Auto-scaling of Containers: the Impact of Relative and Absolute Metrics

Emiliano Casalicchio
Department of Computer Science and Engineering
Blekinge Institute of Technology
Karlskrona, Sweden
Email: emiliano.casalicchio@bth.se

Vanessa Perciballi
Spindox S.p.A.,
Milano, Italy
Email: vanessa.perciballi@spindox.it

***Abstract*—Today, The cloud industry is adopting the container technology both for internal usage and as commercial offering. The use of containers as base technology for large-scale systems opens many challenges in the area of resource management at run-time. This paper addresses the problem of selecting the more appropriate performance metrics to activate auto-scaling actions. Specifically, we investigate the use of relative and absolute metrics. Results demonstrate that, for CPU intense workload, the use of absolute metrics enables more accurate scaling decisions. We propose and evaluate the performance of a new autoscaling algorithm that could reduce the response time of a factor between 0.66 and 0.5 compared to the actual Kubernetes' horizontal auto-scaling algorithm.**

I. Introduction

Operating system and application virtualization, also known as container (e.g. Docker [1]) and LXC [2]), became popular since 2013 with the launch of the Docker open source project (docker.com) and with the growing interest of cloud providers [3], [4] and Internet Service Providers (ISP) [5].

A container is a software environment where one can install an application or application component (the so called microservice) and all the library dependencies, the binaries, and a basic configuration needed to run the application. Containers provide a higher level of abstraction for the process lifecycle management, with the ability not only to start/stop but also to upgrade and release a new version of a containerized application or of a microservice in a seamless way.

Containers potentially may solve many distributed application challenges [6], e.g. portability and performance overhead. With containers, a microservice or an application can be executed on any platform running a container engines [7]. Containers are lightweight and introduce lower overhead compared to Virtual Machines (VMs) [8], [9], [10], [11]. That are some of the reasons that make the cloud computing industry to adopt container technologies and to contribute to the evolution of this technology [12], [4], [13]. Cloud service providers today offer container-based services and container development platforms [3], for example: Google container engine [12], Amazon ECS (Elastic Container Service), Alauda (alauda.io), Seastar (seastar.io), Tutum (tutum.com), Azure Container Service (azure.microsoft.com). Containers are also adopted in HPC (e.g. [14]) and to deploy large scale big data applications, requiring high elasticity in managing a very large amount of concurrent components (e.g. [15], [16], [17]).

The use of containers as base technology for deploying large-scale applications opens many challenges in the area of resource management at run-time. In this paper we focus on container auto-scaling mechanisms and we analyze the impact of leveraging relative versus absolute resource usage metrics to take adaptation decisions. Absolute usage metrics account for the actual utilization of resources in the host system (virtual machine or physical server). Relative metrics measure the share that each container has of the resources used. From our previous study [11] emerges that absolute and relative metric assume quite different values.

Relative metrics are today used in container orchestration frameworks such as Cabernets (kubernetes.io). For example the Cabernets' horizontal auto-scaling mechanisms, *Cabernets Horizontal Pod Auto-scaling* (KHPA), use relative metrics to trigger the scaling actions and to compute the number of containers to be deployed to keep the resource utilization below a specified threshold value. Relative metrics allow to easily configure horizontal scaling thresholds and the sharing of resources among different containers (e.g. defining usage quotas). However, when it comes with the proper resource allocation needed to satisfy service level objectives, such as the application response time, absolute metrics are more appropriate and relative metrics should be used carefully.

The goal of this paper is twofold. First, we study the correlation between the values of the relative and the absolute metrics. We determine the correlation analyzing data obtained from measurements on a real system and we extract a correlation model that could be used in capacity planning and autoscaling of container platforms. Second, we propose a new auto-scaling algorithm, named KHPA-A, based on absolute metrics. The new algorithm is conceived to be plugged into the Kubernetes controller (cf. Figure 1). KHPA-A can be configured with the same parameters of the original algorithms KHPA, that is relative metrics thresholds, and it determines the number of containers to be deployed on the basis of the absolute metric values obtained using the correlation model we have extracted from real data.

