Cluster Computing
https://doi.org/10.1007/s10586-018-02890-1

A study on performance measures for auto-scaling CPU-intensive containerized applications

Emiliano Casalicchio[1,2]

Received: 9 April 2018 / Revised: 10 September 2018 / Accepted: 19 December 2018

Abstract
Autoscaling of containers can leverage performance measures from the different layers of the computational stack. This paper investigate the problem of selecting the most appropriate performance measure to activate auto-scaling actions aiming at guaranteeing QoS constraints. First, the correlation between absolute and relative usage measures and how a resource allocation decision can be influenced by them is analyzed in different workload scenarios. Absolute and relative measures could assume quite different values. The former account for the actual utilization of resources in the host system, while the latter account for the share that each container has of the resources used. Then, the performance of a variant of Kubernetes' auto-scaling algorithm, that transparently uses the absolute usage measures to scale-in/out containers, is evaluated through a wide set of experiments. Finally, a detailed analysis of the state-of-the-art is presented.

Keywords Autonomic computing · Auto-scaling · Docker · Container · Kubernetes · Performance evaluation · Correlation

1 Introduction

A container (e.g. Docker [24]) and LXC [18]) is a software environment where one can install an application component (the so called microservice) or an application and all the library dependencies, the binaries, and a basic configuration needed for the execution. Containers provide a higher level of abstraction for the application life-cycle management and potentially may solve many distributed application challenges [9], e.g. portability and performance overhead. With containers, a microservice or an application can be executed on any platform running a container engine [23]. Containers are lightweight and introduce lower overhead compared to Virtual Machines (VMs) [11, 14, 22, 25]. Those are some of the reasons that make the cloud computing industry to adopt container technologies and to contribute to their evolution [7, 8, 33].

✉ Emiliano Casalicchio
emiliano.casalicchio@uniroma1.it

[1] Department of Computer Science and Engineering, Blekinge Institute of Technology, Karlskrona, Sweden

[2] Department of Computer Science, Sapienza University of Rome, Rome, Italy

Cloud service providers today offer container-based services and container development platforms [13]: Google container engine, Amazon Elastic Container Service and Microsoft Azure Container Service are examples of widely used platforms. Containers are also adopted in HPC (e.g. [37]) and to deploy large scale big data applications, requiring high elasticity in managing a very large amount of concurrent components (e.g. [15, 28, 36]).

The use of containers as base technology for deploying large-scale applications opens many challenges in the area of resource management at run-time [9, 29]. Many container orchestration frameworks are available, for example Kubernets [8], Docker Swarm, Mesosphere Marathon, Cloudify. This paper focuses on container auto-scaling mechanisms. First, the correlation between absolute and relative usage measures [11] and how a resource allocation decision can be influenced by them is analyzed in different workload scenarios. Absolute and relative measures could assume quite different values [11]. The former account for the actual utilization of resources in the host system (e.g. virtual machine or phys- ical server), while the latter account for the share that each container has of the resources used. Then, the performance of a variant of Kubernetes' auto-scaling algorithm, that transparently uses

Published online: 05 January 2019

the absolute usage measures to scale-in/out containers, is evaluated through a wide set of experiments.

Relative usage measures are today adopted by container orchestration frameworks like Kubernetes. For example, the *Kubernetes Horizontal Pod Auto-scaling* (KHPA) algorithm uses relative CPU utilization to trigger the scaling actions and to estimate the number of containers to be deployed to keep the resource utilization below a specified threshold value. Relative measures allow to easily configure horizontal scaling thresholds and the sharing of resources among different containers (e.g., defining usage quotas). However, as demonstrated in this paper (cf. Sect. 2), relative usage measures underestimate the required capacity, hence are not appropriate to determine the amount of resources needed to satisfy service level objectives, like response time.

In a previous work of the author [10] was modelled the correlation between relative and absolute metrics analyzing data obtained from measurements on a real system. Moreover, a new auto-scaling algorithm based on absolute metrics (named KHPA-A) was proposed. KHPA-A was conceived to be plugged into the Kubernetes controller and to makes the concept of absolute metrics transparent to the users, letting them to use the more intuitive concept of relative metrics adopted by KHPA. KHPA-A relies on an absolute-relative CPU utilization correlation model to predict the absolute metric values and to determine, more accurately, the number of containers to be deployed.

This paper extends the set of experiments in [10] to assess the performance of KHPA-A versus KHPA in a more realistic scenario. Specifically, while in [10] a single container was used to stress the CPU, in this paper a concurrent workload (multiple instances of containers) is executed on the same host to emulate the contention of the virtual/physical resources. Therefore, new correlation models are determined and the advantage of using KHPA-A versus KHPA is confirmed in the new scenarios. This performance study shows that the use of absolute metrics to trigger the container scaling actions and to dimension the appropriate number of Pods allows to properly control the application response time and, eventually, to keep it below thresholds specified by service level objectives (cf. Sect 5.2.3). The analysis presented is valid for high loaded servers (absolute CPU utilization higher than 75%) and shows that: for the single instance workload, the responce time obtained with the KHPA is a factor between 1.5 and 2 higher than the response time obtainable with the KHPA-A algorithm; for the concurrent workload, the responce time obtained with the KHPA is between 2 and 3 order of magnitude higher than the response time obtainable with the KHPA-A algorithm; the KHPA-A algorithm always provides performance comparable with the expected per-Pod CPU utilization and response time. Although the focus of the paper is only on CPU intensive workload and on Kubernetes' auto-scaling algorithms, the results of this work could be considered a guideline when implementing any containers' auto-scaling algorithm.

Finally, the paper's contribution is positioned in the context of the state-of-the-art in performance evaluation and autoscaling of containers.

The paper is organized as in what follow. Section 2 introduces the concept of relative and absolute measures, it describes how the horizontal containers auto-scaling works in Kubernetes and it provides the motivating example for this study. Section 3 is devoted to the container workload characterization in term of CPU utilization. A workload model is presented both for relative and absolute CPU utilization. The models of the correlation between relative and absolute CPU utilization are described in Sect. 4. The KHPA-A algorithm is presented in Sect. 5, along with the performance evaluation results. An extensive review of the state-of-the-art is presented in Sect. 6. Finally, Sect. 7 gives the concluding remarks.

2 Motivating example

2.1 Relative and absolute measures

Relative are those performance measures which values are based on the data collected from the `/cgroup` virtual file system using tools like `docker stats` or `cAdvisor`. For example, in Docker, the relative CPU utilization measures the share of CPU used by a container with respect to the other containers. Relative CPU utilization is reported as percent of total host capacity. For example, given two containers each using as much CPU as they can, each allocated the same CPU shares by Docker, then the `docker stats` command for each would register 50% utilization, though in practice their CPU resources will be fully utilized.

Absolute performance measures report about the cumulative activity counters in the operating system. For example, the absolute CPU utilization reports the percentage of CPU used to perform specific activity on a specified processor, e.g. executing at the user level, or serving interrupts. Absolute metrics are collected from the `/proc` filesystem using standard monitoring tools like `mpstat` or `sar`.

2.2 The Kubernetes' auto-scaling example

As the driving example is considered the horizontal Pods auto-scaling in Kubernetes. Kubernetes allows to create and to deploy units called *Pods*. A Pod represents a running process on a cluster and encapsulates an application

container (or, in some cases, multiple containers), storage resources, a unique network IP, and options that govern how the container(s) should run. A Pod represents a unit of deployment: a single instance of an application in Kubernetes, which might consist of either a single container or a small number of containers that are tightly coupled and that share resources (cf. Kubernetes documentation at kubernetes.io). This study assumes a one to one mapping between Pods and containers. In order to replicate an application's instance, it is enough to replicate and deploy the Pod containing the application container.

