# When Less is More: Core-Restricted Container Provisioning for Serverless Computing

Gaetano Somma*, Constantine Ayimba‡§, Paolo Casari♯, Simon Pietro Romano*, Vincenzo Mancuso‡

*University of Naples "Federico II," Italy
‡IMDEA Networks Institute, Madrid, Spain
§University Carlos III of Madrid, Spain
♯DISI – University of Trento, Italy

***Abstract*—Cloud applications are exposed to workloads whose intensity can change unpredictably over time. Hence, the ability to quickly scale the amount of computing resources provisioned to applications is essential to minimize costs while providing reliable services. In this context, *containers* are deemed to be a promising technology to enable fast elasticity in resource allocation schemes.**

**In this paper, we propose and experimentally test an efficient container-based cloud computing provisioning system. First, we address the container deployment problem and discuss how to manage container provisioning and scaling. Second, we devise a resource management mechanism leveraging on both admission control and auto-scaling techniques. We propose to drive auto-scaling decisions through a Q-learning algorithm, which is agnostic to the specific computing environment, and proceeds based only on the load of the physical processors assigned to a container. We evaluate our solution in two experimental setups showing that it yields significant advantages when compared to popular container managers such as Kubernetes.**

***Index Terms*—Autoscaling, Provisioning, Q-Learning, Container, Docker, Kubernetes**

## I. Introduction

The adoption rate of Linux containers is growing at a rapid pace. This is mainly due to their comparatively simpler management with respect to virtual machines (VMs), and to the efficient resource utilization they enable. Moreover, the performance of containers is comparable to that of native computing platforms in terms of throughput and CPU utilization, whereas VMs typically impose non-negligible overhead [1], [2]. In fact, features like fast deployment, small boot time [2], easy network management and the use of layered, small-sized container images have fostered container adoption in global systems like the Google Cloud Platform and Amazon Web Services (AWS) [3]. The above characteristics make it possible to achieve rapid elasticity, namely the ability to quickly scale the amount of allocated resources according to workload intensity variations over time.

Managing container deployments in cloud environments is still an open issue that mainly involves container provisioning and scaling strategies. The performance of a container depends not only on how many CPU threads it is using, but also on which core of the CPU these threads are executed, as well as on resource contention against other applications. In order to make the cost calculation transparent to the user, such an inter-dependency must be avoided, and the relationship between the received level of service and the amount of used resources clarified. Existing container provisioning platforms delegate the scheduling of CPUs to the operating system. Consequently, since current operating systems are not hyper-threading aware, interference among running containers cannot be completely avoided, and load is typically distributed sub-optimally across CPU threads. This results in unpredictable service times.

Containers are currently managed using software platforms such as HPA by Kubernetes or Docker Swarm, which offer reliability by default [4]. However, some limitations exist regarding performance guarantees and adaptability to rapid changes in the operating environment. For instance, currently adopted solutions favor over-provisioning policies against rapid elasticity: this impedes system adaptation and means that capacity allocations are statically sized to serve peak loads, hence resources remain underutilized most of the time. However, we argue that over-provisioning is neither efficient nor strictly needed; in contrast, we show that automatic scaling is a preferred solution, provided that containers can be mapped onto hardware resources to avoid resource access conflicts.

The objective of our work is twofold: $(i)$ to design mechanisms that make access to computing resources simple and effective for container provisioning engines; and $(ii)$ to validate an adaptive scaling strategy based on reinforcement learning, which optimizes throughput and costs, without sacrificing the application's response time.

We implement *core pinning* to ensure that a CPU core is reserved for a container thereby forestalling contention side-effects due to hyper-threading. This approach also simplifies pricing models by making the cost of a container proportional to that of a CPU core. With regard to provisioning, we show that scaling the number of containers allotted to an application yields better performance than scaling the amount of resources allotted to a single container. Scaling the number of containers makes service time predictable, and thus provides the technical basis for, e.g., the stipulation of *Service Level Agreements* (SLAs) between service providers and customers. Moreover, we implement an automatic scaling system that predicts the required amount of resources and proactively makes scaling decisions in order to maximize the application workload processing throughput, and minimize the infrastructure allocation costs. Our automatic scaling subsystem is a Q-Learning agent. As Q-Learning is a model-free reinforcement learning technique, this agent can learn the operating environment autonomously, and adapt its scaling policies without any manual intervention.