Our performance study show that, using absolute metrics to trigger the container scaling actions and to dimension the

978-1-5090-6558-5/17 $31.00 © 2017 IEEE
DOI 10.1109/FAS-W.2017.149

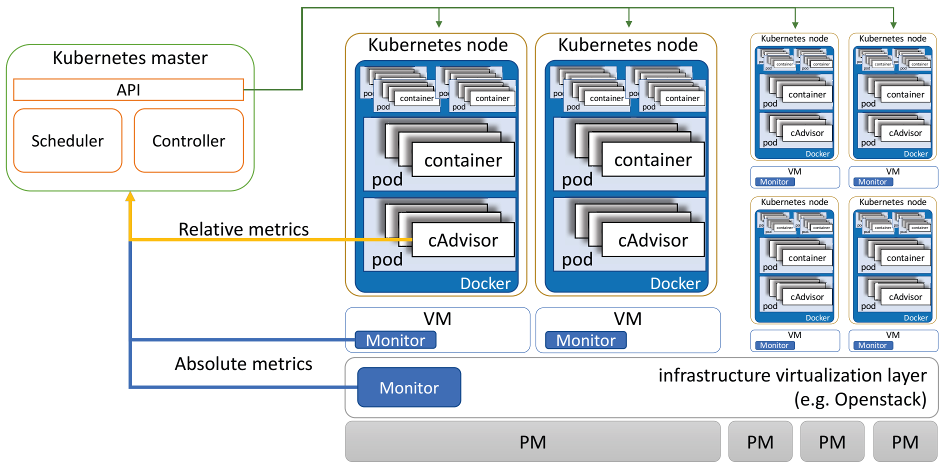

Fig. 1. The high level architecture of the auto-scaling controller in Kubernetes. Relative metrics are collected by cAdvisor. Absolute metrics could be collected at host level (virtual machines) and/or infrastructure level (physical machines)

appropriate number of Pods, allow to properly control the application response time and to keep it below thresholds introduced by service level objectives. Specifically, for high loaded servers (absolute CPU utilization in the interval 85%-90%) the response time obtained with the KHPA is a factor between 1.5 and 2 higher than the response time obtainable with the KHPA-A algorithm.

Although we focused only on CPU intensive workload and on Kubernetes' auto-scaling algorithms, the results of this work should be considered as an important guideline when implementing any container's auto-scaling algorithm. Future works will investigate the correlation of absolute and relative metrics for a wide range of workloads, and the effect of using relative and absolute metrics in other orchestration platforms, e.g. Swarm, Fleet, Marathon and Aurora.

The paper is organized as follows. Section II introduces the concept of relative and absolute metrics, it describes how the horizontal containers auto-scaling works in Kubernetes and it provides the motivating example for our study. Section III 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 correlation model between relative and absolute CPU utilization is determined in Section IV. The KHPA-A algorithm is presented in Section V along with the performance evaluation results. Related work are discussed in Section VI. Finally, Section VII gives the concluding remarks.

II. Motivating Example

A. Relative and Absolute metrics

The performance metrics used in containers' resource scheduling could be classified as *relative metrics* or *absolute metrics*.

We call *relative* the metrics 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 measure the share of CPU used by a container respect to the other containers. Relative CPU utilization is reported as percent of total host capacity. So if you have 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 would be fully utilized.

Absolute metric values report about the cumulative activity counters in the operating system. For example the absolute CPU utilization report 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`.

B. Kubernetes auto-scaling

As driving example we consider 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). We focus on Pods running a single container. To replicate an application's instance it is enough to replicate and deploy the Pod containing the application container.

The Kubernete's Horizontal Pods Auto-scaling algorithm (KHPA) is based on a control loop, with a period $\tau = 30$ sec (cf. Algorithm 1, line 7). The KHPA algorithm is implemented as part of the Kubernetes' *controller* in the Master Node, cf. Figure 1.