Algorithm 1 KHPA algorithm. It returns the number of Pods to be deployed

Input: U_{target}, *ActivePods*
// Target utilization and the set of active Pods
Output: P // The target number of Pods to deploy
1: **while true do**
2: **for all** $i \in ActivePods$ **do**
3: $U_i =$ `getRelativeCPUUtilization`(i);
4: $\mathbf{U} = \mathbf{U} \cup \{U_i\}$
5: **end for**
6: $P =$`ceil( sum( U ) /` U_{target} `)`;
7: `wait(`τ`)` // wait τ seconds, the control loop period
8: **end while**

The Kubernetes' Horizontal Pods Auto-scaling algorithm is based on a control loop, with a period $\tau = 30$ s (cf. Algorithm 1, line 7).

KHPA takes as input the target relative utilization U_{target} (as percentage of the requested CPU), and the set of active Pods *ActivePods*, deployed in the previous control period (τ seconds before). The output is the target number of Pods P to deploy. Each τ seconds, the algorithm gathers the relative CPU utilization of the Pods, measured with `cAdvisor` (line 3) and stores it in the vector $\mathbf{U}$ (lines 3 and 4). Finally, at line 5, the target number of Pods P, is computed using the following formula:

$$P = \left\lceil \frac{\sum_{i \in ActivePods} U_i}{U_{target}} \right\rceil. \quad (1)$$

The KHPA algorithm implemented in Kubernetes includes more features, not considered in this study because not influenced by relative/absolute usage measures, like: the possibility to define the minimum and maximum number of Pods to instantiate; and the possibility to postpone the allocation/deallocation of resources to avoid instability (ping pong effects).

Let us suppose now that $U_{target} = 66\%$, three application replicas are running, and the per-Pod CPU utilization is 79%, 75% and 83%, respectively. At the next control period, the KHPA algorithm determines that a new Pod should be deployed $P = 4$. The load will be distributed among the Pods and the estimated per-Pod utilization will be $\sum_{i=1}^{P} U_i/P = 59.25$.

The horizontal Pods auto-scaling in Kubernetes uses the relative CPU utilization. Since $U_{relative} \leq U_{absolute}$ and because Eq. 1, the inequality $P_{relative} \leq P_{absolute}$ always holds.

The deployment of $P_{relative}$ rather then $P_{absolute}$ Pods could hurt the performance of an application. That could be demonstrated, for example, using the classical response time model $R = S/(1-U)$, where S is the service time. The inequality $S/(1-U_{absolute}) \geq S/(1-U_{relative})$ is always valid, that means the expected response time (based of the relative CPU utilization) is less then the actual response time, that is $R_{absolute}$. Hence, the scaling of container based on $U_{relative}$ could produce, in practice, the violation of the service level objective defined on the response time.

3 Workload characterization

To understand the impact of relative and absolute usage measures on the performance of autoscaling algorithms a CPU intensive workload is generated and characterized (specifically by means of $U_{relative}$ and $U_{absolute}$). Although many different distributed application workload generators exists, e.g. YCSB or JMeter, for this study is important to use tools that allow to directly control the amount and type of stress produced on the CPU of the systems. Hence, `stress-ng` and `sysbench` have been selected.

Because the goal of the paper is to study the behaviour of auto-scaling algorithms under heavy-load conditions, different combination of the stress tools' parameters are used to bring the absolute CPU utilization of the testing environment in the range 60% – 95% (cf. Sect. 3.1). Specifically, `stress-ng` multiplies square matrix and allows to set as input the size N of the matrix and the number of workers W (or threads) that do the multiplications. `sysbench` verifies prime numbers by doing standard division of the input number N by all numbers between 2 and the square root of the input number. It also allows to specify the number of workers W (or threads) as input.

Two types of workload are generated. First, a single instance of the containerized stress tool is executed (i.e. a single container runs). Then, multiple containers are executed concurrently. In the second case no quotas on the CPU utilization are set to increase the interference among containers in sharing the physical/virtual resources.

Table 1 shows the values of the parameters used to generate the workload. For each value of N, experiments are executed with all the values of W. Hence, 16 different dataset are generated, and each contain $n = 10$ runs to account for system uncertainty. The same set of experiments has been conducted running one and three

Table 1 Workload parameters

Tool	Input size ($N \times 10^3$)	Num. of workers W	Num. of instances
`stress-ng`	32, 64, 128, 256	1, 2, 4, 6	1,3
`sysbench`	16, 32, 64, 128		

concurrent containers on the same host environment. In the first case we have no interference in using resources. In the second case, we use three containers for two main reasons: first, the machine we use for the experiments is a dual core, hence running three or more containers generate interference in the contention of physical resources. Second, because the limited capacity of our testbed, we empirically found that we can run three instances of a stress tool, plus the containers used to deploy the monitoring infrastructure that consist of three components: cAdvisor, Prometheus and Grafana (cf. Sect. 3.1). Performance data are sampled each 1 s.

Figures 1 and 2 show the probability distribution function (pdf) of the CPU utilization measured under the single instance workload. `stress-ng` generates a more intense workload than `sysbench`. In the `sysbench` workload $U_{relative}$ range from 65% to 91% but the majority of samples are between 80 and 90%. $U_{absolute}$ range from 82 to 91%. With the `stress-ng` workload $U_{relative}$ is in the range 76–91% and the $U_{absolute}$ in the range 90–94%.

3.1 Experimental environment and monitoring tools

The CPU usage is measured running the `sysbench` and `stress-ng` workloads on a server equipped with: 2 CPU AMD Turion(tm) II Neo N40L Dual-Core 1.5 GHz; 2 GB of RAM; ATA DISK HDD 250GB (ext4). The software platform is characterized by: Ubuntu 14.04 Trusty, Docker v 1.12.3, Grafana 3.1.1, Prometheus 1.3.1, cAdvisor 0.24.1. The containers running the monitoring infrastructure (Grafana and Prometheus) are configured to run on the cores of one of the two CPUs. In the same way, the containers running the stress tool and an instance of cAdvisor

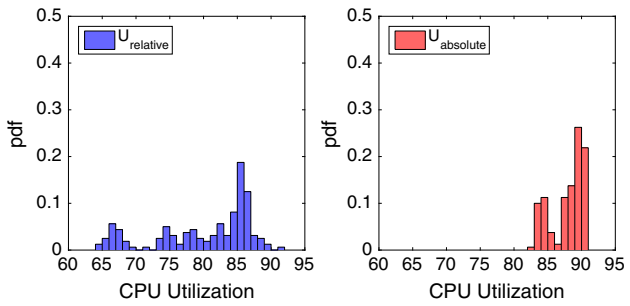

Fig. 1 The `sysbench` workload: (*left*) the probability distribution function (pdf) of the relative CPU utilization; and (*right*) the pdf of the absolute CPU utilization

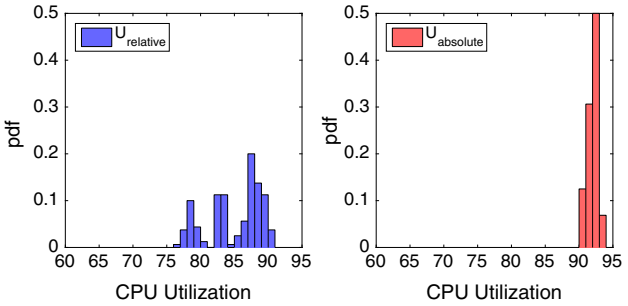

Fig. 2 The `stress-ng` workload: (*left*) the probability distribution function (pdf) of the relative CPU utilization; and (*right*) the pdf of the absolute CPU utilization

share the cores of the other CPU. We chose `cAdvisor` rather then `docker stat` because experiments show they provide the same results, but `cAdvisor` can be directly connected with Prometheus.. `mpstat` is used to collect absolute metrics on the host. Prometheus (prometheus.io) is used to extract the data sampled by `cAdvisor` each 1 second. Finally, `Grafana` (grafana.org) queries the data extracted by `Prometheus` and enables the export and the visualization of data. Although the impact of those tools on the CPU utilization of the testbed is negligible their execution is bound on the cores of one of the two CPUs available.

