but to allocate resources to those frameworks. Apache Mesos is becoming very popular due to a large number of frameworks supported. Among them, Marathon [2] and Chronos [3] provide means for deploying reliable services and periodic batch applications.

Applications deployed on Marathon can be scaled up and down according to specific triggers. However, this horizontal elasticity is appropriate (e.g. stateless web services scaling on a variable access workload) when the prob-

[1]The work in this article has been funded by projects BIGCLOE and EUBra-BIGSEA, BIGLOE is funded by the Spanish "Ministerio de Economía, Industria y Competitividad" with reference number TIN2016-79951-R and EUBra-BIGSEA is funded jointly by the European Commission under the Cooperation Programme, Horizon 2020 grant agreement No 690116 and the Brazilian Ministério de Ciência, Tecnologia e Inovação (MCTI).

[*]Corresponding author: Tel. +34963877356

Email addresses: `serlohu@upv.es` (Sergio López-Huguet), `igornsa@lsd.ufcg.edu.br` (Igor Natanael), `andrey@dsc.ufcg.edu.br` (Andrey Brito), `iblanque@dsic.upv.es` (Ignacio Blanquer)

lems that are solved are inherently parallel, but when the problem cannot benefit from an increase of the number of resources, another type of elasticity must be considered. Vertical elasticity consists of resizing dynamically the resources assigned to each application to meet varying workload demands or a previously Quality of Service (QoS) agreed. Vertical elasticity can be preferred from horizontal elasticity [4] in some scenarios. Studies propose proactive elasticity [5] as an effective mechanism to improve QoS.

This work uses the RMS Apache Mesos, whose objective is to execute applications by using distributed frameworks and controlling resources such as storage, CPU and memory in a collection of computational nodes. Depending on the framework features, it is possible to scale-out or scale-in worker nodes (horizontal elasticity) or adjust the CPU share (vertical elasticity) associated to each execution instance. There are some frameworks among Apache Mesos but this work focuses on Marathon and Chronos. Marathon is a production-grade container orchestration platform with high availability and fault tolerant framework. Furthermore, it is designed for managing long running applications, which is a common type of application that executes on Cloud infrastructure. Besides, Marathon provides methods to realise horizontal and vertical elasticity but only on stateless applications. Chronos is a distributed and fault-tolerant scheduler designed as a replacement for Linux Cron [6]. Furthermore, it supports the execution of Docker containers and also provides a mechanisms for varying their allocated resources. Both frameworks are complementary and widely used together. There is certain type of applications in Cloud computing that runs several iterations and, with this framework, it is possible to manage such class. More precisely, the work of this wants to address two use cases, each one with one of the frameworks:

- Chronos can schedule a set of application iterations with a given periodicity and start them at a given time. It is not targeted at the bag-of-tasks execution model, in which applications are independent and embarrassingly parallel, which could be controlled by simpler horizontal elasticity mechanisms. Chronos can execute applications defined as a workflow, in which each stage requires the completion of a previous stage. We consider the simple case of an iterative workflow with a fixed number of iterations of one application. Therefore, vertical elasticity is essential to define the most appropriate amount of resources to execute each iteration in the desired timeframe.

- Marathon can run a multithreaded application whose computational cost is known previously, so it is expected to complete once a certain amount of CPU time has been consumed. We cannot apply horizontal elasticity as it runs in a single node. If vertical elasticity is not utilised, then the resource allocation must ensure that the deadline is met. Given a scenario of nodes with 4 cores each, if, for example, an application requires 1.5 CPUs for a given time, and this allocation is static, the remaining 2.5 cores will be wasted if no matching request comes to the system. In our approach, the job will use the 4 cores even if 1.5 cores are allocated if no competing applications are allocated. If a new request comes to the system asking for 2.5 cores, the system could allocate it in the same node, reducing the CPU share according to the 1.5 / 2.5 ratio. Therefore, the application could have advanced computation, anticipating load to earlier time slots and leading to a better load balancing.

Mesos, Marathon and Chronos are only examples of the tools that appear to potentialize the usage of containers. Containers enable the developer to encapsulate the applications and their libraries winning more flexibility, scalability, boot up time and resource efficiency than the Virtual Machines (VM) [7] [8]. Thus, the containers become the most popular way for packaging and deploying applications on Cloud infrastructures. Docker containers, in turn, have become the most popular containers platform. For this reason, the development proposed in this work is based on Docker containers instead of Mesos native containers.

Thereby, the work is focused on designing and implementing a mechanism that, given a job specification and a QoS target, it can deploy it in Mesos framework (Chronos or Marathon) using a Docker container, monitor such job and vary the assigned resources with the objective of achieving the agreed QoS.

This paper is organised as follows. Section 2 describes the problem in detail and the defines the two possibles scenarios according to the above use cases and the frameworks. Section 3 describes the past works of elasticity in VM and Docker containers. Section 4 describes the technologies mentioned in this paper. Section 5 presents the mechanism developed. Section 6 describes the tests for evaluating this mechanism and then discusses the obtained results. Finally, Section 7 presents the conclusions of the work explained in the paper.

2. Description of the problem

The motivation of the work is to provide a mechanism for adjusting the allocation of CPU resources to a running application inside a Docker container managed by a Mesos Framework (Marathon or Chronos). The design principles are the following:

- There is a known, feasible deadline for a given execution of a job or, if not, there is a Quality of Service target, expressed as the CPU time that must be allocated to the application in a given timeframe.
- The application can benefit from multiple cores.
- Multiple applications are competing for the resources – otherwise, the QoS guarantees would not have any meaning.

Therefore, we identify the following requirements:

- Deadline should be provided at submission time, so this information is associated with the application.
- A fine-grain monitoring of the task (e.g. consumed CPU time for the individual task) is needed. Considering that the tasks run in a distributed environment, this may require identifying the physical resources where the task runs.
- The progress of the application must be computed and defined. We will use three states: *under-progress*, if the application progress is below a given threshold of the expected progress or CPU allocation; *ontime*, if the

application progress is between the acceptable, expected progress interval, given a threshold; and *overprogress*, if an application has advanced more than a given threshold with respect to the expected progress. The user can configure these thresholds at submission time.

- An actuator will trigger a new resource allocation if needed. An *underprogress* state will lead to an increase of resources and an *overprogress* state to a reduction of such resources. The user can configure these increments or decrements at submission time.
- In case of the application state needs to be frozen and restored, checkpointed is needed.

The motivation for these requirements can be seen in the following scenarios:

- Chronos applications: The user submits an application to Chronos that iterates multiple times through independent executions, which could not take place concurrently due to flow dependencies, external data dependencies, etc. The user wants to guarantee that all of the individual executions of an application are completed before a given deadline. Each iteration is started with specific resource allocation and, according to the application progress at the end of each iteration, the developed mechanism decides to modify the allocated CPU share to meet the QoS agreed. This modification is decided by the developed mechanism but is performed by Chronos.
- Long-living Marathon applications: The user submits a long-living application to Marathon with the guarantee that a minimum share of the CPU time has been allocated to that application in a given time frame. The developed mechanism queries, periodically, the monitoring system to compute the application progress. If a modification of the allocated CPU share is needed, the mechanism varies it using Marathon. It should be pointed out that the application state must be preserved because, unlike the first scenario, the modification of allocated resources is performed during the execution. This scenario assumes that other tasks have been scheduled in the same resources with different QoS requirements and submission times, so temporarily speeding up or decelerating other application may lead to a lower overall QoS violation ratio. It should be noted that the mechanism is designed for deciding to modify the allocated CPU share of an application based only on its application progress (not considering the the progress of all applications). Similar to the Chronos scenario, the modification of allocated resources is performed by the framework but is decided by the developed mechanism.

3. Related work

Elasticity is an interesting research area in Cloud Computing. There are many works focused on horizontal elasticity and vertical elasticity. Horizontal elasticity consists of providing additional resources (more VMs or Docker containers) dynamically to meet an increase or decrease of service demand (e.g. increase of requests in a Cloud service). This approach is appropriate when the problems can be resolved in a parallel way. Vertical elasticity consists of modifying the allocated resources of a Virtual Machine (VM) or a Docker container to meet an increase of demand (e.g. a scientific application that needs more RAM memory for finalizing its execution). The work of this paper is focused on vertical elasticity.

Most hypervisors and Cloud IaaS support vertical elasticity in VMs. OpenNebula and OpenStack provide with resize functions in which stopped VM can be provided with higher or lower number of CPUs and RAM memory. However, there is no support in OpenNebula and OpenStack for dynamic resizing of VMs, as they will require acting both at the level of the virtualisation hypervisor (varying the assigned resources to the VM) and at the VM's guest Operating System (updating the resources available in the OS).

One approach for providing vertical elasticity of CPU for VM's can be leveraging the CPU CAP (the maximum amount of CPU resources a VM can use) and the physical memory allocated by the virtualisation hypervisor. It should be pointed out that these techniques do act only at the level the virtualisation hypervisor. This way, the internal configuration of the VM's OS remains the same, but it is provided with a higher share of physical resources, so the virtual CPU can run faster or slower, and have more RAM mapped on physical RAM. [9] presents a mechanism named CloudScale, to automate fine-grained elastic resource scaling for multi-tenant Cloud computing infrastructures. In contrast, other approaches such as CPU hotplugging requires that both the guest Operating System supports dynamic plugging of CPUs and that the applications inside the VM are built for multithreaded or multicore parallelism.

From [10], vertical elasticity approaches can be categorised into performance-based, capacity-based, and hybrid approaches. The virtualisation hypervisor supports these approaches with two mechanisms: add or remove memory, also named hot memory plugging, and memory ballooning ([11], [10]). However, in such cases, the allocation of resources is performed at the level of the whole Virtual Machine, which will affect all the applications running on it.

Vertical elasticity in Docker containers are also considered in a few works. [12] provides a tool to perform manually vertical elasticity in Cloud infrastructures. [13] presents Elasticdocker, which is a mechanism that modifies the allocated resources (CPU time, vCPU cores and memory) of a Docker container according to the workload demand. Elasticdocker monitors the CPU time, CPU utilization vCPUs and memory utilization to take the elasticity decisions and implements these decisions modifying the `cgroups` pseudo-files of the Docker container directly.