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( $\mathbf{U}$ ) / $U_{target}$ );
7:     wait($\tau$) // wait $\tau$ seconds, the control loop period
8: end while
```

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, is computed the target number of Pods P, defined by

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

The KHPA algorithm implemented in Kubernetes includes also: 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). None of these features are considered here.

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 determine that a new Pod should be deployed $P = 4$. The load will be distributed among the Pods and the estimated per-Pod utilization became $\sum_{i=1}^{P} U_i/P = 59.25$.

The horizontal Pods auto-scaling in Kubernetes uses the relative CPU utilization. Because $U_{relative} \leq U_{absolute}$ then always is $P_{relative} \leq P_{absolute}$.

The deployment of $P_{relative}$ rather then $P_{absolute}$ Pods could hurt the performance of an application. For example, If we consider the response time $R = S/(1-U)$, where S is the service time, we have

$$\frac{S}{(1-U_{absolute})} \geq \frac{S}{(1-U_{relative})}$$

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 all the service level objective defined on the response time.

III. Workload characterization

The first step to understand the impact of relative and absolute metrics on the performance of container autoscaling is to characterize the container workload. Specifically, the characterization should be both from the relative and absolute point of view. We proceed as follow:

1) a workload is generated using two different CPU intensive benchmarks, i.e. `stress-ng` [18] (matrix multiplication) and `sysbench` [19] (divisions and comparison)
2) we measured the CPU utilization, both relative and absolute values
3) we estimate the empirical probability distribution function of $U_{relative}$ and $U_{absolute}$.

The second step to assess the impact of relative and absolute metrics is to model the correlation between $U_{relative}$ and $U_{absolute}$. The correlation analysis and the correlation model are presented in the next section.

A. *Workload model*

As before mentioned, we are interested in studying the autoscaling performance when the system is highly loaded. Hence we started from generating a workload that, on our server (cf. Sec. III-B for details on the experimental environment) demand for an absolute CPU utilization in the range 80% – 95%. To achieve this goal we used different combination of the benchmarks' parameters. Specifically, `stress-ng` allows to set the size N of the square matrixes to multiply and the number of workers W (or threads) that do the multiplications. `sysbench` verifies the prime numbers by doing standard division of the input number N by all numbers between 2 and the square root of the input number. Hence the `sysbench` input consist of an natural number and of the number of workers W (or threads) that do the divisions and comparisons.

TABLE I
Workload generation parameters

Benchmark	Input size ($N \times 10^3$)	Number of workers W
`Stress-ng`	32, 64, 128, 256	1, 2, 4, 6
`Sysbench`	16, 32, 64, 128	1, 2, 4, 6

Table I shows the values of the parameters used for the workload generation. For each value of N we run experiments with all the values of W. Hence, we runs 16 different workloads. For each workload we did $n = 10$ runs, to account for system uncertainty, and we sample performance data each $\Delta t_{sample} = 1$ seconds. In that way we obtain a dataset to characterize the workload.

Figures 2 and 3 show the model of the workload generated with `sysbench` and `stress-ng` respectively. `stress-ng` generate a more intense workload than `sysbench`. To be more precise, the empirical probability distribution function of the `sysbench` workload shows that $U_{relative}$ range from 65% to 91% with the majority of sample between 80% and 90%. $U_{absolute}$ range from 82 to 91%. For the `stress-ng` workload $U_{relative}$ is in the range 76 – 91% and the $U_{absolute}$ in the range 90 – 94%.

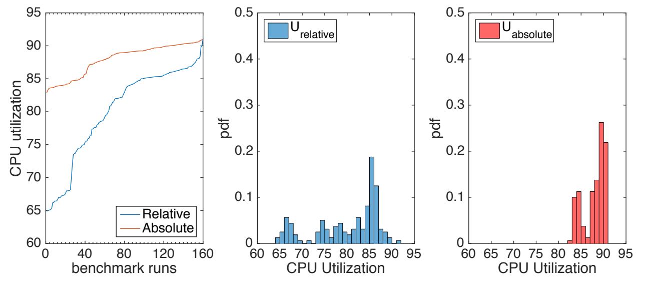

Fig. 2. The `sysbench` workload: *(left)* The range of value for the CPU utilization; *(center)* the probability distribution function (pdf) of the relative CPU utilization and *(right)* the pdf of the absolute CPU utilization.

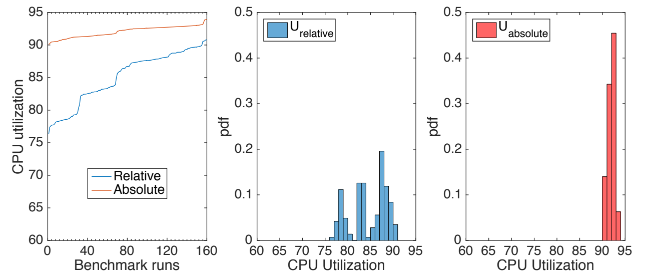

Fig. 3. The `stress-ng` workload: *(left)* The range of value for the CPU utilization; *(center)* the probability distribution function (pdf) of the relative CPU utilization and *(right)* the pdf of the absolute CPU utilization.

B. Experimental environment and monitoring tools

The CPU usage is measured running the `sysbench` and `stress-ng` workloads on a server equipped with: 2 CPU MD Turion(tm) II Neo N40L Dual-Core; 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.

To collect performance data we used three open source performance profilers: `mpstat`, `docker stats` and `cAdvisor`.