4 Correlation model

The correlation between the relative CPU utilization and the absolute CPU utilization is evaluated using: the Pearson's correlation coefficient (ρ) and the related 95% confidence bounds; and the p-value (that gives a measure of the static significance of the correlation coefficient—if $p < 0.05$ ρ is statistically significant).

The correlation coefficients for the datasets obtained are very similar ($\rho \approx 0.91$) and the 95% confidence interval overlaps. The p-value is always less than 0.05. The correlation model determined is linear, that is

$$U_{absolute} = b + a \cdot U_{relative} \tag{2}$$

In Fig. 3 is plotted the linear correlation for the four workloads defined in Sect. 3. The values for the linear fitting coefficients and the 95% confidence bounds are in Table 2.

The rationale behind Eq. 2 is the following. While using containers, practitioners (e.g. DevOps team) are used with

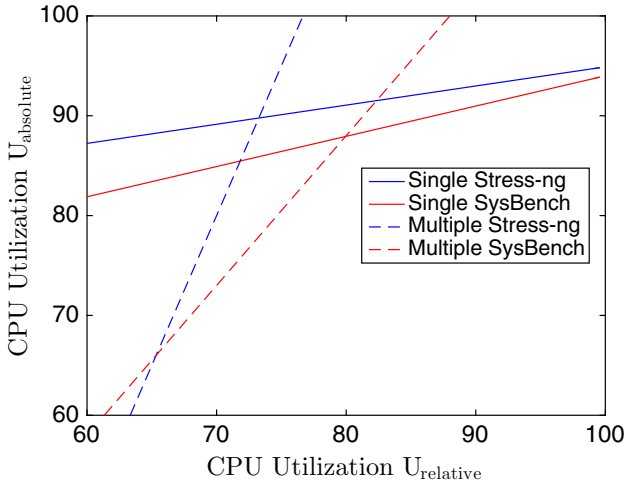

Fig. 3 The correlation model (linear interpolation) between relative CPU utilization (x-axis) and absolute CPU utilization (y-axis) for the four datasets

Table 2 Correlation model: linear fitting coefficients for the four datasets

Workload	a (95% conf. bounds)	b (95% conf. bounds)
Single	0.19	75.73
`stress-ng`	(0.17, 0.20)	(74.56, 76.91)
Single	0.30	63.71
`sysbench`	(0.28, 0.32)	(62.04, 65.37)
Concurrent	3.0	− 130.3
`stress-ng`	(2.97, 3.20)	(− 131.2, − 129.7)
Concurrent	1.5	− 32.1
`sysbench`	(1.48, 1.53)	(− 32.73, − 32.31)

relative metrics that, because their definition (c.f. Sect. 2.1), are more intuitive and practical than absolute metrics. Moreover, Docker uses quotas and limits based on relative metrics and Kubernetes autoscaling algorithm is configured defining thresholds on relative metrics. Hence, our goal was to provide a solution that make the use of the absolute metrics transparent to the users. Equation 2, the core of KHPA-A algorithm, enables a DevOps team: to use relative metrics in the configuration of their execution and orchestration environment; to use cAdvisor rather then collecting absolute metrics from the underlining physical/virtual system (not always accessible); and to take decisions based on the absolute metrics. The proposed solution imply also that the KHPA-A algorithm can be plugged in an existing environment without the need of any reconfiguration of the system and of the scaling policy parameters.

One of the drawback of the proposed approach is the uniqueness of the correlation model for a specific workload. Hence the correlation model (coefficients a and b) should be periodically evaluated, starting from run-time monitored data, and the linear correlation coefficients should be dynamically updated, according to the data collected.

5 Auto-scaling performance evaluation

The new algorithm KHPA-A takes only two additional inputs with respect KHPA, that are the linear fitting coefficients a and b of the correlation model (c.f. Table 2 and Eq. 2). U_{target} is expressed as a absolute utilization value, e.g. obtained from a response time constraint. The pseudocode for KHPA-A is in Algorithm 2. At line 4 the absolute utilization is computed using Eq. 2 (the relative utilization is collected using cAdvisor). At line 7 the number of target Pods P is computed using the $\mathbf{U}_{absolute}$ value.

Algorithm 2 `KHPA-A` algorithm. It returns the number of Pods to be deployed

Input: U_{target}, a, b, $ActivePods$
// Target utilization, correlation coefficients and the set of active Pods
Output: P // The target number of Pods to deploy
1: **while true do**
2: **for all** $i \in ActivePods$ **do**
3: $U_{relative,i}$ = `getRelativeCPUUtilization`(i);
4: $U_{absolute,i} = b + a \cdot U_{relative,i}$;
5: $\mathbf{U}_{absolute} = \mathbf{U}_{absolute} \cup \{U_{absolute,i}\}$
6: **end for**
7: P =`ceil( sum(` $\mathbf{U}_{absolute}$ `) /` U_{target} `)`;
8: `wait(`τ`)` // wait τ seconds, the control loop period
9: **end while**

5.1 Performance metrics and experiments setup

The metrics used to compare the performances of the KHPA and KHPA-A auto-scaling algorithms are: the average per-Pod absolute CPU utilization $\overline{U}$ after adaptation; the average application response time $\overline{R}$ after adaptation; and the difference ΔP between the number of pods allocated by KHPA-A and KHPA. $\overline{U}$, $\overline{R}$ and ΔP are defined by the following equations:

$$\overline{U} = \frac{1}{P} \times \sum_{i=1}^{P'} U'_i \tag{3}$$

$$\overline{R} = \frac{S}{1 - \overline{U}} \tag{4}$$

$$\Delta P = \frac{P'_{KHPA-A} - P'_{KHPA}}{P'_{KHPA-A}} \tag{5}$$

where: P' and U' are the target number of Pods and the average per-Pod utilization computed in the previous adaptation period (τ seconds earlier); S is the service time. In the results, $\overline{U}$ and $DeltaP$ are plotted as a percentage (%) rather then in the range [0, 1]. $\overline{U}$ and $\overline{R}$ are also compared with the expected per pod CPU utilization and expected average response time.

The performance comparison is based on a simulation study. The simulator is implemented in Matlab and the simulation logic is as follows:

- *Step 1* the target utilization is set and one Pod is deployed
- *Step 2* the per-Pod service demand is randomly generated, i.e. the relative CPU utilization $U_{relative}$, is generated using the empirical pdf obtained from our datasets (cf Sect. 3 and Figs. 1 and 2)
- *Step 3* each τ seconds the new number of Pods needed to match the target utilization is computed using KHPA or KHPA-A, and then Pods are deployed. The assumption is that the workload is balanced among the deployed Pods and that is available enough computational capacity (e.g. VMs) to run the Pods
- *Step 4* when the system is stable and before the τ control period expires a new workload for all the deployed Pods is generated (i.e. jump back to *Step 2*))

Fifty control periods have been simulated (i.e., 1500 s) and at each control period the total CPU demand is generated according to the probability distributions functions early determined for $U_{relative}$. However, a cap for the number of Pods is set to deploy no more than 2000 Pods. If that limit is reached the simulation is terminated.