There is also some related work on mechanisms to address elasticity aspects in Mesos frameworks. [14] provides a mechanism to scale in and out stateless Marathon services (horizontal elasticity) but considering that services do not store a persistent state so they can be restarted. This approach considers both memory and CPU. Regarding vertical elasticity, a Mesos executing and monitoring framework called Makeflow [15] is proposed for a bag of tasks, which adjusts the number of vCPUs of a series of independent jobs according to their actual utilization. To the best of our knowledge, there are currently no tools to provide vertical elasticity for applications embedded inside Docker Containers executed by Mesos frameworks (Marathon and Chronos). Thus, the work described in this paper aims to design and implement a flexible mechanism for applications that varies the percent of CPU time to a specific application in a highly loaded infrastructure to meet the agreed QoS.

4. Underlying technologies

This work relies on a set of well-known technologies for containers, resource orchestration, checkpointing, job submission and monitoring.

4.1. Application Delivery

Applications were run on Docker containers. Formally, a container is a group of processes that are executed in an isolated and controlled way on the host kernel on an independent and virtualised filesystem. It provides a convenient mechanism to embed software dependencies for application delivery. Containers hold the whole execution context for an application. In this sense, CRIU [16] is a project for Linux operating system that enables to freeze a running application as a collection of files called checkpoint. Using checkpointing a user can freeze and restore the application at the same execution point as it was during the time when the checkpoint was made, even in another machine. Currently, CRIU is included into many Linux distributions and it is integrated into containers platforms like OpenVZ [17], LXC [18] or Docker (with the experimental release).

4.2. Resource management

Apache Mesos is a middleware (set of daemons/services) for cluster management that provides efficient resource isolation and sharing across distributed applications (frameworks) or roles (users). Mesos can run on top of Virtual Machines, bare metal or Docker containers. Mesos are composed of two main components: the master and a set of agents (or worker nodes).

The Mesos master manages agent daemons running on the worker nodes. Worker nodes run Mesos agents that register with the master and offer their resources (CPU, disk, ports, special hardware, etc.). Mesos provides two mechanisms for reserving the registered resources of the worker nodes for the execution frameworks: static and dynamic reservation. Static reservation consists of specifying the resources reserved for a certain role in the agent startup. Dynamic reservation enables roles and authorised frameworks to reserve or liberate resources after agent startup.

Execution frameworks, in turn, are composed of two main components: scheduler and executor. The framework scheduler registers with the Mesos master for receiving resource offers. When an offer is available, the framework scheduler sends the information of the amount of resources that it will be assigned to the framework's tasks to the Mesos master. Then, the Mesos master sends the framework's tasks to the corresponding Mesos agent, which allocates appropriate resources to the framework's executor. The framework's executors are in charge of launching the framework's tasks in the Mesos agent. The Figure 1 illustrates this procedure.

There are some frameworks [20] for Mesos to execute long running services (Aurora, Marathon, etc), Big Data processing (Spark, Hadoop, Storm, etc), batch scheduling (Chronos, Cook, Jenkins, etc.), data storage (Cassandra, Ceph, etc) and machine learning (TFMesos).

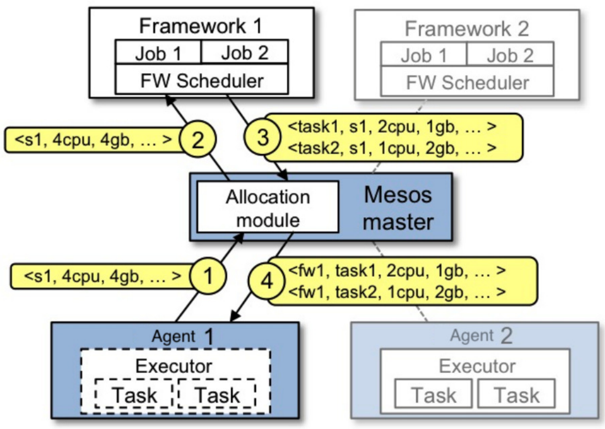


Figure 1: Example of resource offer from Mesos agent and how it is assigned to a task of a framework's executor. Image source: [19].

Marathon is high availability and fault-tolerant Mesos framework that acts like a *Platform as a Service* (*PaaS*). This framework is designed for managing long run services, i.e., services that must be permanently running. Marathon obtains high availability thanks to periodic checks of the task's health. Activating the checkpointing feature in Marathon enables tasks to continue running during Mesos-slave restarts and Marathon scheduler failovers. The users can interact with Marathon using the REST protocol or graphical web interface.

Chronos is a distributed and fault-tolerant Mesos framework that can be used for job orchestration. This framework is a suitable scheduler for periodic tasks that need to be triggered periodically. Besides, it supports the creation of dependent jobs. These type of applications will run when a list of "parent" applications are executed before at least once. Chronos provides similar features as Marathon but more focused on periodic jobs and the applications that cannot be composed by more than one instance. The users can interact with Chronos using REST protocol or graphical web interface.

4.3. Monitoring

Monitoring is a key issue when managing elasticity. Monitoring should be fine-grained to focus on the real consumption of resources of the specific task that should be monitored, and lightweight for causing the minimum overhead and disturbance. For this purpose, Openstack Monasca [21] has been selected, which is an open source, highly scalable, multitenant and fault tolerant monitoring tool that can be integrated with OpenStack [22].

In Monasca, a *Metric* defines a type of monitored resource. A metric is defined by a name, often representing a hierarchy, and a set of dimensions. The unit of monitoring is a measurement, which contains values for the dimensions and a timestamp. When a *Metric* is stored in Monasca, the user can realise queries or define *Alarms*, which are a composed by one boolean expression and a *Notification* method. This expression is an evaluation of a *Metric* with a threshold. In turn, each expression can be formed by one or more expressions.

The data is stored in Monasca by two entities: the users and the Monasca Agent [23]. The users can send *Metrics* to Monasca, once they are authenticated by Openstack KeyStone [24], using command line interface or by REST API

[25]. The Monasca Agent [23] is a Python tool that gathers *Metrics* using available plug-ins [26]. There are default plugins for standard resources (such as IO, network, CPU usage) and even for monitoring higher-level services and applications (such as CEPH, MongoDB, Kafka, and tens of others). In this work, Monasca data is stored by the Monasca Agent and the component Supervisor (described in Section 5) of the mechanism developed.

5. The proposed system architecture

As mentioned before, the two scenarios considered are a Chronos application with multiple independent iterations and long-living Marathon applications. According to the scenario, the architecture varies in some aspects, which will be explained along this section. Figures 2 and 3 depict the design of the architecture for both scenarios. Applications are specified using JavaScript Object Notation (JSON) format.

The mechanism architecture is composed of three main components: Launcher, Supervisor and Executor. These components are implemented in Python.

The Launcher is a command-line tool in charge of the submission of the application. The user runs the Launcher specifying the application in JSON format with the QoS information, the parameters to connect and configure the Supervisor, the address of Supervisor and the credentials of the Mesos frameworks (Chronos and Marathon). First, the Launcher assigns each application a unique identifier (UUID). Then, the Launcher creates a new application specification based on the user application specification. The changes (that are detailed in Section 5.1) for creating the modified specification are done according to the selected scenario. Afterwards, the Launcher submits via REST the new application specification to the appropriate framework. Once submitted, the Launcher sends through a REST call the relevant information (this information is detailed in Section 5.1) to the Supervisor.

The Supervisor is a REST service that receives the information for monitoring from the Launcher and the Docker containers (applications) that are running on the working nodes. In addition, it computes the application performance state and decides if scaling is needed. If the allocated CPU percentage must be modified, the Supervisor obtains the application specification from the framework scheduler (Marathon or Chronos), modifies it and re-submits it to the framework. Then, the framework relaunches the application with the new value of the percentage of CPU time assigned (in case of Marathon, the framework first terminates the current execution).

The Supervisor monitors the applications with two approaches: passive and active. The passive approach is chosen for Chronos scenario and active for Marathon scenario. In the first approach, the Supervisor computes the progress state and decides, if it necessary, to scale when the Supervisor receives the notification from the application at the end of the execution of each iteration. In the active approach, when it receives a notification from the Executor (this component is described below), the Supervisor starts monitoring the application state. This monitoring is periodically and, for each instant of monitoring, the Supervisor decide if an update in the allocated resources is needed. If it is required to update the allocated resources, the Supervisor notifies the framework (Marathon or Chronos). Once the execution of the application is completed (the Supervisor receives a notification from the Executor), the Supervisor stops monitoring the application. Meanwhile, the Supervisor sends metrics to Monasca, so progress can be displayed.

All the applications are embedded in Docker containers. In the Chronos scenario, the scaling decisions are implemented between iterations, so checkpointing is not required to maintain the execution state. In the Marathon scenario, the scaling decisions are implemented during the execution of the Docker container. The container will be forced to restart by Marathon, thus, checkpointing is needed for preserving the execution state. An additional component, the Executor, is required to start and resume the execution of a container from a Checkpoint. This requirement is a consequence of the fact that Marathon cannot initiate Docker containers based on checkpoints, thus it cannot handle the stop-resume process after an adjustment. It should be noted that, when applications are not embedded in a Docker container in Chronos or Marathon, these applications are embedded in Mesos Native container, so the Executor is executed inside a Mesos Native container.

First, the Executor performs several preprocessing tasks: registering initial data on Monasca, checking the existence of a previous checkpoint to be resumed and notifying that the application execution will be started through a REST call to the Supervisor. Then, it starts or resumes the execution of the Docker container from a checkpoint stored in an accessible directory for all worker nodes. Afterwards, the Executor waits until the Docker container's execution is completed. For this purpose, it runs a command (`docker logs -f container_id`) that ends when the Docker container's execution is finished.

When the Supervisor indicates to the Marathon scheduler that the allocated CPU percentage must be modified, the Marathon scheduler notifies that the current execution will be terminated to the Marathon executor. If the Executor receives a signal from the Marathon executor, it makes a checkpoint and stores it to a directory shared by all Mesos agents. Once the execution of the Docker container is ended, the Executor notifies it through a REST call to the Supervisor and cleans the shared directory. As mentioned above, the Executor detects when the Docker container ends its execution because the Executor is running a command that ends when the Docker container is finished.