`mpstat` is a standard tools available for the Linux OS platform. It is part of the `sysstat` package and collect information from the Linux `/proc` virtual file system.

Performance statistics for the Docker containers are stored in the `/cgroup` virtual file system. `docker stats` is a Docker command, it runs in the Docker engine and it queries directly the `/cgroup` hierarchy. `docker stats` returns a live data stream for running containers, that is: the CPU utilization, memory used (and the maximum available for the container), the network I/O (data generated and received). It is possible to track all the containers or a specific one. `cAdvisor` (container Advisor) runs in a container and it uses the Docker Remote API to obtain the statistics. `cAdvisor` is a running daemon that, for each container, keeps resource isolation parameters, historical resource usage, histograms of complete historical resource usage and network statistics. To extract the data sampled by `cAdvisor` each 1 second we use `Prometheus` (prometheus.io), an open-source systems

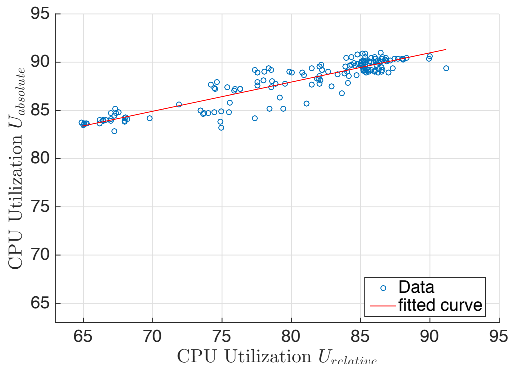

Fig. 4. The correlation between Relative CPU utilization (x-axis) and Absolute CPU utilization (y-axis) for the SysBench dataset

monitoring and alerting toolkit that scrapes metrics from instrumented jobs and store the resulting time series. Finally, `Grafana` (grafana.org) query the data extracted by `Prometheus` and enable the export and the visualization of data. The impact of those tools on the CPU utilization is negligible.

From the experiments conducted, emerged that `docker stat` and `cAdvisor` provide essentially the same performance measures.

IV. Correlation Model

The correlation $corr(U_{relative}, U_{absolute})$ between the relative CPU utilization and the absolute CPU utilization in our datasets is evaluated using:

- the Pearson's correlation coefficient (ρ) and the related 95% confidence bounds;
- the p-value (that gives a measure of the static significance of the correlation coefficient – if $p < 0.05$ ρ is statistically significant);

The value for the correlation coefficients is reported in Table II. The correlation coefficients are very similar and the 95% confidence interval overlaps (see lower and upper bounds in Tab. II). This is the range of values for ρ that contains with a 95% confidence the *true* correlation coefficient.

TABLE II
Correlation coefficients for the two datasets

Dataset	corr. coef. ρ (95% conf. bounds)	p-value
`Stress-ng`	0.9172 (0.8865, 0.9398)	0
`SysBench`	0.9170 (0.8882, 0.9386)	0

In Figures 4 and 5 the scatter plots visualize the correlation. The linear interpolation of the data points, determined using the linear fitting ($y = b + a \cdot x$), gives the expression for the correlation between the absolute and relative metrics, that is our correlation model:

$$U_{absolute} = b + a \cdot U_{relative} \qquad (2)$$

The values for the linear fitting coefficients and the 95% confidence bounds are in Table III.

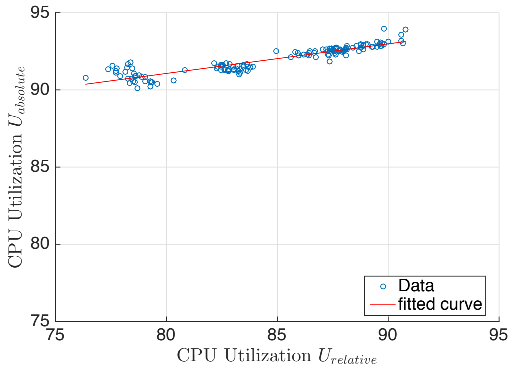

Fig. 5. The correlation between Relative CPU utilization (x-axis) and Absolute CPU utilization (y-axis) for the StressNG dataset

One of the drawback of our approach is that each workload has a different correlation model (with different values of a and b). Hence, the correlation model should be periodically evaluated, starting from run-time monitored data, and the linear correlation coefficients should be dynamically changed, according to the data collected.

TABLE III
LINEAR FITTING COEFFICIENTS FOR THE TWO DATASETS

Dataset	a (95% conf. bounds)	b (95% conf. bounds)
`Stress-ng`	0.1917 (0.1778, 0.2056)	75.73 (74.56, 76.91)
`SysBench`	0.3029 (0.2822, 0.3236)	63.71 (62.04, 65.37)

V. AUTO-SCALING PERFORMANCE EVALUATION

We have modified the KHPA algorithm (Algorithm 1) to work with absolute metrics. The new algorithm KHPA-A takes as input the target relative utilization U_{target} (expressed as relative metric), the linear fitting coefficients of the correlation model (cf. Table III and Eq. 2) and the set of active Pods $ActivePods$. 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.

The rational for our choice is that, with KHPA-A the DevOps teams can still work with relative metrics, that are more intuitive, and at the same time they can rely on an algorithm that allocate the proper amount of containers without the risk of Pods' under-provisioning.

In this study we focus only on the scale-up of Pods with the goal to maintain the average per-Pod target utilization when the resource demand increases. We do not consider neither the scaling down nor the stability mechanisms implemented by the Kubernetes autoscaler.

A. Performance metrics and experiments setup

The metrics used to compare the performances of the KHPA and KHPA-A auto-scaling algorithms are:

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($\tau$) // wait $\tau$ seconds, the control loop period
9: end while
```