To account for the randomness of the workload, Steps 1–4 are repeated twenty times and the average value of the metrics is computed over the 20 runs.

5.2 Experimental results

In the experiments we compare the performance of KHPA and KHPA-A under the four workloads presented in Sect. 4. For each workload three scenarios are considered: U_{target} =75%, 80% and 85%. The service time is always $S = 0.001$ s (it is only a scaling factor for $\overline{R}$).

5.2.1 Single instance workload

This correlation model does not consider any interference among running containers, hence is optimistic. Plots in Fig. 4 shows the results for the Sysbench workload. KHPA-A is always capable to maintain the absolute CPU utilization $\overline{U}$ at the level of the expected CPU utilization $U_{relative}$. Hence, also the response time $\overline{R}$ is close to the expected value. With the KHPA algorithm the absolute utilization is higher than the expected and, therefore the response time. This means that with KHPA is not possible to satisfy such kind of service level objectives.

In the 75% workload an higher number of Pods is needed to maintain the lower level of system utilization and the cap (maximum number of pods) is reached after 35 control periods. Another interesting trend is that when the target CPU utilization U_{target} increase (from 75% to 85%) the difference between the response time $\overline{R}$ achievable with KHPA and KHPA-A increase significantly. In the 75% case $\overline{R}_{KHPA}$ is about 1.5 time higher that $\overline{R}_{KHPA-A}$ and in the 85% case $\overline{R}_{KHPA}$ is about 2 time higher that $\overline{R}_{KHPA-A}$.

Is also interesting to observe how ΔP converge to the value of 10%. This behaviour is more evident in the cases of a target utilization equal to 75% and 80%. That means KHPA-A use more containers than KHPA. In case of stringent constraints on the CPU utilization, like in the case of $U_{target} = 75\%$. For very large deployments, KHPA-A can allocate also 200 containers more than KHPA.

The performance behaviour under the `stress-ng` workload is the same as for `sysbench`. Plots in Figs. 5 show the results. `stress-ng` produces an higher CPU utilization than `sysbench`. For the case target utilization $T = 80\%$ the maximum number of Pods is reached after 43 control periods, and for $T = 75\%$ after 32 control periods. With $T = 85\%$ the cup is not reached in the 50 control periods.

Under the `stress-ng` workload, the ΔP converge to the value of 7.58. This means that KHPA reacts well to CPU intensive workload, but always allocate an under-estimated number of Pods.

5.2.2 Concurrent workload

The most interesting cases are for U_{target} = 80% and 85%.

Plots in Fig. 6 shows the results for `stress-ng`. For U_{target} = 80%, due to the contention of the CPU, KHPA is not capable to deploy the right number of containers and the $\overline{U}_{KHPA}$ is about 18% higher than the expected CPU utilization. The higher $\overline{U}$ results in a response time $\overline{R}_{KHPA}$ that is more than 2 times the $\overline{R}_{Expected}$. On the contrary, KHPA-A algorithm provides performance close to the expected value and hence, as allocation policy, could guarantee responce time based constraints better than KHPA. When the target utilization increases to U_{target} = 85%, and KHPA algorithm is used, the system becames more unstable. Indeed the higher per Pod utilization brings the host utilization to 95% and up to 100% (and more) when the workload increases. That results in a $\overline{R}_{KHPA}$ that is from 1 to 3 orders of magnitude higher than $\overline{R}_{Expected}$: the typical behaviour of a saturated system. In both cases, the value of ΔP converge to 14.8 that mans, for very large

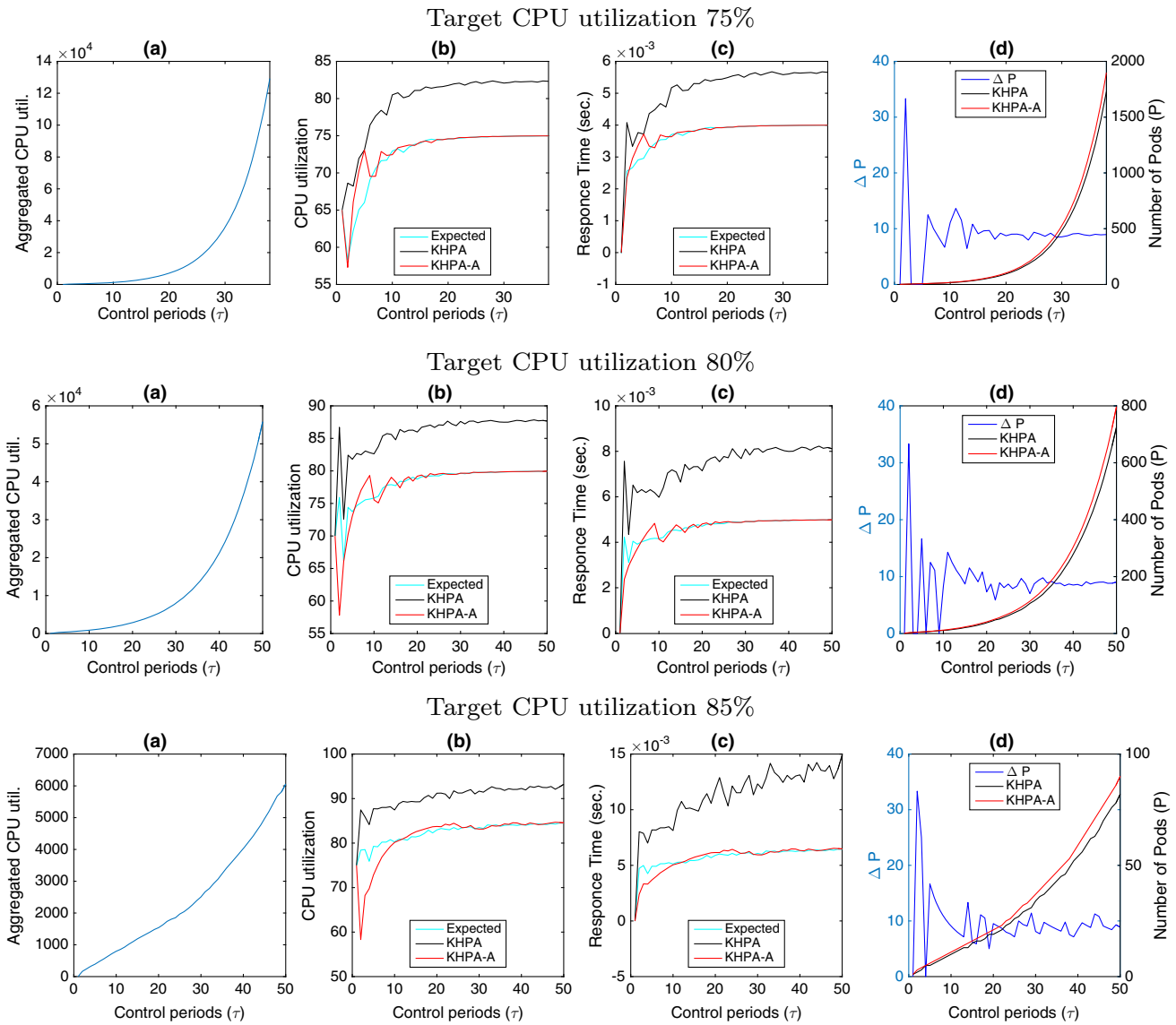

Fig. 4 Sysbench workload, single instance. **a** The aggregated relative CPU utilization demanded by all the deployed Pods. **b** The average per-Pod CPU utilization $\overline{U}$. **c** The response time $\overline{R}$. **d** ΔP and the number of deployed Pods P

clusters, up to 300 more containers could be needed to guarantee the target CPU utilization.