The architecture's flow can be divided into four stages: application submitting and execution, monitoring, decision-making, and adjustment.

5.1. Application submission and execution

The Chronos scenario deals with several executions of an application, which must be completed according to the given deadline. As described above, checkpointing is not required for this scenario. Therefore, the Docker container is directly managed by the Chronos executor. The modifications required to create the new application specification for this scenario are only focused on the monitoring process. The Launcher adds a REST request to the Supervisor in the original command field of the application specification. This request is inserted in the command field after the original command's value, making the Docker container do a REST request to Supervisor once the execution is complete. The request's message is a JSON object formed by the start and the end of the application execution, the application name and UUID.

The Marathon scenario aims to guarantee that the mechanism allocates to the application, at least, a predefined

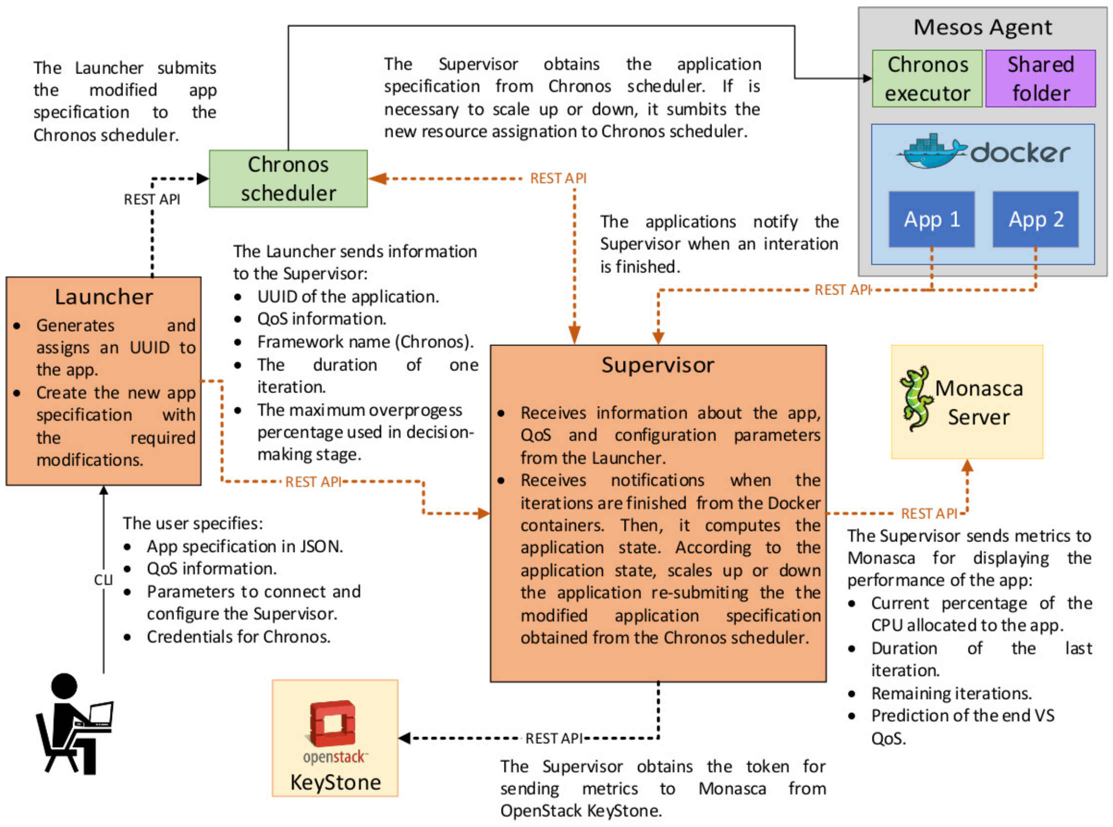

The Launcher submits the modified app specification to the Chronos scheduler.
The Supervisor obtains the application specification from Chronos scheduler. If is necessary to scale up or down, it sumbits the new resource assignation to Chronos scheduler.
Mesos Agent
Chronos executor
Shared folder
docker
App 1
App 2
Chronos scheduler
REST API
REST API
The applications notify the Supervisor when an interation is finished.
REST API
Launcher
Generates and assigns an UUID to the app.
Create the new app specification with the required modifications.
The Launcher sends information to the Supervisor:
UUID of the application.
QoS information.
Framework name (Chronos).
The duration of one iteration.
The maximum overprogess percentage used in decision-making stage.
REST API
Supervisor
Receives information about the app, QoS and configuration parameters from the Launcher.
Receives notifications when the iterations are finished from the Docker containers. Then, it computes the application state. According to the application state, scales up or down the application re-submiting the the modified application specification obtained from the Chronos scheduler.
Monasca Server
REST API
The Supervisor sends metrics to Monasca for displaying the performance of the app:
Current percentage of the CPU allocated to the app.
Duration of the last iteration.
Remaining iterations.
Prediction of the end VS QoS.
The user specifies:
App specification in JSON.
QoS information.
Parameters to connect and configure the Supervisor.
Credentials for Chronos.
CLI
openstack
KeyStone
REST API
The Supervisor obtains the token for sending metrics to Monasca from OpenStack KeyStone.

Figure 2: Design of the architecture for the Chronos scenario.

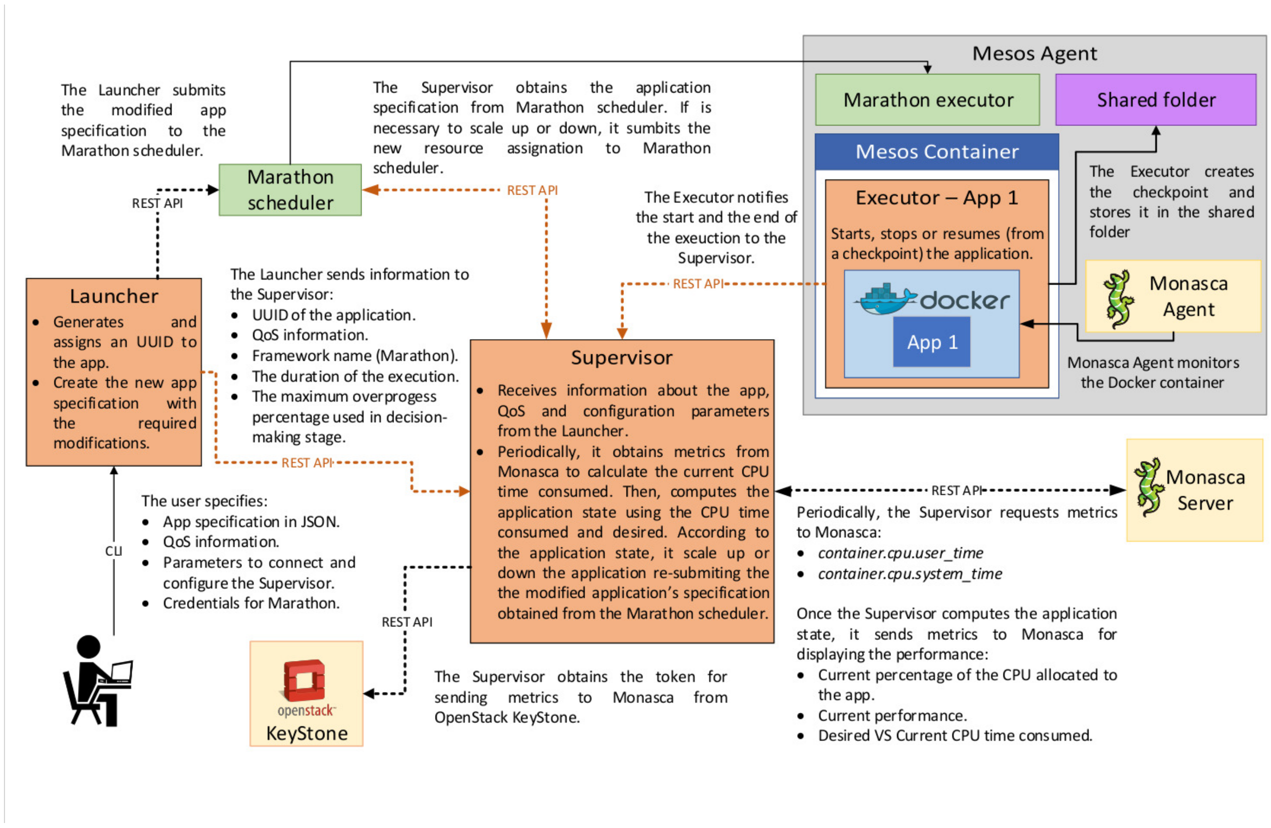

Mesos Agent
Marathon executor
Shared folder
Mesos Container
Executor – App 1
Starts, stops or resumes (from a checkpoint) the application.
docker
App 1
The Executor creates the checkpoint and stores it in the shared folder
Monasca Agent
Monasca Agent monitors the Docker container
The Launcher submits the modified app specification to the Marathon scheduler.
Marathon scheduler
REST API
The Supervisor obtains the application specification from Marathon scheduler. If is necessary to scale up or down, it sumbits the new resource assignation to Marathon scheduler.
REST API
The Executor notifies the start and the end of the exeuction to the Supervisor.
REST API
Launcher
Generates and assigns an UUID to the app.
Create the new app specification with the required modifications.
The Launcher sends information to the Supervisor:
UUID of the application.
QoS information.
Framework name (Marathon).
The duration of the execution.
The maximum overprogess percentage used in decision-making stage.
REST API
Supervisor
Receives information about the app, QoS and configuration parameters from the Launcher.
Periodically, it obtains metrics from Monasca to calculate the current CPU time consumed. Then, computes the application state using the CPU time consumed and desired. According to the application state, it scale up or down the application re-submiting the the modified application's specification obtained from the Marathon scheduler.
REST API
Monasca Server
Periodically, the Supervisor requests metrics to Monasca:
container.cpu.user_time
container.cpu.system_time
Once the Supervisor computes the application state, it sends metrics to Monasca for displaying the performance:
Current percentage of the CPU allocated to the app.
Current performance.
Desired VS Current CPU time consumed.
CLI
The user specifies:
App specification in JSON.
QoS information.
Parameters to connect and configure the Supervisor.
Credentials for Marathon.
REST API
openstack
KeyStone
The Supervisor obtains the token for sending metrics to Monasca from OpenStack KeyStone.