- The average per-Pod absolute CPU utilization after adaptation $\overline{U}$, and
- The average per-replica application response time after adaptation $\overline{R}$

$\overline{U}$ and $\overline{R}$ are defined as

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

and

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

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.

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

- *Step 1)*, we fix the target utilization and we deploy 1 Pod.
- *Step 2)*, we randomly generated the per-Pod service demand, i.e. the relative CPU utilization $U_{relative}$, using the empirical pdf obtained from our datasets (cf Sec. III and Figures 2 and 3).
- *Step 3)*, each τ seconds we compute, using KHPA (and KHPA-A), the new number of Pods needed to match the target utilization and we deploy them. We assume that after a transient period the workload will be equally distributed among the deployed Pods.
- *Step 4)*, when the system is stable and before the τ control period expires we generate a new workload for all the deployed Pods, that is we jump back to *Step 2)*.

We simulate fifty control periods (i.e. 1500 seconds) and at each control period the total CPU demand is generated according to the probability distributions functions we have early determined for $U_{relative}$. However, we set an cap for the number of Pods, that is we do not deploy more than 1000 Pods. If that limit is reached the simulation is terminated.

To account for the randomness of the workload we repeat Steps 1 - 4 twenty times and we compute the average value of the metrics over the 20 runs.

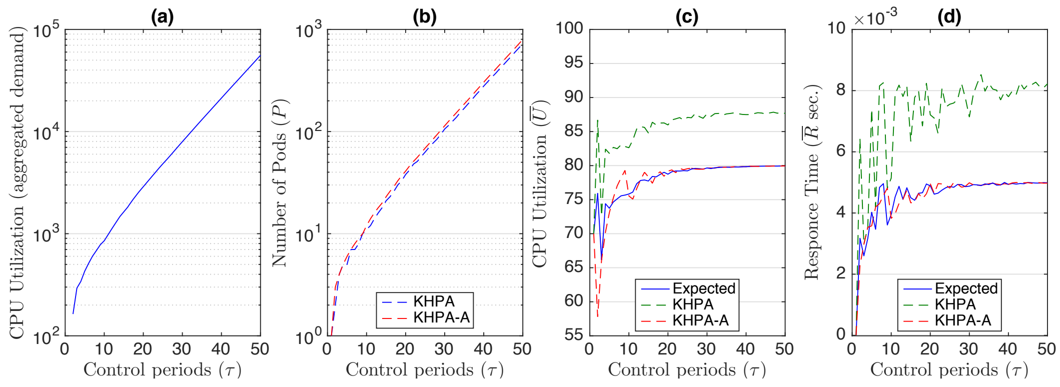

Fig. 6. Results for Sysbench workload, target utilization U_{target}=80%. For short in the legend U_{target} is replaced with T. (a) The system workload, that is the aggregated relative CPU utilization demanded by all the deployed Pods. (b) the number of deployed Pods P. (c) The average per-Pod CPU utilization $\overline{U}$ and the expected value. (d) The response time $\overline{R}$.