In Fig. 7 the results for `sysbench` are reported. In this case, the overall workload is less intense than the case for `stress-ng`, hence there is no saturation phenomenon. However, when the target value for the utilization is 85%, the $\overline{U}_{KHPA}$ exceed the 90% and the responce time could be between 60 up to 100% higher than the expected. In both cases ΔP converge to the value of 8.5.

5.2.3 Discussion

From the above results is evident the advantage of using KHPA-A versus KHPA. Shortly speaking, for CPU intensive workloads KHPA under-dimensions the number of deployed Pods because the use of relative metrics. As consequence, that will make impossible to setup an auto-scaling policy capable to satisfy QoS requirements like constraints on the responce time. On the contrary, with KHPA-A, the scaling in/out actions are determined by the absolute metrics (or better an estimation of the value assumed by the absolute metrics). Absolute metrics, e.g. the value of the CPU utilization of the virtual/phisical host, have the advantage to measure the real usage of the host system end hence can be used to compute QoS metrics like the responce time. With KHPA-A a constraint on the responce time can be translated into a constraint on the CPU utilization, for example using the well know formula $R = S/(1 - U)$ as explained in Sect. 2.1. In conclusion, we

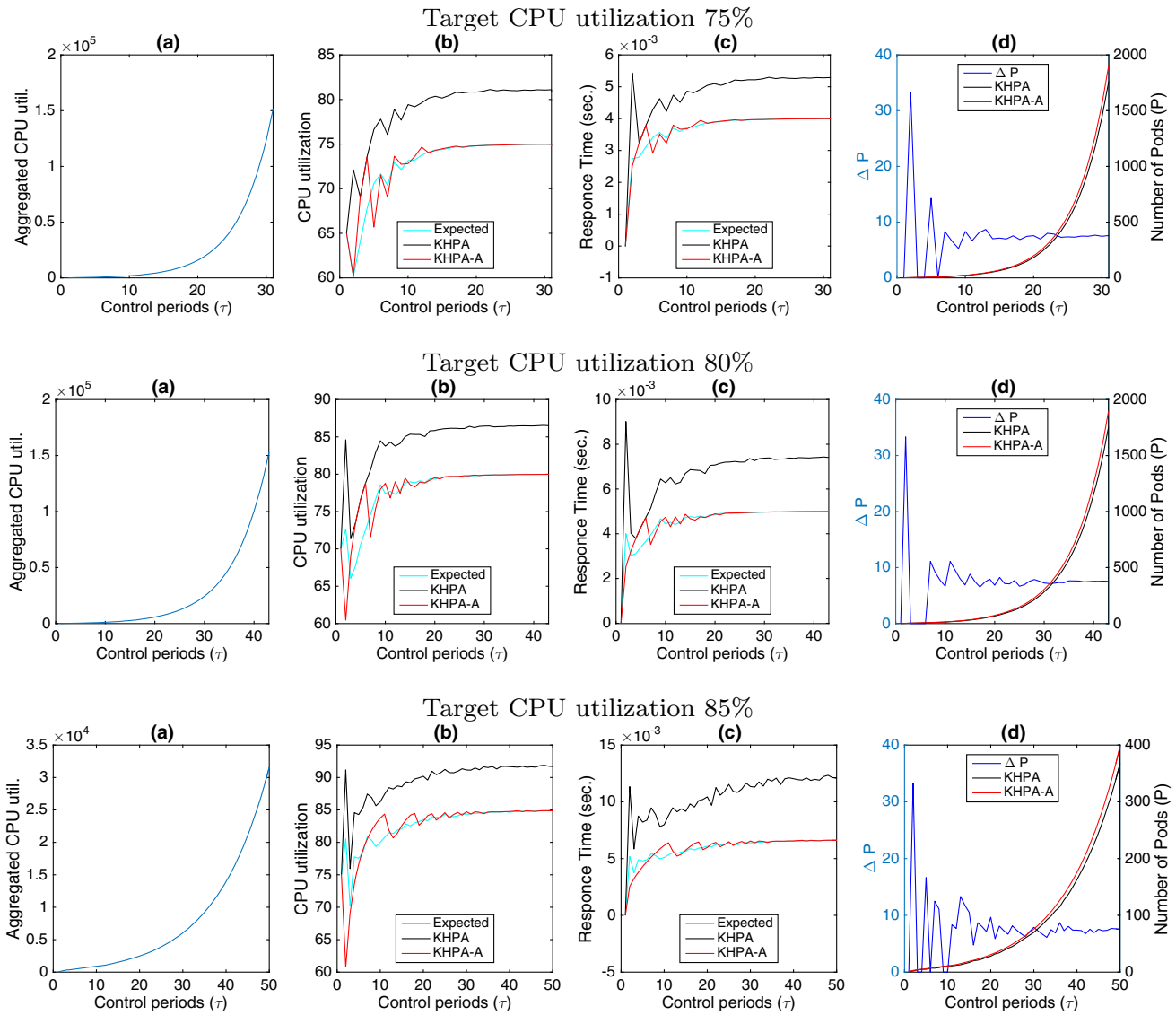

Fig. 5 Stressng workload, single instance. **a** The aggregated relative CPU utilization demanded by all the deployed Pods. **b** The average per-Pod CPU utilization $\overline{U}$. **c** The response time $\overline{R}$. **d** ΔP and the number of deployed Pods P

recommend to use KHPA-A each time is important to meet QoS constraints.

6 Related works

The state-of-the-art studies on performance evaluation of container and on auto-scaling algorithms for containers are reviewed in what follow.

6.1 Performance

Many research works have assessed the performance of containers against virtual machines confirming the small footprint of containers. Those results boosted the container adoption. However, this works proved also the poor I/O throughput of containers, that is prohibitive for network intensive applications. Other works aim at evaluating and predicting the performance of containerized application, to provide models for capacity planning, optimal deployment and run-time adaptation. Finally, some research studies aim at solving monitoring challenges.

The first seminal work on container performance evaluation [14] provides an extensive comparison among a native Linux environment, Docker and KVM. In this work the performances of the three environments are compared in presence of CPU intensive, I/O intensive, Network intensive, and NoSQL/SQL workloads. A similar study, aimed at comparing the performance of containers with hypervisors is [25]. In [30] as been investigated the