Figure 3: Design of the architecture for the Marathon scenario.

minimum percentage of the CPU time according to a given deadline. As described above, the application specification must be modified to manage the Docker container, i.e., to run our component Executor. The modifications in the application specification consist of three parts. First, the Launcher obtains, from the application definition, all the information required for launching the Docker container. As the Executor will manage the Docker container, the Launcher removes the Docker container definition of the application specification. With this changes, the task launched by Marathon will be allocated the same resources as the original application but it will be embedded in a Mesos Native container instead of a Docker container. Finally, the Launcher changes the command of the original application specification for the command that runs the Executor with the information required for notifying Supervisor, for launching the container (obtained at the start of modification of the application specification), and the application UUID.

Once these changes are done, the Launcher submits through a REST call the application to the appropriate framework (Chronos or Marathon). Besides, for both scenarios, the Launcher sends through a REST call the relevant information to the Supervisor. The relevant information comprises the application name, the application UUID, the framework name, the duration of one application execution, the targeted QoS, the maximum overprogress percentage, and, in Chronos case, the number of iterations is also sent. The framework name can be Chronos or Marathon. The duration of one execution, in Chronos case, is the duration of the execution of one iteration, and in Marathon case, is the minimum CPU time that the developed mechanism must allocate before the deadline for completing the execution. In Chronos case, the targeted QoS is expressed as the deadline in timestamp format. In Marathon case, the targeted QoS is expressed as the available time frame for executing the application in seconds. This time interval is the time between the submission time and the deadline, which both are in timestamp format. The maximum overprogress percentage is the acceptable threshold above which an application is considered to be over progressing.

5.2. Monitoring stage

Once that the application is launched in the Mesos agent by the framework executor, the mechanism monitors the application to collect its performance for, if it is necessary, updating the resources assigned. As described in Section 5, this work considers two approaches for monitoring: passive and active.

The passive approach is used in the Chronos scenario. In this approach, monitoring is performed by the Supervisor. The Supervisor receives a message from the application (Docker container) each time an iteration finishes its execution. This message is the result of the modification done by the Launcher in the application specification (the Launcher puts a REST request in command field after the original command) described in Section 5.1. When this message arrives at the Supervisor, it starts the decision-making stage to meet the QoS agreed. In addition, the Supervisor decrements the remaining iterations of the application.

As the aim of the Marathon scenario is not to run a determined number of iterations, monitoring is doing during the execution of the application. For this purpose, the active approach is used in the Marathon scenario. As mentioned above, one of the first tasks of Executor is to notify the Supervisor using a REST request (the body of its message is formed by the application UUID and the timestamp of the start of the execution) that the application is started. Once the Supervisor receives this message, it starts to monitor periodically. As the Supervisor does not have access to the Mesos agents, it uses the metrics of the Docker container's performance from Monasca. The Monasca Agent collects these metrics for each container. It should be noted that a modified Docker plug-in for the Monasca Agent (available in the GitHub repository of this work [27]) is required to enable monitoring of restored containers from checkpoints. Whenever the Supervisor obtains the collected data from Monasca, it starts the decision-making stage to meet the QoS agreed. This procedure is periodically repeated until the Executor notifies that the application execution is finished to the Supervisor. The body of this notification message is formed by the application UUID and the timestamp of the end of the execution.

5.3. Decision-making stage

Once the data collection in monitoring stage is completed, the next step consists in determining the application performance state. As discussed in Section 2, there are three states for the application: *underprogress*, *ontime* and *overprogress*. The Supervisor uses the Algorithm 1 to determine the state of the application. This algorithm has the same input for both scenarios: the application performance percentage ($performance$), the over-progress threshold (threshold$_{overprogress}$) and under-progress threshold (threshold$_{underprogress}$). The threshold$_{underprogress}$ is set by default to 0.9, which means that an application is considered in the under-progress state when its performance is a maximum of 10% worse than the required performance. The threshold$_{overprogress}$ is the obtained using the Equation 1 with the maximum over-progress percentage (*max_overprogress*), which is a parameter sent by the Launcher to tune the Supervisor's configuration for each application launched. If *max_overprogress* is set to 0.2 by the user, the threshold$_{overprogress}$ is 1.2, which means that an application is considered in the over-progress state when its performance is a minimum of 20% better than the required performance.

$$threshold_{overprogress} = 1.0 + max\_overprogress \tag{1}$$

where *max_overprogress* is the maximum overprogress percentage sent by the Launcher in 5.1.

According to the scenario, the $performance$ is obtained in a different way. The user specifies the QoS to the Launcher. As it is described in Section 5.1, the Launcher specifies the target QoS in two formats: as a deadline in timestamp format and as a time frame in seconds. This action is done for facilitating the computation of the $performance$ to the Supervisor. In the Chronos scenario, as the mechanism guarantees that the application is executed a determined number of iterations, the QoS is a time limit to complete all the iterations. For example, the QoS of an application establishes that ten iterations of this application must be executed before 9 am (QoS is 9 am but in timestamp format). In the Marathon scenario, as the mechanism guarantees a predefined minimum percentage of the CPU time to the application according to a given deadline, the QoS is the time frame for allocating the predefined minimum percentage of the CPU time. For example, QoS is 300 seconds for an application that needs 275 seconds of CPU time (the duration of the application) for be executed time before the next five minutes.

Algorithm 1: Algorithm used by the Supervisor to obtain the application state

```
Input : performance, threshold$_{overprogress}$ and threshold$_{underprogress}$
Result: application_state
begin
    if $performance > threshold_{overprogress}$ :
        application_state = overprogress ;          /* Decrease allocated resources */
    else if $performance < threshold_{underprogress}$ :
        application_state = underprogress ;         /* Increase allocated resources */
    else:
        application_state = ontime ;                /* Nothing to do */
end
```

In case of the Chronos scenario, the decision-making stage is done when each iteration of the application finalizes its execution. Once an iteration completes its execution, the Supervisor obtains the $performance$ value using the Equation 2 for determining the application state using the Algorithm 1. This equation requires the estimated finalization time of all iterations, $t_{prediction}$, which is obtained using Equation 3.

$$performance = \frac{t_{prediction}}{t_{qos}} \tag{2}$$

where $t_{prediction}$ is the estimated finalization time of all iterations in timestamp format obtained using the Equation 3 and t_{qos} is the QoS agreed in timestamp format (obtained from the Launcher in Section 5.1).

$$t_{prediction} = t_{current} + rem\_iter * (d_{iteration} + d_{deployment}) \tag{3}$$

where $t_{current}$ is the current time in timestamp format, $d_{iteration}$ is the expected duration of one iteration execution (sent by the Launcher), $d_{deployment}$ is the upper bound value of the deployment time of the new iteration in the infrastructure (sent by the Launcher), and rem_iter is the number of remaining iterations.

In case of the Marathon scenario, the decision-making stage is periodically done (just before the monitoring stage). Once the Supervisor ends each monitoring stage, it immediately obtains the $performance$ at determined time t of the execution using the Equation 4. Once the $performance$ is obtained, the Supervisor uses the Algorithm 1 to determine the application state.

$$performance(t) = \frac{cputime_{current}(t)}{cputime_{desired}(t)} \tag{4}$$

where $cputime_{current}(t)$ is the real CPU time consumed at determined time t for the application (the obtention of this value is detailed below) and $cputime_{desired}(t)$ the expected CPU time at determined time t for meeting the QoS agreed.

As real progress information is not available, the Supervisor estimates it by computing the CPU time consumed on a given execution time $cputime_{current}(t)$. It can be computed from the information obtained by two queries to Monasca (*container.cpu.user_time* and *container.cpu.system_time*). These values correspond to the total user and system clock ticks consumed by the container in the working node where it is running. These values are transformed into seconds, dividing them by the clock ticks per second constant of the system.

The $cputime_{desired}(t)$ is the CPU time that the application would have to consume at determined time t for meeting the QoS agreed. The Supervisor uses the Equation 5 to calculate $cputime_{desired}(t)$ in seconds. As it is described above, in the Marathon scenario, the duration of the application execution ($d_{application}$) is the CPU time that the developed mechanism must allocate in the time frame agreed (t_{qos}). As t_{qos} is the available time frame for allocating $d_{application}$ seconds, it is necessary to calculate the percent of the time frame available (d_{qos}) that the application has consumed at a determined time t. This percentage is calculated with $\frac{t_{current}-t_{start}}{t_{qos}}$. Using this percentage and the CPU time that the application has to consume before to meet to QoS, $d_{application}$, the CPU time that the application would have to consume at determined time t can be calculated.

$$cputime_{desired}(t) = \begin{cases} (\frac{t_{current}-t_{start}}{t_{qos}}) * d_{application} & \text{if } t_{current} \leq t_{start} + t_{qos} \\ d_{application} & \text{otherwise} \end{cases} \tag{5}$$

where $t_{current}$ is the current time in timestamp format, t_{start} is the time when the application start its execution in timestamp format (sent by the Executor), $d_{application}$ is the CPU time in seconds that the developed mechanism must allocate in the time frame agreed (sent by the Launcher) and t_{qos} in seconds is the time frame for allocating $d_{application}$ seconds.

At this point of the architecture's flow, the Supervisor downloads the application specification for obtaining the last value of the share of CPU time that was assigned. Then, it sends through a REST call information about the application performance and the percentage of CPU time to Monasca for displaying it.

5.4. Adjustment

Once the Supervisor completes the decision-making stage, it determines, according to the application state, if the application must be resized. If the application state is *overprogress* or *underprogress*, the Supervisor varies the percentage of CPU time assigned to the application. It should be remembered that the Supervisor determines the scaling decision (increment or decrement the percentage of the assigned CPU time), but is the framework (Chronos or Marathon) executor who implements it. The mechanism implements the variation of the resources by changing and re-submitting the application specification to the framework by means of a REST API. In both scenarios, the framework halts and restarts the application, starting it on a (different or the same) working node.