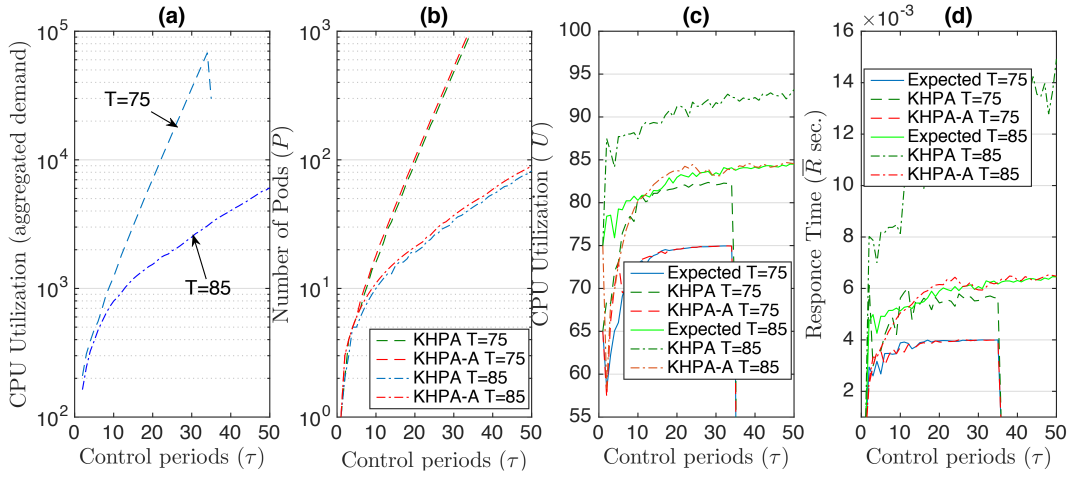

Fig. 7. Results for Sysbench workload, target utilization U_{target}=75% and 85%. For short in the legend U_{target} is replaced with T. (a) The system workload, that is the aggregated relative CPU utilization demanded by all the deployed Pods. (b) the number of deployed Pods P. (c) The average per-Pod CPU utilization $\overline{U}$ and the expected value. (d) The response time $\overline{R}$.

B. Experimental results

In the experiments we compute $\overline{U}$ and $\overline{R}$ for KHPA and KHPA-A. As baseline for comparison we use, respectively, the expected CPU utilization, that is the relative CPU utilization $U_{relative}$, and the expected per replica average response time (computed using $U_{relative}$).

For both workloads we considered three scenarios: U_{target} =75%, 80% and 85%. The service time is always $S = 0.001$ sec. (it has no impact on the results).

Figures 6 and 7 show 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 cup is reached after 35 control periods. This is why there is a drop to zero in the plots of the metrics for the 75% case. 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 $\overline{R}_{KHPA-A}$.

For the `stress-ng` workload we observed the same performance behavior as for `sysbench`. Figures 8 and 9 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 39 control periods, and for $T = 75\%$ after 29 control periods. With $T = 85\%$ the cup is not reached in the 50 control periods.

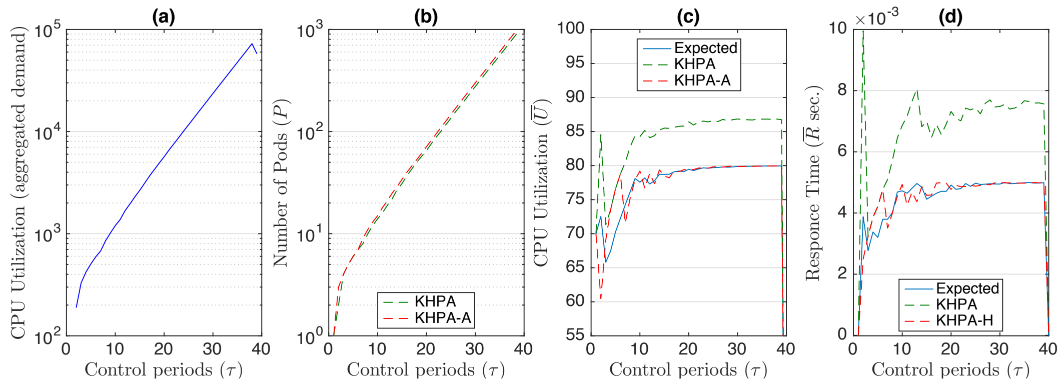

Fig. 8. Results for Stress-ng workload, target utilization U_{target}=80%. For short in the legend U_{target} is replaced with T. (a) The system workload, that is the aggregated relative CPU utilization demanded by all the deployed Pods. (b) the number of deployed Pods P. (c) The average per-Pod CPU utilization $\overline{U}$ and the expected value. (d) The response time $\overline{R}$.