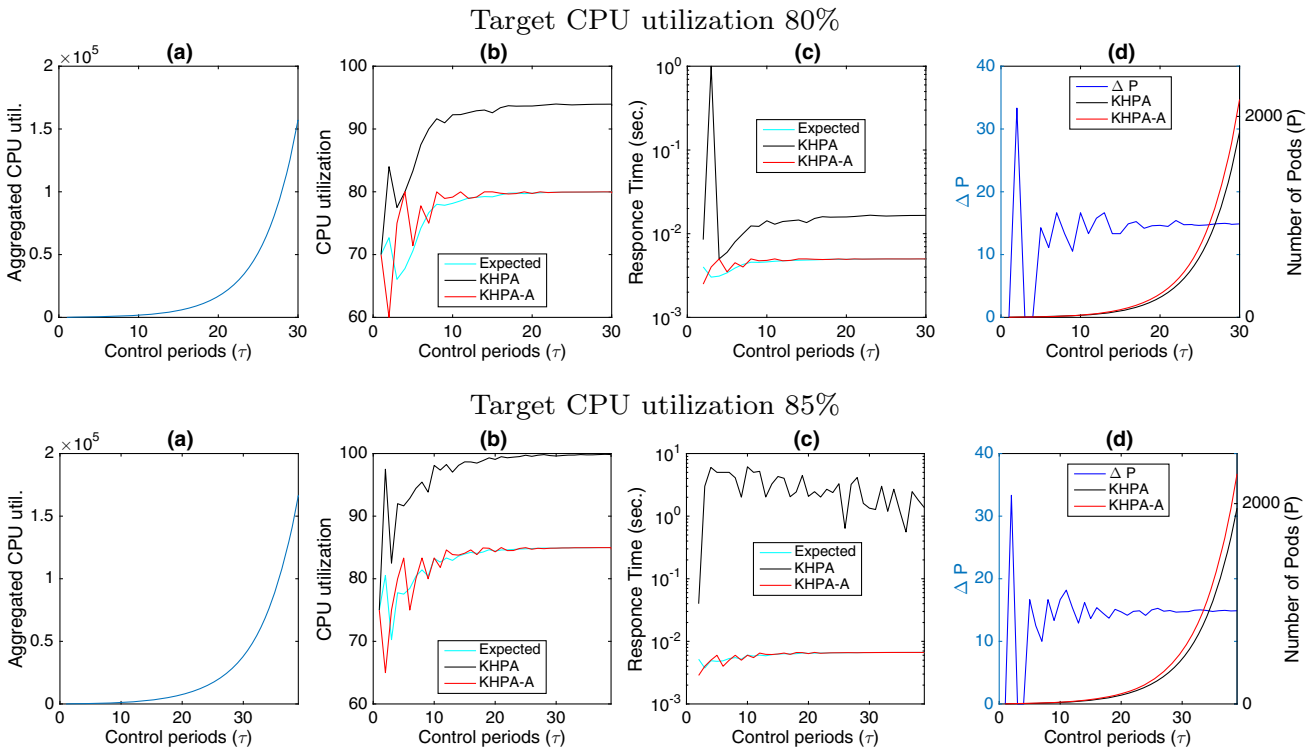

Fig. 6 Stressng workload, multiple instances. **a** The aggregated relative CPU utilization demanded by all the deployed Pods. **b** The average per-Pod CPU utilization $\overline{U}$. **c** The response time $\overline{R}$. **d** ΔP and the number of deployed Pods P

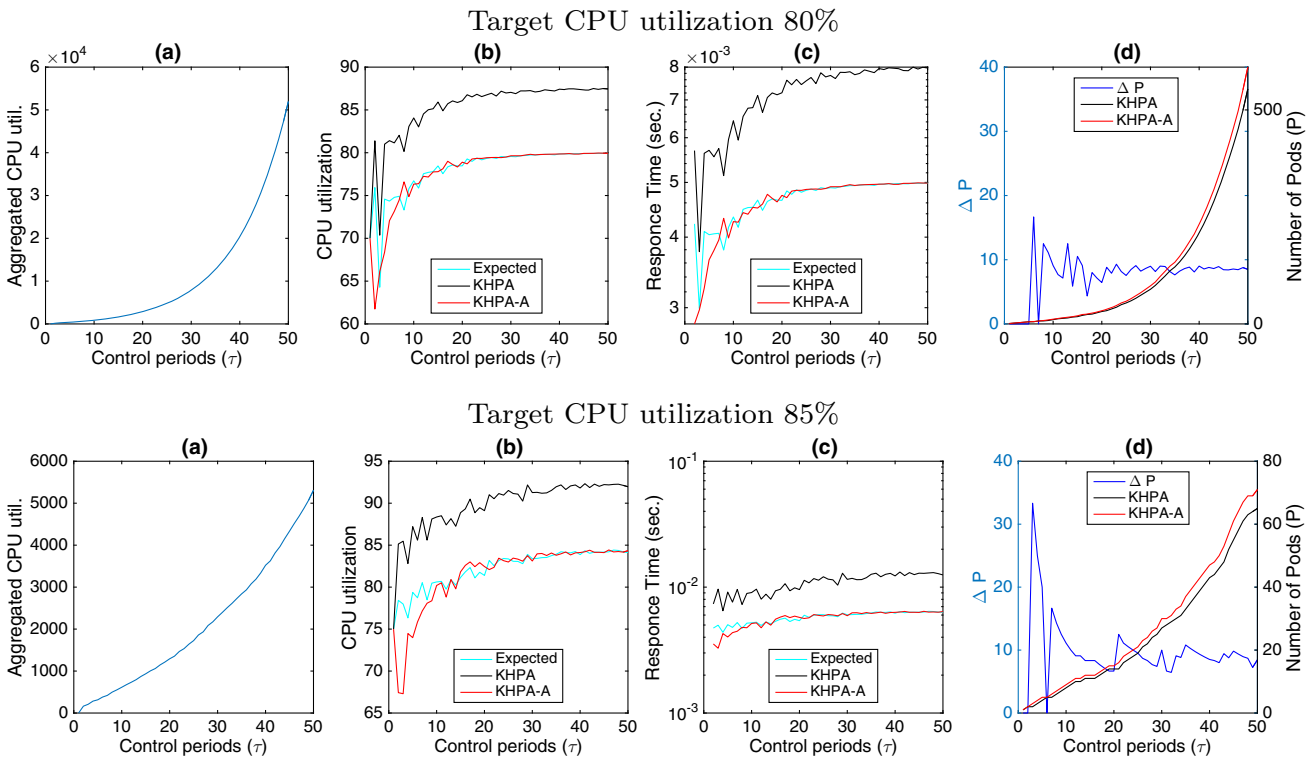

Fig. 7 Sysbench workload, multiple instances. **a** The aggregated relative CPU utilization demanded by all the deployed Pods. **b** The average per-Pod CPU utilization $\overline{U}$. **c** The response time $\overline{R}$. **d** ΔP and the number of deployed Pods P

performance of Cassandra while running on: a bare-metal cluster, VMware VMs and Docker containers. While standard benchmarks were used in the previous studies, a scientific workload is considered in [3]. The authors shown that Docker memory configuration can be tuned to make container performance be slightly better than VMs.

Other studies investigate the performance of containers running on cloud infrastructures. A performance comparison of Docker versus Flockport (LXC) is presented in [22], using the same benchmarks as in [14]. The containers are deployed on top of the NeCTAR cloud. The comparison was intended to explore the performance of CPU, memory, network and disk. Docker versus Joyent's Triton is studied in [5]. The authors shows that on top of AWS EC2 virtual machines Joyent's Triton performs better or almost the same than Docker with the advantage of enhanced security features.

A performance prediction model based on the Support Vector Regression has bee proposed in [35] to forecast the performance of data stream processing applications implemented with Apache Spark under different configurations and resource competition settings. In [32] the authors proposed a Learning Classifier System that, on the basis of multi-layer monitored data, incrementally learns rules representing the containerized application QoS behaviour. A Layered Queuing Network-based performance prediction model for multi-tier containerized applications is proposed in [6]. The model allows to predict the resource demand.

Performance monitoring is a topic of increasing interest for the containers' research community. In [11] the authors assess the different measurement methodology used to collect performance counters for CPU and disk I/O intensive Docker workload. The importance of using information about the performance of all the stack components to take effective deployment and adaptation decisions is addressed in [20] where the authors propose Elascale, a cloud service for monitoring performance metrics at all the layers of the computational stack. The same approach is used in [32], where a multi-layer monitored data are used to drive the run-time adaptation.