In case of the Chronos scenario, the Supervisor modifies (increment or decrement the percentage of CPU time and ensure that the remaining iterations value of the application is correct) of the application specification, which was downloaded at the end of the decision-making stage. The amount of the percentage of the CPU time that it is incremented or decremented is set by default at the startup of the Supervisor. After several tests, it has been observed that incrementing or decrementing by 0.4 the percentage of the CPU time offers the best results (1.0 corresponds with the all of percentage of CPU time for one CPU). Once the modification is done in the application specification, the Supervisor sends it using a POST request to the REST API of the Chronos scheduler. Therefore, the new iteration of the application will be executed with the new value of the percentage of CPU time. As the adjustment stage in Chronos scenario is done between iterations, is not necessary to preserve the state of the execution.

The adjustment stage in the Marathon scenario is more complex than in the Chronos scenario because the mechanism resizes the allocated resources while the application is running. As it is described above, the Executor is in charge of managing the Docker container. According to the state of the application obtained in Section 5.3, the Supervisor modifies the percentage of CPU time assigned of the application specification downloaded at the end of the decision-making stage. When the Supervisor re-submits the modified application specification with the new value of the percentage of CPU time using a PUT request to the REST API of the Marathon scheduler, the Marathon executor (on the Mesos agent) sends a termination signal to the process that is running the application before removing it. This process is the Executor, which is running embedded in a Mesos container. The termination signal sent by the Marathon executor is captured by the Executor, which creates a checkpoint of the running container and stores it in an accessible location by all worker nodes (Mesos agents) of the infrastructures. As soon as the application with the new specification of the percentage of CPU time is running, the Executor restores the checkpoint created previously to resume the execution at the same point of the termination signal was captured.

6. Results and discussion

In this work, a Mesos cluster has been set up. It comprises five Mesos agents, version 1.2.0, a Marathon (version 1.4.3) and a Chronos (version 2.1.0) framework. Monasca (version 1.6.1) is used for monitoring. Docker (version 17.05.0-ce) running under the experimental mode is used as container engine and CRIU (version 2.6) is used for checkpointing. All machines are virtual machines (VM) deployed on top of an OpenNebula IaaS [28] with 2 CPUs and 4 GB of memory RAM, and they are connected through a private network. Only the front-end has a public IP and all the main services have been deployed there. Figure 4 shows an architecture diagram of the solution. It is important to outline that we configure Mesos to enable running jobs to use all the free CPU cycles available in the node, beyond the allocation of resources given by Mesos. Therefore, the CPU allocation will only be limited if other applications are competing in the same node, considering the resource share distribution requested in the application deployment. This feature is desirable in our system, as applications will make use of idle CPU cycles from powered-on resources, over progressing and tentatively releasing more resources in the future for other applications closer to the deadline.

Deployment has been done using the Infrastructure Manager [29] Orchestrator and using the Elastic Cloud Computing Cluster (EC3) [30] client recipes from project EUBra-BIGSEA [31]. This environment is easy to reproduce as the recipes are publicly available in the Docker container *eubrabigsea/ec3client* available in [32]. A Network File System (NFS) is used along all the nodes to share a directory where the CRIU checkpoint images will be stored. No

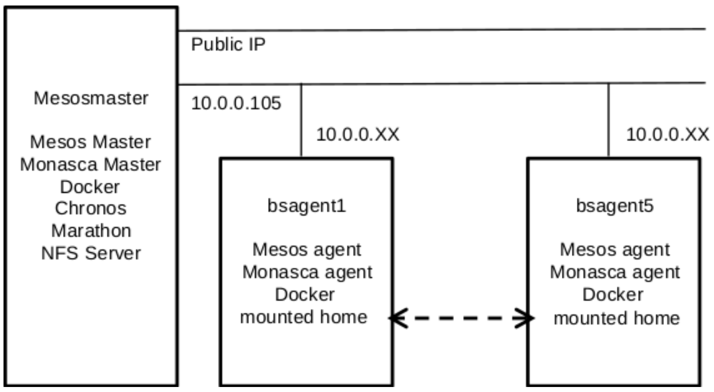


Figure 4: Infrastructure and software deployment.

changes have been applied to any of the above components, to facilitate migrating to new versions. We rely on the REST API of Marathon [33], Chronos [34] and Monasca [25] and develop the necessary services at the front end to deal with the application registration and scaling decisions. The code is done in Python and it is publicly available in GitHub [27].

Many applications may benefit from the proposed system, but it is specially useful for applications with high CPU usage. For this reason, the benchmark LINPACK [35] is selected for testing the system. As the worker nodes have 2 CPUs, the tests use a parallel version of LINPACK benchmark (source code is available in [36]). The parallel version was optimized using OpenMP [37]. Source code of the parallel version and binary executable are available in a Docker image [38].

We evaluated the overhead of the monitoring system by registering the CPU usage (both user and system). The CPU usage of the monitoring probes and services lied below 10% (mostly below 5%) of a single CPU core. As the monitoring service is needed to register other variables of the system, the monitoring system cannot be entirely switched off. Therefore, this overhead is an upper bound of the effect of introducing the mechanism proposed in the article.

6.1. Use case 1: Chronos

The Chronos use case involves the execution of six applications. All applications are the LINPACK benchmark (previously mentioned). Each application will require to execute a different number of iterations and different matrix input size. Table 1 shows the information about these applications. In this table, each row corresponds to a different application instance with different requirements in memory and execution time. The column *iteration* denotes the number of Chronos iterations executed per application. The columns *Av. Time* denote the average execution time of a single iteration with 1 or 2 CPUs respectively. The execution time depends on the size of the problem, shown in the last column (*Matrix dim.*). The use of VMs mainly causes the speedup of the multithreaded application to be very reduced. In the Chronos scenario, the *App duration* column is the expected duration of an iteration in seconds. In the Marathon scenario, the *App duration* column is the CPU time in seconds that the mechanism must allocate in

Table 1: Information about QoS and averaged execution completion time of the applications (time in seconds).

Name	Iterations	Av. Time 1 iter - 1 CPU	Av. Time 1 iter 2 CPUs	App Duration	Prediction 1 CPU	Prediction 2 CPUs	Results	Matrix Dim.
openmpLinpack1	4	480.4	362.2	400	2101.6	1628.8	-328	6000
openmpLinpack2	4	480.4	362.2	400	2101.6	1628.8	-340	6000
openmpLinpack3	13	162.75	113.2	130	2700.75	2056.6	36	4000
openmpLinpack4	11	162.75	113.2	130	2285.25	1740.2	-105	4000
openmpLinpack5	12	162.75	113.2	130	2493	1898.4	133	4000
openmpLinpack6	14	92.36	66.32	80	1923.04	1558.48	-290	3500

the given time frame. It is important to point out that, instead of asking for specific resource allocation, applications define a feasible time frame in which they will require more than 1 CPU (and less than 2 CPUs, which is the per-node limit). The *Prediction* columns depicts the expected total execution time. This value is calculated using the maximum deployment time for each iteration (this value is set to 45 seconds) and the expected duration of one iteration (the column *App duration*). Finally, column *Results* is the difference between the QoS agreed and the real execution time of each application in the experiment. The negative values of the column *Results* show that the application ended before the target QoS. Figure 5 depicts the difference in seconds between the prediction of the execution finalization of all iterations and the QoS agreed (time limit to finalise all iterations).

This use case considers a highly-loaded infrastructure. If there were always resources for the application to run in the best conditions, vertical elasticity would not be needed. In this use case, all the iterations from each application have a similar completion time. Thus, it is reasonable conclude that the completion time of all executions will grow linearly.

Therefore, it is possible to compute an estimation of the finalization time using the averaged completion time of each execution (the column *App Duration*) described in Table 1 and the deployment time, which it is set to 45 seconds in the infrastructure used for the experiment. The deployment time is the maximum time observed deploying applications in the Chronos framework. The same table shows also these predictions and the results obtained in the test.

If the mechanism allocates 1 CPU for each application, the targeted QoS is not achieved. Due to the fact that the infrastructure provides five node with 2 CPUs each and there are six applications, at least one of them will not reach the QoS agreed. Furthermore, according to the column *Prediction*, two applications (*openmpLinpack3* and *openmpLinpack5*) cannot finishing on time. Therefore, the best possible result for the test is to complete four out of six applications on time , which was successfully achieved by the system.

Finally, we can state that no overhead is introduced due to the rescheduling of the Docker containers in Chronos, as the change of the allocated resources is done between iterations. Chronos always reschedules each iteration as it were a new job. Therefore, changing the allocation of resources does not make any difference.

Figure 5 depicts the difference between prediction of finalization time and the application time limit. The Y-axis

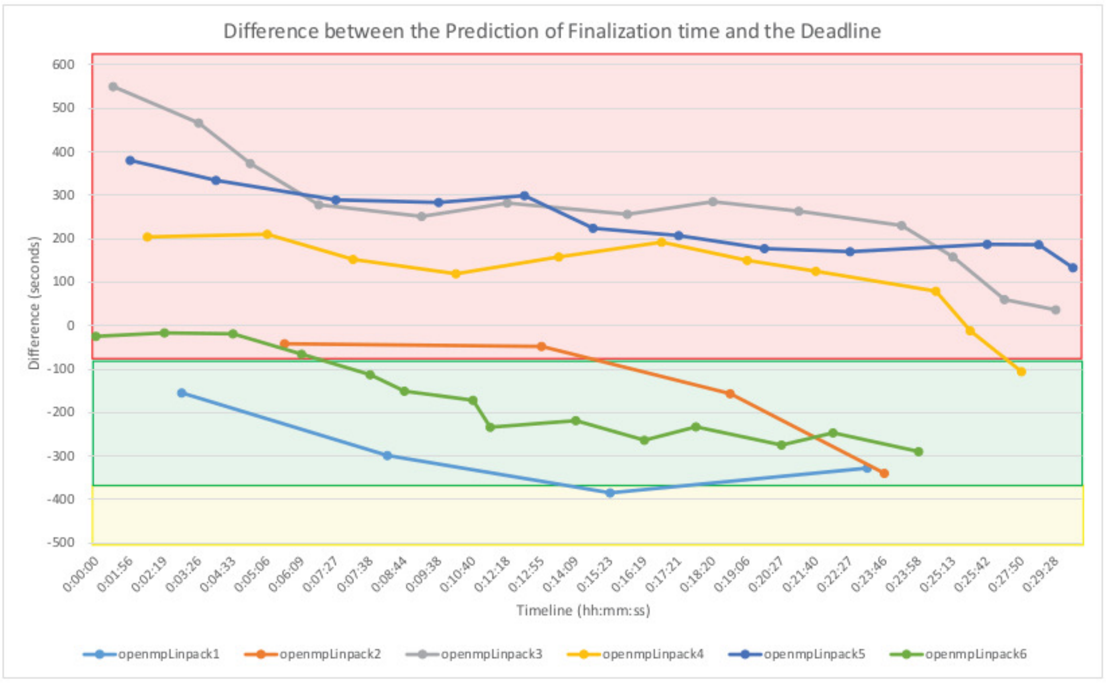
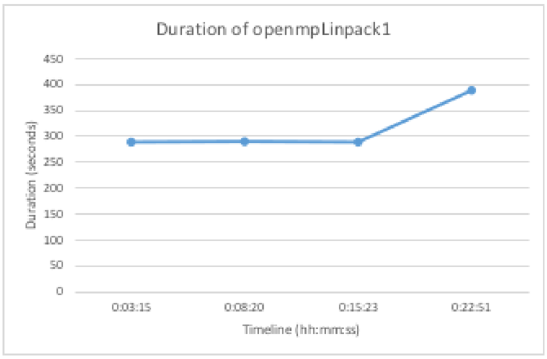