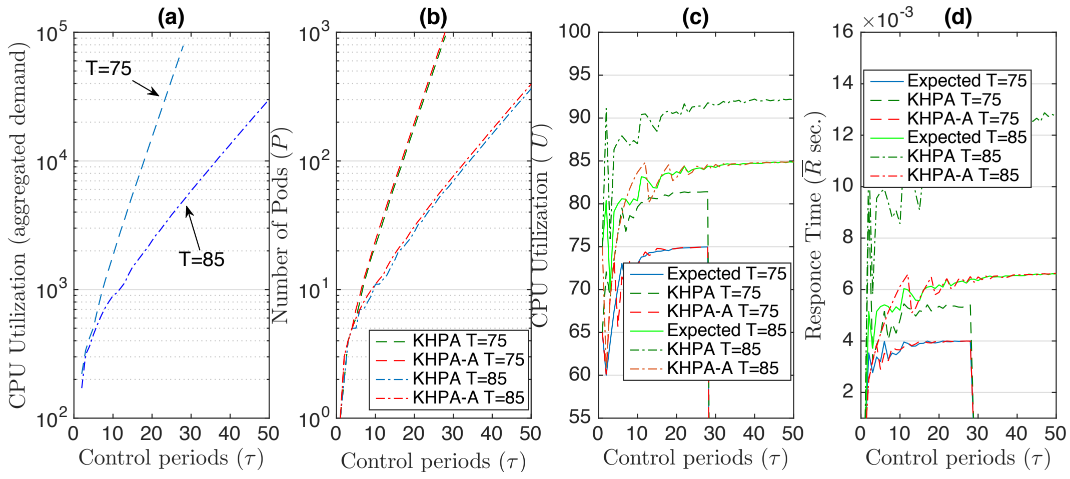

Fig. 9. Results for Stress-ng workload, target utilization U_{target}=75% and 85%. For short in the legend U_{target} is replaced with T. (a) The system workload, that is the aggregated relative CPU utilization demanded by all the deployed Pods. (b) the number of deployed Pods P. (c) The average per-Pod CPU utilization $\overline{U}$ and the expected value. (d) The response time $\overline{R}$.

VI. Related work

The idea of container dates back to 1992 [20] and matured over the year with the introduction of Linux namespace [21] and the LXC project (linuxcontainers.org), a solution designed to execute full operating system images in containers. From operating system virtualization, the idea has moved to application container [1]. The vantage point for containers are portability [7] and the negligible overhead compared to hardware virtualization [8], [9], [10], [11]. Containers technologies are strongly supported by PaaS [3], [12], IaaS [22] and Internet service providers [5] and are used to develop and deploy large scale applications for big data processing (e.g. [16], [23]) and scientific computing (e.g. [15]).

The adoption of container technologies call for new autonomic management solutions. An early study on container management is presented in [24]. Here the authors compare a VM-based and Elastic Application Container based resource management with regards to their feasibility and resource-efficiency. The results show that the container-based solution outperforms the VM-based approach in terms of feasibility and resource-efficiency. C-Port [25] is the first example of orchestrator that makes it possible to deploy and manage container across multiple clouds. The authors focus on dynamic resource discovery and selection constraints by availability, cost, performance, security, or power consumption. In [6] the author discussed the challenges in autonomic container orchestration proposing a reference architecture. In [23] the authors provide a general formulation of the elastic provisioning of virtual machines for container deployment as an integer linear programming problem, which takes explicitly into account the heterogeneity of container requirements and virtual machine resources. Only QoS and cost are considered in the problem formulation. In [26] the authors propose an adaptive multi-instance container-based architecture targeting time-critical applications. The solution is implemented using Docker and Kubernetes.

Our paper analyzes the behaviour of the Kubernetes Horizontal Pods Auto-scaling algorithm and proposes a new solution to make a more appropriate allocation of resources to fulfil application response time constraints.

VII. Conclusions

In this paper we have analyzed the behavior of the Kubernetes Horizontal Pod Auto-scaling algorithm, that work with relative metrics, and we have proposed a new autoscaling algorithms that base the scaling decisions on absolute metrics.

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 (absolute CPU utilization in the interval 85%-90%) the response time obtained with the new KHPA-A algorithm is lower than the response time obtained with the KHPA algorithm. Specifically, the response time in applications deployed with KHPA is approximately a factor between 1.5 and 2 higher than the response time obtained with the KHPA-A algorithm.

Although we focused only on CPU intensive workload and on Kubernetes' auto-scaling algorithms, the results of this work should be considered as an important guideline when implementing any container's auto-scaling algorithm. Future works will investigate the correlation of absolute and relative metrics with a wide range of workloads, and the effect of using relative and absolute metrics in other orchestration platforms, e.g. Swarm, Fleet, Marathon and Aurora.

Acknowledgment

This work is funded by the research project "Scalable resource-efficient systems for big data analytics" - Knowledge Foundation grant n. 20140032, Sweden.

References

[1] D. Merkel, "Docker: Lightweight Linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, Mar. 2014.

[2] M. Helsley, "Lxc: Linux container tools," *IBM devloperWorks Technical Library*, p. 11, 2009.

[3] R. Dua, A. R. Raja, and D. Kakadia, "Virtualization vs containerization to support PaaS," in *Proc. of 2014 IEEE Int'l Conf. on Cloud Engineering*, ser. IC2E '14, March 2014, pp. 610–614.

[4] D. Bernstein, "Containers and cloud: From lxc to docker to kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, Sept 2014.

[5] S. Natarajan, A. Ghanwani, D. Krishnaswamy, R. Krishnan, P. Willis, and A. Chaudhary, "An analysis of container-based platforms for nfv," IETF, Tech. Rep., April 2016.

[6] E. Casalicchio, "Autonomic orchestration of containers: Problem definition and research challenges,," in *10th EAI International Conference on Performance Evaluation Methodologies and Tools*. EAI, 2016.

[7] B. D. Martino, G. Cretella, and A. Esposito, "Advances in applications portability and services interoperability among multiple clouds," *IEEE Cloud Computing*, vol. 2, no. 2, pp. 22–28, Mar 2015.