This paper enhances the literature in different way. First has been extended the correlation models proposed in [10] considering a concurrent workload. Then, the correlation model is used to take appropriate auto-scaling decisions. This work confirms also the importance of monitoring the appropriate performance counters, as suggested in [11].

6.2 Auto-scaling

Container orchestrators offer many features to aid DevOps teams in managing the container life cycle both in the offline and run-time phases. One of the run-time management tool is auto-scaling. Kubernetes and Docker SwarmAre orchestrators that offer threshold based auto-scaling algorithms based on relative metrics (mainly CPU and memory utilization).

An early study on container resource management [17] shows that Elastic Application Container-based resource management outperforms the VM-based approach in terms of feasibility and resource-efficiency. C-Port [1] is the first example of orchestrator that use a constraint-programming model for dynamic resource discovery, selection and deployment. The C-Port orchestrator is used in [2] to realize a distributed software defined environment that deploys applications with docker over the CometCloud federated cloud infrastructure. In [26] the authors provide a general formulation of the elastic provisioning of virtual machines for container deployment as an integer linear programming problem. The heterogeneity of the environment, QoS and cost constraints are considered in the problem formulation. The proposed algorithm provides either the optimal VMs allocation and the scaling of containers. In [27] the authors propose Adaptive Container Deployment (ACD), a general model of the deployment and adaptation of containerized applications, expressed as an Integer Linear Programming problem. Besides acquiring and releasing geo-distributed computing resources, ACD can optimize multiple run-time deployment goals, by exploiting horizontal and vertical elasticity of containers. An adaptive multi-instance container-based architecture targeting time-critical applications is proposed in [31]. ElasticDocker, an autonomic controller powering vertical elasticity of Docker containers autonomously is presented in [4]. ElasticDocker scales up and down both CPU and memory assigned to each container according to the application workload. An architecture of a SaaS application manager based on Docker and Kubernetes is proposed in [33]. The paper describes the high level architecture based three autonomic managers that should be capable to adapt the multi-cloud infrastructure and the multi technology data storage level with the goal of guarantee tenants' SLAs. In [19] the authors propose an Ant Colony Optimization (ACO) algorithm to schedule docker containers with the ultimate goal to use resources more efficiently. The ACO algorithm performance is compared with the greedy scheduling algorithm in DockerSwarm. A framework for Application Oriented Docker container (AODC) resource allocation to minimize the application deployment cost in datacenters is presented in [16]. AODC considers deployment of container on PM and is compared against optimal VM placement algorithms. Elascale [20] applies a default threshold-based, reactive auto-scaling algorithm for all application's micro and macro services. In [21] the authors propose a proactive autoscaling mechanism that scale-in/out containers and distribute the load

among instances on the base of the network traffic intensity. In [12] the authors propose Virtual Hadoop a framework for deploying containerized Hadoop clusters on heterogeneous nodes. The framework include an scaling algorithm to meet application time constraints. In [34] the authors propose an autoscaling mechanisms for Data Stream Processing Platform Service, deployed using Docker and Kubernetes. The proposed solution takes decisions on the basis of the predicted data arrival rate.

This work, differently from the others aim to enhances the auto-scaling of containers in Kubernetes proposing a QoS aware auto-scaling algorithm that make a transparent use of the platform level (absolute) metrics.

7 Conclusions

This work propose and evaluate the performance of KHPA-A an enhanced version of the KHPA auto-scaling algorithm. The proposed solution leverage, in a transparent way, absolute usage measures rather then relative. KHPA-A can be plugged in any existing system orchestrated with Kubernetes without the need to change the system configuration.

The performance comparison shows that the use of absolute metrics allows to properly control the application response time and to keep it below thresholds introduced by service level objectives. Specifically, for high loaded servers. In the single instance workload case, the responce time obtained with the KHPA is a factor between 1.5 and 2 higher than the response time obtainable with the KHPA-A. For the concurrent workload, the KHPA's responce time is between 2 and 3 order of magnitude higher than the KHPA-A's responce time.

Although this study focused only on Kubernetes' auto-scaling algorithm, the results presented should be considered as a guideline when implementing any container's auto-scaling algorithm for managing CPU intensive workloads with QoS constraints.

Acknowledgements This work is funded by the BigData@BTH Knowledge Foundation Grant No. 20140032, Sweden. The author thanks Vanessa Perciballi, Spindox S.p.A., for the work done on part of the experiments.

References

1. Abdelbaky, M., Diaz-Montes, J., Parashar, M., Unuvar, M., Steinder, M.: Docker containers across multiple clouds and data centers. In: Proceedings of the 2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC), pp. 368–371 (2015). https://doi.org/10.1109/UCC.2015.58
2. Abdelbaky, M., Diaz-Montes, J., Parashar, M.: Towards distributed software-defined environments. In: Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid '17, pp. 703–706. IEEE Press, Piscataway, NJ, USA (2017). https://doi.org/10.1109/CCGRID.2017.30
3. Adufu, T., Choi, J., Kim, Y.: Is container-based technology a winner for high performance scientific applications? In: Proceedings of the 17th Asia-Pacific Network Operations and Management Symposium, APNOMS 2015, Busan, South Korea, August 19-21, 2015, pp. 507–510. IEEE (2015). https://doi.org/10.1109/APNOMS.2015.7275379
4. Al-Dhuraibi, Y., Paraiso, F., Djarallah, N., Merle, P.: Autonomic vertical elasticity of docker containers with elasticdocker. In: Proceedings of the 2017 IEEE 10th International Conference on Cloud Computing, pp. 472–479 (2017). https://doi.org/10.1109/CLOUD.2017.67
5. Balasubramanian Sekar, V., Patil, V., Giusti, M., Bhide, A., Gupta, A.: Aws ec2 vs. joyent's triton: a comparison of docker container-hosting platforms. In: Proceedings of the 8th Workshop on Scientific Cloud Computing, pp. 33–36. ACM, New York, NY, USA (2017). https://doi.org/10.1145/3086567.3086572
6. Barna, C., Khazaei, H., Fokaefs, M., Litoiu, M.: Delivering elastic containerized cloud applications to enable DevOps. In: Proceedings of the 12th International Symposium on Software Engineering for Adaptive and Self-managing Systems (SEAMS), IEEE/ACM (2017)
7. Bernstein, D.: Containers and cloud: from lxc to docker to kubernetes. IEEE Cloud Comput. **1**(3), 81–84 (2014)
8. Burns, B., Grant, B., Oppenheimer, D., Brewer, E., Wilkes, J.: Borg, omega, and kubernetes. ACM Queue **14**, 70–93 (2016)
9. Casalicchio, E.: Autonomic orchestration of containers: problem definition and research challenges,. In: Proceedings of the 10th EAI International Conference on Performance Evaluation Methodologies and Tools, EAI (2016)
10. Casalicchio, E., Perciballi, V.: Auto-scaling of containers: the impact of relative and absolute metrics. In: Proceedings of the 2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems (FAS*W), pp. 207–214 (2017). https://doi.org/10.1109/FAS-W.2017.149
11. Casalicchio, E., Perciballi, V.: Measuring docker performance: What a mess!!! In: Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion, ICPE '17 Companion, pp. 11–16. ACM, New York, NY, USA (2017). https://doi.org/10.1145/3053600.3053605
12. Chen, Y.W., Hung, S.H., Tu, C.H., Yeh, C.W.: Virtual hadoop: Mapreduce over docker containers with an auto-scaling mechanism for heterogeneous environments. In: Proceedings of the International Conference on Research in Adaptive and Convergent Systems, RACS '16, pp. 201–206. ACM, New York, NY, USA (2016). https://doi.org/10.1145/2987386.2987408
13. Dua, R., Raja, A.R., Kakadia, D.: Virtualization versus containerization to support PaaS. In: Proceedings of 2014 IEEE International Conference on Cloud Engineering, IC2E '14, pp. 610–614 (2014)
14. Felter, W., Ferreira, A., Rajamony, R., Rubio, J.: An updated performance comparison of virtual machines and Linux containers. Technical Report RC25482(AUS1407-001), IBM, IBM Research Division, Austin Research Laboratory (2014)