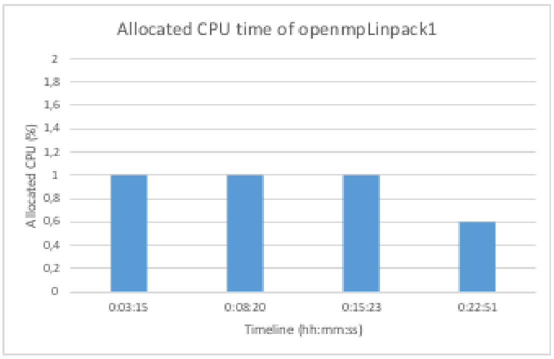
Figure 5: Difference between prediction of finalization time and the application deadline for use case 1.

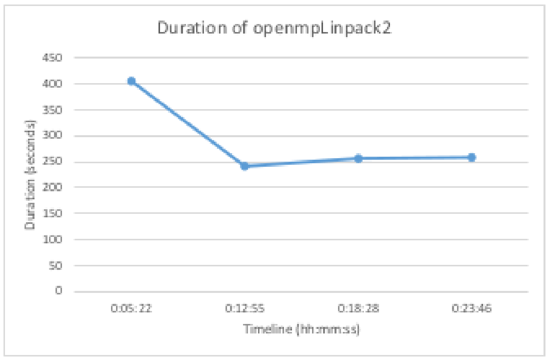
and X-axis represent, respectively, the difference in seconds and the time when the sample was collected in CEST time. The Y-axis colored ranges in represent the application state: red, green and yellow for, respectively, *underprogress*, *ontime* and *overprogress*.

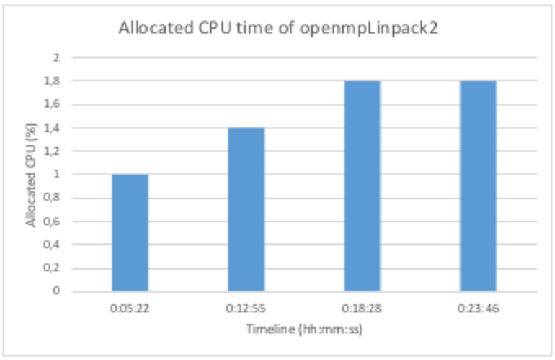
The perfect case for Figure 5 would be that an application that begins far from the zero value (in the Y axis) terminates in the zero. If an application begins above the zero mark, means that is into underprogress state. If it is more than the defined threshold below the zero mark, it is into overprogress state. Otherwise, it is into ontime state. If an application concludes its execution at the zero mark means that the application has finished its execution just at the moment before breaking the targeted QoS. Thus, if an application has a downwards tendency it tendency means that is improving its performance. This decreasing tendency is common in applications that have been in underprogress state because the mechanisms had increased their allocated resources. Using the same reasoning, when an application has an increasing tendency means that the application is running slower than what is needed to meet the targeted QoS.

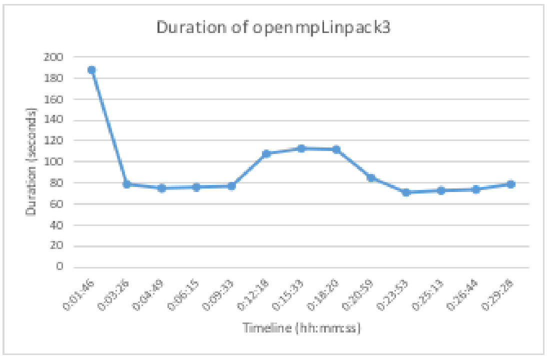
From Figure 5 it can be seen that the red and green region applications have a decreasing tendency. In case of the applications of the red region, this tendency is consequence of the increase of allocated resources during the executions. In case of the green region applications, the applications have assigned more resources than needed for meeting the target QoS. In fact, the performance of the application *openmplinpack1* is so high that the mechanism can reduce its allocated resources. It should be pointed out that, the resizing of the application *openmplinpack1* causes the change of its tendency.

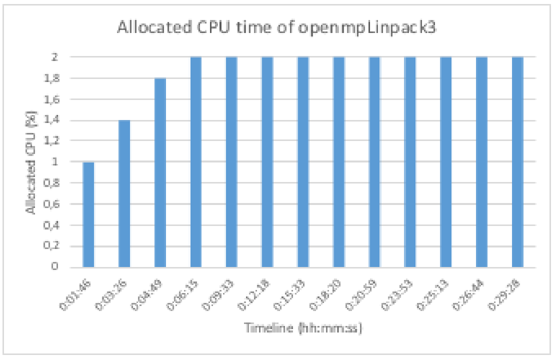

Duration of openmpLinpack1
Duration (seconds)
Timeline (hh:mm:ss)
0:03:15
0:08:20
0:15:23
0:22:51

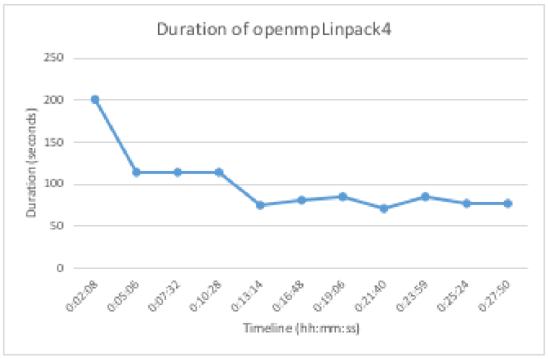

Allocated CPU time of openmpLinpack1
Allocated CPU (%)
Timeline (hh:mm:ss)
0:03:15
0:08:20
0:15:23
0:22:51

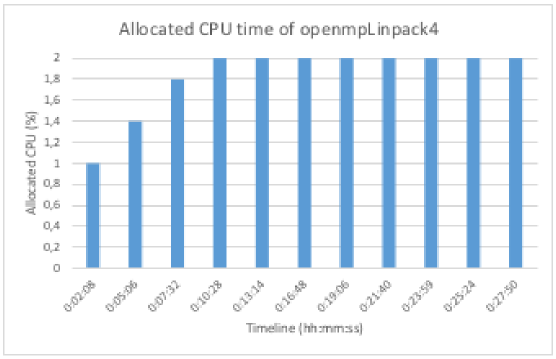

Duration of openmpLinpack2
Duration (seconds)
Timeline (hh:mm:ss)
0:05:22
0:12:55
0:18:28
0:23:46

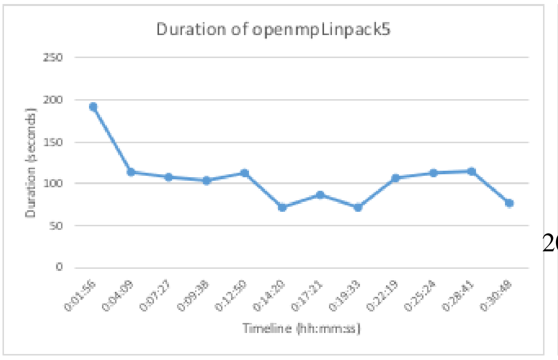

Allocated CPU time of openmpLinpack2
Allocated CPU (%)
Timeline (hh:mm:ss)
0:05:22
0:12:55
0:18:28
0:23:46

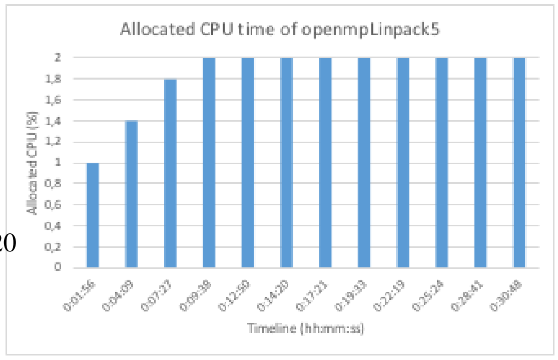

Duration of openmpLinpack3
Duration (seconds)
Timeline (hh:mm:ss)
0:01:46
0:03:26
0:04:49
0:06:15
0:09:33
0:12:18
0:15:33
0:18:20
0:20:59
0:23:53
0:25:13
0:26:44
0:29:28


Allocated CPU time of openmpLinpack3
Allocated CPU (%)
Timeline (hh:mm:ss)
0:01:46
0:03:26
0:04:49
0:06:15
0:09:33
0:12:18
0:15:33
0:18:20
0:20:59
0:23:53
0:25:13
0:26:44
0:29:28


Duration of openmpLinpack4
Duration (seconds)
Timeline (hh:mm:ss)
0:02:08
0:05:06
0:07:32
0:10:28
0:13:14
0:16:48
0:19:06
0:21:40
0:23:59
0:25:24
0:27:50


Allocated CPU time of openmpLinpack4
Allocated CPU (%)
Timeline (hh:mm:ss)
0:02:08
0:05:06
0:07:32
0:10:28
0:13:14
0:16:48
0:19:06
0:21:40
0:23:59
0:25:24
0:27:50