[8] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and Linux containers," IBM, IBM Research Division, Austin Research Laboratory, Tech. Rep. RC25482(AUS1407-001), July 2014.

[9] R. Morabito, J. Kjällman, and M. Komu, "Hypervisors vs. lightweight virtualization: A performance comparison," in *2015 IEEE International Conference on Cloud Engineering*, March 2015, pp. 386–393.

[10] Z. Kozhirbayev and R. O. Sinnott, "A performance comparison of container-based technologies for the cloud," *Future Generation Computer Systems*, vol. 68, pp. 175 – 182, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X16303041

[11] E. Casalicchio and V. Perciballi, "Measuring docker performance: What a mess!!!" in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, ser. ICPE '17 Companion. New York, NY, USA: ACM, 2017, pp. 11–16. [Online]. Available: http://doi.acm.org/10.1145/3053600.3053605

[12] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes," *ACM Queue*, vol. 14, pp. 70–93, 2016. [Online]. Available: http://queue.acm.org/detail.cfm?id=2898444

[13] E. Truyen, D. Van Landuyt, V. Reniers, A. Rafique, B. Lagaisse, and W. Joosen, "Towards a container-based architecture for multi-tenant saas applications," in *Proceedings of the 15th International Workshop on Adaptive and Reflective Middleware*, ser. ARM 2016. New York, NY, USA: ACM, 2016, pp. 6:1–6:6. [Online]. Available: http://doi.acm.org/10.1145/3008167.3008173

[14] J. A. Zounmevo, S. Perarnau, K. Iskra, K. Yoshii, R. Gioiosa, B. C. V. Essen, M. B. Gokhale, and E. A. Leon, "A container-based approach to os specialization for exascale computing," in *First Workship on Containers 2015 (WoC)*, 03/2015 2015.

[15] W. Gerlach, W. Tang, K. Keegan, T. Harrison, A. Wilke, J. Bischof, M. D'Souza, S. Devoid, D. Murphy-Olson, N. Desai, and F. Meyer, "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*, ser. DataCloud '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 25–32. [Online]. Available: http://dx.doi.org/10.1109/DataCloud.2014.6

[16] D.-T. Nguyen, C. H. Yong, X.-Q. Pham, H.-Q. Nguyen, T. T. K. Loan, and E.-N. Huh, "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*, ser. IMCOM '16. New York, NY, USA: ACM, 2016, pp. 60:1–60:6. [Online]. Available: http://doi.acm.org/10.1145/2857546.2857607

[17] R. Zhang, M. Li, and D. Hildebrand, "Finding the big data sweet spot: Towards automatically recommending configurations for hadoop clusters on docker containers," in *2015 IEEE International Conference on Cloud Engineering*, March 2015, pp. 365–368.

[18] "stress-ng: a tool to load and stress a computer system." [Online]. Available: http://kernel.ubuntu.com/~cking/stress-ng/

[19] "Scriptable database and system performance benchmark." [Online]. Available: https://github.com/akopytov/sysbench

[20] R. Pike, D. Presotto, K. Thompson, H. Trickey, and P. Winterbottom, "The use of name spaces in plan 9," *SIGOPS Oper. Syst. Rev.*, vol. 27, no. 2, pp. 72–76, Apr. 1993.

[21] E. W. Biederman, "Multiple instances of the global Linux namespaces," in *2006 Ottawa Linux Symposium*, 2006.

[22] W. Vogels, "Under the hood of amazon ec2 container service." [Online]. Available: http://www.allthingsdistributed.com/2015/07/under-the-hood-of-the-amazon-ec2-container-service.html

[23] M. Nardelli, C. Hochreiner, and S. Schulte, "Elastic provisioning of virtual machines for container deployment," in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, ser. ICPE '17 Companion. New York, NY, USA: ACM, 2017, pp. 5–10. [Online]. Available: http://doi.acm.org/10.1145/3053600.3053602

[24] S. He, L. Guo, Y. Guo, C. Wu, M. Ghanem, and R. Han, "Elastic application container: A lightweight approach for cloud resource provisioning," in *2012 IEEE 26th International Conference on Advanced Information Networking and Applications*, March 2012, pp. 15–22.

[25] M. Abdelbaky, J. Diaz-Montes, M. Parashar, M. Unuvar, and M. Steinder, "Docker containers across multiple clouds and data centers," in *2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC)*, Dec 2015, pp. 368–371.

[26] V. Stankovski, J. Trnkoczy, S. Taherizadeh, and M. Cigale, "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*, ser. iiWAS '16. New York, NY, USA: ACM, 2016, pp. 453–457. [Online]. Available: http://doi.acm.org/10.1145/3011141.3011202