15. Gerlach, W., Tang, W., Keegan, K., Harrison, T., Wilke, A., Bischof, J., D'Souza, M., Devoid, S., Murphy-Olson, D., Desai, N., Meyer, F.: Skyport: Container-based execution environment management for multi-cloud scientific workflows. In: Proceedings of the 5th International Workshop on Data-Intensive Computing in the Clouds, pp. 25–32. IEEE Press, Piscataway, NJ, USA (2014). https://doi.org/10.1109/DataCloud.2014.6
16. Guan, X., Wan, X., Choi, B.Y., Song, S., Zhu, J.: Application oriented dynamic resource allocation for data centers using docker containers. IEEE Commun. Lett. **21**(3), 504–507 (2017)
17. He, S., Guo, L., Guo, Y., Wu, C., Ghanem, M., Han, R.: Elastic application container: A lightweight approach for cloud resource provisioning. In: Proceedings of the 2012 IEEE 26th International Confernec on Advanced Information Networking and Applications, pp. 15–22 (2012). https://doi.org/10.1109/AINA.2012.74
18. Helsley, M.: Lxc: Linux container tools. IBM devloperWorks Technical Library p. 11 (2009)
19. Kaewkasi, C., Chuenmuneewong, K.: Improvement of container scheduling for docker using ant colony optimization. In: Proceedings of the 2017 9th International Conference on Knowledge and Smart Technology, pp. 254–259 (2017). https://doi.org/10.1109/KST.2017.7886112
20. Khazaei, H., Ravichandiran, R., Park, B., Bannazadeh, H., Tizghadam, A., Leon-Garcia, A.: Elascale: autoscaling and monitoring as a service. In: Proceedings of the 27th Annual International Conference on Computer Science and Software Engineering, pp. 234–240. IBM Corp., Riverton, NJ, USA (2017). http://dl.acm.org.miman.bib.bth.se/citation.cfm?id=3172795.3172823
21. Kim, W.Y., Lee, J.S., Huh, E.N.: Study on proactive auto scaling for instance through the prediction of network traffic on the container environment. In: Proceedings of the 11th International Conference on Ubiquitous Information Management and Communication, pp. 17:1–17:8. ACM, New York, NY, USA (2017). https://doi.org/10.1145/3022227.3022243
22. Kozhirbayev, Z., Sinnott, R.O.: A performance comparison of container-based technologies for the cloud. Future Gener. Comput. Syst. **68**, 175–182 (2017)
23. Martino, B.D., Cretella, G., Esposito, A.: Advances in applications portability and services interoperability among multiple clouds. IEEE Cloud Comput. **2**(2), 22–28 (2015)
24. Merkel, D.: Docker: lightweight Linux containers for consistent development and deployment. Linux J. **2014**, 239 (2014)
25. Morabito, R., Kjällman, J., Komu, M.: Hypervisors vs. lightweight virtualization: a performance comparison. In: Proceedings of the 2015 IEEE International Conference on Cloud Engineering, pp. 386–393 (2015). https://doi.org/10.1109/IC2E.2015.74
26. Nardelli, M., Hochreiner, C., Schulte, S.: Elastic provisioning of virtual machines for container deployment. In: Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion, Companion, pp. 5–10. ACM, New York, NY, USA (2017). https://doi.org/10.1145/3053600.3053602
27. Nardelli, M., Cardellini, V., Casalicchio, E.: Multi-level elastic deployment of containerized applications in geo-distributed environments. In: Proceedings of 2018 IEEE 6th International Conference on Future Internet of Things and Cloud. IEEE (2018). https://doi.org/10.1109/FiCloud.2018.00009
28. Nguyen, D.T., Yong, C.H., Pham, X.Q., Nguyen, H.Q., Loan, T.T.K., Huh, E.N.: An index scheme for similarity search on cloud computing using mapreduce over docker container. In: Proceedings of the 10th International Conference on Ubiquitous Information Management and Communication, pp. 60:1–60:6. ACM, New York, NY, USA (2016). https://doi.org/10.1145/2857546.2857607
29. Pahl, C.: Containerization and the paas cloud. IEEE Cloud Comput. **2**(3), 24–31 (2015). https://doi.org/10.1109/MCC.2015.51
30. Shirinbab, S., Lundberg, L., Casalicchio, E.: Performance evaluation of container and virtual machine running Cassandra workload. In: Proceedings of the 3rd International Conference on Cloud Computing Technologies and Applications (IEEE CloudTech17). IEEE, Rabat, Marocco (2017)
31. Stankovski, V., Trnkoczy, J., Taherizadeh, S., Cigale, M.: Implementing time-critical functionalities with a distributed adaptive container architecture. In: Proceedings of the 18th International Conference on Information Integration and Web-based Applications and Services, pp. 453–457. ACM, New York, NY, USA (2016). https://doi.org/10.1145/3011141.3011202
32. Taherizadeh, S., Stankovski, V.: Incremental learning from multi-level monitoring data and its application to component based software engineering. In: Proceedings of the 41st Annual Computer Software and Applications Conference, vol. 2, pp. 378–383. IEEE (2017). https://doi.org/10.1109/COMPSAC.2017.148
33. Truyen, E., Van Landuyt, D., Reniers, V., Rafique, A., Lagaisse, B., Joosen, W.: Towards a container-based architecture for multi-tenant saas applications. In: Proceedings of the 15th International Workshop on Adaptive and Reflective Middleware, pp. 6:1–6:6. ACM, New York, NY, USA (2016). https://doi.org/10.1145/3008167.3008173
34. Wu, Y., Rao, R., Hong, P., Ma, J.: Fas: A flow aware scaling mechanism for stream processing platform service based on lms. In: Proceedings of the 2017 International Conference on Management Engineering, Software Engineering and Service Sciences, pp. 280–284. ACM, New York, NY, USA (2017). https://doi.org/10.1145/3034950.3034965
35. Ye, K., Ji, Y.: Performance tuning and modeling for big data applications in docker containers. In: Proceedings of the 2017 International Conference on Networking, Architecture, and Storage, pp. 1–6 (2017). https://doi.org/10.1109/NAS.2017.8026871
36. Zhang, R., Li, M., Hildebrand, D.: Finding the big data sweet spot: Towards automatically recommending configurations for hadoop clusters on docker containers. In: Proceedings of the 2015 IEEE International Conference on Cloud Engineering, pp. 365–368 (2015). https://doi.org/10.1109/IC2E.2015.101
37. Zounmevo, J.A., Perarnau, S., Iskra, K., Yoshii, K., Gioiosa, R., Essen, B.C.V., Gokhale, M.B., Leon, E.A.: A container-based approach to os specialization for exascale computing. In: First Workship on Containers 2015 (2015)

Emiliano Casalicchio is an Associate Professor in Computer Science. Emiliano's research interests include large scale distributed systems, with a focus on the performance of Cloud-based systems and on the security of Critical Infrastructures. More recently, he has been working on autonomic cloud computing, big data security, identity theft prevention in mobile systems. He is author of more than 90 papers and he serves as the PC member in international conferences, as reviewer for the international journals.