Duration of openmpLinpack5
Duration (seconds)
Timeline (hh:mm:ss)
0:01:56
0:04:09
0:07:27
0:09:38
0:12:50
0:14:20
0:17:21
0:19:33
0:22:19
0:25:24
0:28:41
0:30:48


Allocated CPU time of openmpLinpack5
Allocated CPU (%)
Timeline (hh:mm:ss)
0:01:56
0:04:09
0:07:27
0:09:38
0:12:50
0:14:20
0:17:21
0:19:33
0:22:19
0:25:24
0:28:41
0:30:48

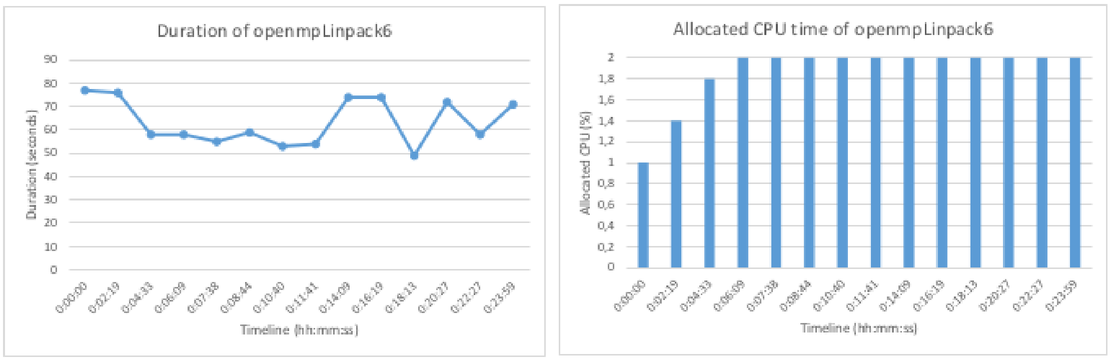


Figure 6: Shared CPU time and duration of completion time for each application.

Figure 6 shows two graphs for each application. The points in these graphs are collected when an iteration is completed (X-axis) in CEST time. The graphs on the right show the share of the CPU time assigned for the application and the graphs on the left shows the duration of the execution. As it can be seen in the figure, all applications started with only one CPU allocated, and the vertical elasticity system varied this resource allocation to correct the evolution of the execution. This fact explains why most of the applications started to be underperforming and progressively improved the performance to fulfil the expected deadline (see figure 5). From the results obtained, it seems that the estimation for applications 3, 4 and 5 was too optimistic, which led to a violation of the deadline in two of them.

In Figure 6 it can be seen that sometimes there is not a correlation between the allocated CPU and the job duration. Docker does not prevents processes from consuming free CPU cycles, it only limits CPU shares when competing with other applications. Therefore, the CPU time consumed for each container can be higher than the CPU time assigned for it.

6.2. *Use case 2: Marathon*

Before starting the experiment, we want to analyse the overhead caused by the use of CRIU. We performed a test consisting of the execution of a parallel Linpack job within a Docker container. The job was checkpointed and immediately restarted to evaluate the following variables:

- The total runtime overhead caused by checkpointing, stopping and restarting a container compared to a run without checkpointing.
- The time requested to create the checkpoint both in an NFS and a local folder.
- The size of the checkpoint image for the experiment job. We repeated the execution of the multithread Linpack test for a problem size of dimension 1000.

We repeated the experiment five times, executing one checkpointing and restart per case at different application stages. Table 2 presents the results (comparing to the execution without stopping, checkpointing and restarting).

Table 2: Execution of 5 instances of parallel Linpack with (2) and without (1) checkpointing interruption. In case of (2), the checkpoint was stored in an NFS directory. The checkpointing time is the time taken to create the checkpoint image, which takes place in parallel to the execution (it should not be added or subtracted to the Linpack execution time). Checkpointing in an NFS (3) and a local directory (4) have been measured. Time is in minutes:seconds.

	Execution time without checkp. (1)	Execution time with one checkp. (2)	Checkp. time in NFS (3)	Checkp. time in local (4)
#1	06:29,0	06:45,0	00:30,4	0:01,3
#2	06:32,3	06:44,3	00:29,8	0:01,4
#3	06:33,5	06:33,7	00:46,2	0:01,7
#4	06:33,3	06:30,1	00:56,4	0:01,6
#5	06:25,1	06:33,8	00:56,8	0:01,4
Mean	06:30,7	0:06:37	0:00:44	0:01,5
Std. Dev.	00:03,6	0:00:07	0:00:13	0:00,16

This table shows that the checkpointing only increases (in average) the execution time in 6 seconds. The checkpointing process only considers the layers and memory that are modified in the Docker container, producing a small disk footprint (between 6 and 11 MB). Despite of the fact that the checkpointing in the NFS system takes between 30 and 60 seconds (on the order of 1 second in the case of a local filesystem), the downtime of the application is minimal. Starting the application with the checkpoint takes a negligible cost.

Once the overhead of checkpointing has been analysed, we execute the QoS experiment for the Marathon use case. In this use case, the system tries to guarantee the allocation of a minimum share of the CPU time to the application during a given interval in a congested infrastructure. For this test, the minimum share of the CPU time used (400 seconds) is a value between the duration in seconds of the application using two CPUs (360 seconds) and the given interval for completion time corresponds with the averaged application duration using one CPU (480 seconds).

The test is finalised successfully because the execution is ended in 393 seconds. This test is very interesting because the application enters into the three possible states. This can be seen in Figure 7 where the Y-axis and X-axis represent, respectively, the progress ratio percentage and the time when the sample was collected (in CEST). The Y-axis colored ranges denote the application status: red, green and yellow for, respectively, *underprogress*, *ontime* and *overprogress*.

Computing the real overhead in this case is difficult to evaluate, as several variables are affecting the performance. The previous experiment shown in table 2 shows an absolute overhead of 6 seconds per checkpoint. The experiment managed to execute a Linpack test in 393 seconds. The average execution in an isolated two-core working node is around 362 and 480 seconds in a single-core working node, as shown in table 1. The test incurred in two checkpoints and restarts, thus leading to three different execution intervals. Although the framework only requested 1 CPU, the job will try to use all the free resources temporarily unless other jobs run in the system. This has been the case for the middle interval (Figure 9 depicts the CPU allocation). From the execution results, we observe that the resources allocated during the first and last intervals (68% of the time) were between 1 and 2 CPUs. During the middle interval (32% of the total execution), the resources allocated efficiently were between 0.6 and 1 CPU. The overhead in the system will be lower than 31 seconds (the difference between the observed execution time and the average time with 2 cores), and higher than the 12 seconds required to perform the 2 checkpoints and restarts.

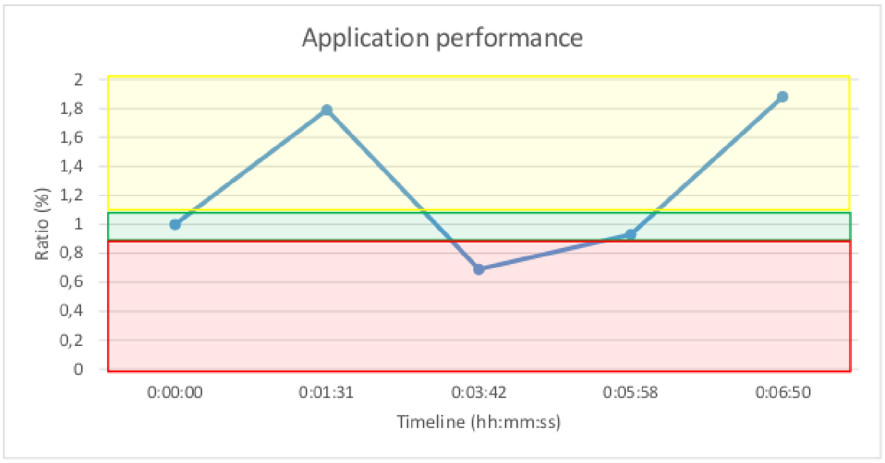


Figure 7: Progress ratio used for compute the application state of the application.

Figure 8 depicts the comparison of CPU time consumed and the CPU time that the system predicts that should be consumed at the moment of which the samples are collected in seconds. These values are used in Equation 4 to calculate the performance of the application, whose values are shown in Figure 7.

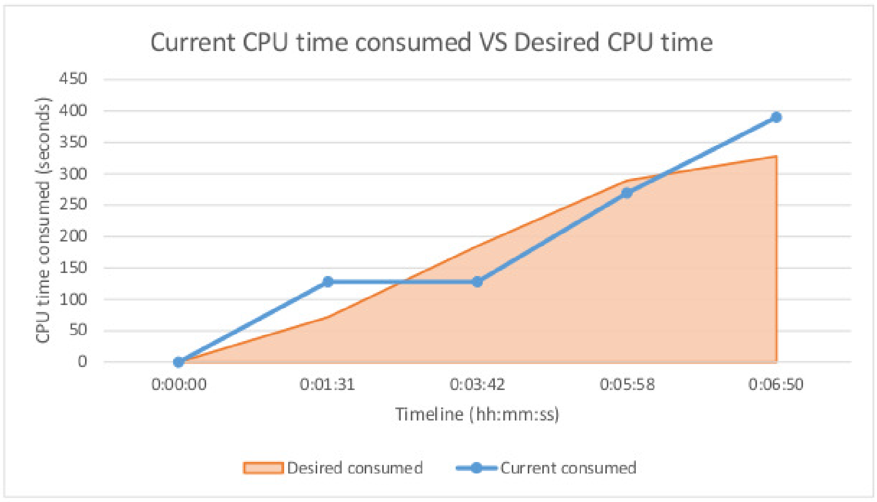


Figure 8: Comparison between the current CPU time consumed by the container and the CPU time that should be consumed (both in seconds).

Figure 9 shows the share of CPU time allocated to the application when the samples are collected. According to application state obtained from ratio progress, the share of CPU time is modified.

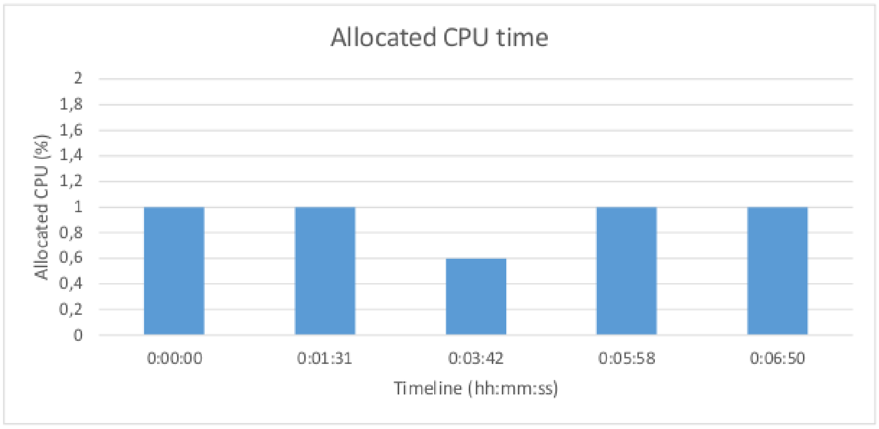


Figure 9: Shared CPU time of the application.

The execution trace can be seen from the tree graphs. The first sample corresponds to the start of the application. The next sample is collected when the application is in *overprogress* state, thus a decrement of share of CPU time is needed. As it can be seen in Figure 8, the two following samples of current time consumed are indicating that the application performance is under the expected but, in case of third sample, it is not low enough to trigger an action, thus the state is *ontime*. For this reason, in Figure 9 the share of CPU is only incremented once. The last sample is obtained when the execution is completed and, as it can be seen in Figures 8 and 7, the application ends fulfilling the targeted quality of service with 87 seconds of margin.

7. Conclusions

Vertical elasticity is the only solution for dynamically speeding up sequential or multicore applications which could not benefit from increasing the number of instances. Vertical elasticity in cloud system usually implies rebooting the services, as hypervisors can resize Virtual Machines only when they are powered off. Memory ballooning and CPU CAP update can be done dynamically without rebooting VMs, but they require privileged access to the hypervisors, which is not the case in most IaaS. The use of container-based applications to dynamically change the resource allocation is a transparent and low-impact technique that can be done in userspace without compromising other's execution.

The article demonstrates that the combination of checkpointing and Mesos Frameworks can be used to keep the application progress and even to benefit from the service redeployment capabilities of Marathon and Mesos to find the rightmost resources. This technique has been applied for both periodic jobs (Chronos) and long-run jobs (Marathon) successfully on a busy environment in which workload is dynamic and competitive. The work aimed at ensuring the desired QoS, although it can be used for other policies in the serverless paradigm, enabling the execution and retake of functions.

References

[1] A. Mesos, Apache Mesos website, `http://mesos.apache.org/`, [Online; accessed July-2017] (2017).

[2] Marathon, Marathon website, `http://mesosphere.github.io/marathon/`, [Online; accessed July-2017] (2017).

[3] Chronos, Chronos website, `https://mesos.github.io/chronos/`, [Online; accessed July-2017] (2017).

[4] M. Sedaghat, F. Hernandez-Rodriguez, E. Elmroth, A virtual machine re-packing approach to the horizontal vs. vertical elasticity trade-off for cloud autoscaling, in: Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference on - CAC '13, 2013, p. 1. `doi:10.1145/2494621.2494628`.
URL `http://dl.acm.org/citation.cfm?doid=2494621.2494628`

[5] T. Lorido-Botran, J. Miguel-Alonso, J. A. Lozano, A Review of Auto-scaling Techniques for Elastic Applications in Cloud Environments, Journal of Grid Computing 12 (4) (2014) 559–592. `doi:10.1007/s10723-014-9314-7`.

[6] Linux Crontab manual website, `http://man7.org/linux/man-pages/man5/crontab.5.html`, [Online; accessed September-2018] (2017).

[7] Barik, R.K. and Lenka, R.K. and Rao, K.R. and Ghose, D., Performance analysis of virtual machines and containers in cloud computing, Proceeding - IEEE International Conference on Computing, Communication and Automation, ICCCA 2016`doi:{10.1109/CCAA.2016.7813925}`.

[8] W. Felter, A. Ferreira, R. Rajamony, J. Rubio, An updated performance comparison of virtual machines and Linux containers, in: ISPASS 2015 - IEEE International Symposium on Performance Analysis of Systems and Software, 2015. `doi:10.1109/ISPASS.2015.7095802`.

[9] Z. Shen, S. Subbiah, X. Gu, J. Wilkes, Cloudscale: Elastic resource scaling for multi-tenant cloud systems, in: Proceedings of the 2Nd ACM Symposium on Cloud Computing, SOCC '11, ACM, New York, NY, USA, 2011, pp. 5:1–5:14. `doi:10.1145/2038916.2038921`.
URL `http://doi.acm.org/10.1145/2038916.2038921`

[10] S. Farokhi, P. Jamshidi, E. Bayuh Lakew, I. Brandic, E. Elmroth, A hybrid cloud controller for vertical memory elasticity: A control-theoretic approach, Future Generation Computer Systems 65 (2016) 57–72. `doi:10.1016/j.future.2016.05.028`.

[11] G. Moltó, M. Caballer, C. de Alfonso, Automatic memory-based vertical elasticity and oversubscription on cloud platforms, Future Gener. Comput. Syst. 56 (C) (2016) 1–10. doi:10.1016/j.future.2015.10.002.
URL https://doi.org/10.1016/j.future.2015.10.002

[12] F. Paraiso, S. Challita, Y. Al-Dhuraibi, P. Merle, Model-Driven Management of Docker Containers, in: 2016 IEEE 9th International Conference on Cloud Computing (CLOUD), IEEE, 2016, pp. 718–725. doi:10.1109/CLOUD.2016.0100.
URL http://ieeexplore.ieee.org/document/7820337/

[13] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, P. Merle, Autonomic Vertical Elasticity of Docker Containers with ELASTICDOCKER, in: IEEE International Conference on Cloud Computing, CLOUD, Vol. 2017-June, 2017, pp. 472–479. doi:10.1109/CLOUD.2017.67.

[14] DO/OS, Autoscaling with Marathon, https://dcos.io/docs/1.7/usage/tutorials/autoscaling/, [Online; accessed July-2017] (2017).

[15] C. Zheng, B. Tovar, D. Thain, Deploying high throughput scientific workflows on container schedulers with makeflow and mesos, in: Proceedings - 2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2017, 2017, pp. 130–139. doi:10.1109/CCGRID.2017.9.

[16] Virtuozzo, CRIU website, https://criu.org/Main_Page, [Online; accessed July-2017] (2011).

[17] Virtuozzo, OpenVZ website, https://openvz.org/Main_Page, [Online; accessed July-2017] (2005).

[18] L. C. (LXC), Linux Containers website, https://linuxcontainers.org/, [Online; accessed July-2017] (2008).

[19] B. Hindman, A. Konwinski, A. Platform, F.-G. Resource, M. Zaharia, Mesos: A platform for fine-grained resource sharing in the data center, Proceedings of the ... (2011) 32doi:10.1109/TIM.2009.2038002.
URL http://static.usenix.org/events/nsdi11/tech/full{_}papers/Hindman{_}new.pdf

[20] Apache Mesos, OpenMP website, http://mesos.apache.org/documentation/latest/frameworks/, [Online; accessed May-2018] (2017).

[21] OpenStack, OpenStack Monasca website, http://monasca.io/, [Online; accessed July-2017] (2014).

[22] O. Source, OpenStack Monasca website, https://www.openstack.org/, [Online; accessed July-2017] (2010).

[23] OpenStack, Github of Monasca Agent, https://github.com/openstack/monasca-agent, [Online; accessed July-2017] (2015).

[24] OpenStack, OpenStack KeyStone website, https://wiki.openstack.org/wiki/Keystone, [Online; accessed July-2017] (2014).

[25] OpenStack, OpenStack Monasca REST API, https://github.com/openstack/monasca-api/blob/master/docs/monasca-api-spec.md, [Online; accessed July-2017] (2014).

[26] OpenStack, Monasca Plugins, https://github.com/openstack/monasca-agent/blob/master/docs/Plugins.md, [Online; accessed July-2017] (2015).

[27] Sergio López Huguet, Source code of the project, https://github.com/eubr-bigsea/vertical_elasticity, [Online; accessed July-2017] (2017).

[28] OpenStack, OpenNebula Project, https://opennebula.org/, [Online; accessed July-2017] (2005).

[29] M. Caballer, I. Blanquer, G. Moltó, C. de Alfonso, Dynamic Management of Virtual Infrastructures, Journal of Grid Computing (2014) 53–70doi:10.1007/s10723-014-9296-5.

[30] Miguel Caballer, Carlos De Alfonso, Fernando Alvarruiz and Germán Moltó, Ec3: Elastic cloud computing cluster, Journal of Computer and System Sciences 79 (8) (2013) 1341–1351. doi:10.1016/j.jcss.2013.06.005.
URL http://dx.doi.org/10.1016/j.jcss.2013.06.005

[31] EUBra-BigSEA, EUBra-BigSEA website, http://www.eubra-bigsea.eu, [Online; accessed July-2017] (2017).

[32] E.-B. project, EC3 client Docker Image, https://hub.docker.com/r/eubrabigsea/ec3client/, [Online; accessed July-2017] (2017).

[33] Marathon, Marathon REST API documentation, https://mesosphere.github.io/marathon/docs/rest-api.html, [Online; accessed July-2017] (2017).

[34] Chronos, Chronos REST API documentation, https://mesos.github.io/chronos/docs/api.html, [Online; accessed July-2017] (2017).

[35] Jack J. Dongarra and Piotr Luszczek and Antoine Petitet, The LINPACK benchmark: Past, present, and future. Concurrency and Computation: Practice and Experience, Concurrency and Computation: Practice and Experience 15 (2003) 2003.

[36] John Burkardt, Sequential and OpenMP versions of Linpack Solver, `https://people.sc.fsu.edu/~jburkardt/c_src/sgefa_openmp/sgefa_openmp.html`, [Online; accessed July-2017] (2017).

[37] OpenMP, OpenMP website, `http://www.openmp.org/`, [Online; accessed July-2017] (2017).

[38] Sergio López Huguet, Linpack parallelized with OpenMP Docker Image, `https://hub.docker.com/r/serlophug/openmplinpack/`, [Online; accessed July-2017] (2017).