The rest of the paper is structured as follows. In Section II, we discuss container provisioning challenges. In Section III, we propose a self-scaling provisioning system. In Section IV, we describe our experimental platform; we then discuss experimental results in Section V. We review the literature on container provisioning in Section VI, and draw final conclusions in Section VII.

## II. Container provisioning

In this section we explain how to avoid interference (i.e., contention in the access to shared computing resources) among running containers and explain how we operate to make service time predictable.

A Linux container is a group of isolated processes running on the host machine without any resource virtualization. A container can be granted an arbitrary amount of resources on the host machine: the amount of actually available resources depends both on the host capacity and on the resources allotted to other containers. For this reason, containers running on the same host will interfere with each other. The use of hyper-threaded CPUs results in additional interference. In fact, each CPU comprises multiple cores, each composed of two threads. These threads share the hardware for the execution phase. Hyper-threading leads to a performance improvement for each core, since it minimizes the impact of cache-miss interruptions. Unfortunately, such an architecture may also yield unpredictable performance, depending on which thread is used to run a process [5], [6]. In fact, as different threads in the same core share part of the architecture [7], execution performance is affected by other processes in the same core.

The solution to these problems is the use of resource limitation in conjunction with core pinning, as demonstrated in [8]. Concretely, this means that we dedicate one or more (entire) cores of a CPU to a given container. The container uses such cores exclusively, avoiding any interference with other processes. This also clearly specifies the number of resources used, as containers are mapped onto a known number of allocated cores. CPU core pinning also leads to efficient resource isolation [9], improved throughput and improved power utilization [10], and eliminates the interference related to L1 caching mechanisms, since each process stably runs on the same core. In practice, both resource limitation and core pinning can be achieved by leveraging the `cgroup` feature in Linux. Using this feature requires the specification of the threads to be used for each container. To avoid disparity in hyper-threaded architectures, we select threads belonging to the same core. For the Linux distribution we use in our testbed, this involves consulting the `cpuinfo` file. With pinning, the cost for a running container over $k$ time intervals can be computed as [11]:

$$\text{Cost}(k) = \alpha \sum_{n=1}^{k} C_{n,n-1} \cdot (t_n - t_{n-1}) \,, \qquad (1)$$

where $C_{n,n-1}$ is the number of cores dedicated to the container during the interval $[t_{n-1}, t_n]$, and $\alpha$ is a configurable cost scaling parameter chosen by the Cloud provider.

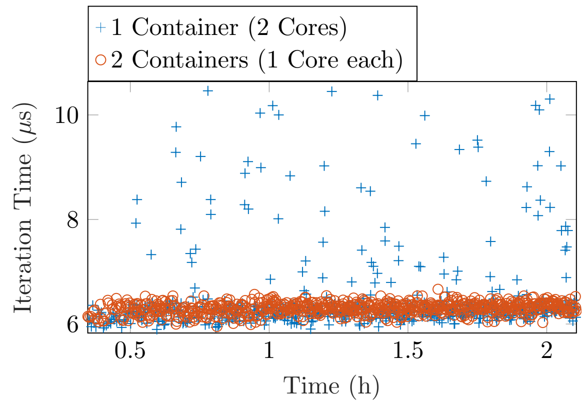



Fig. 1. Vertical vs. horizontal scaling: two containers running on distinct cores provide more predictable performance than one container running on two cores.

As regards scaling, two strategies are possible: (a) vertical scaling, i.e., adding or removing cores to a container as it is running; and (b) horizontal scaling, i.e., instantiating new containers or decommissioning active ones without adjusting their resource allocations. If vertical scaling is used such that multiple cores are assigned to a container, even with CPU core pinning on hyper-threaded cores, Linux's completely fair scheduler (CFS) might assign all processes to some cores, leaving the others in an idle state. This is because these schedulers are hyper-threading unaware, and may introduce interference among competing processes on the same container. Horizontal scaling, instead, allows fine-grained resource management and prevents intra-container interference. This effect is exemplified in Fig. 1 for a simple example reporting the iteration time observed when distributing compute jobs over either one container using two cores or two containers with one core each (to generate the figure, we used the iterative double 256-bit bitcoin hashing as an example of compute task).

In the next section we employ core pinning to spawn containers using different cores. In particular we allocate one core for each container.

## III. Automatic provisioning system

Here we exploit the container-provisioning approach to build a system capable of optimizing resource utilization. Our system scales the number of allocated containers to align with the varying demand in order to minimize costs while maintaining a high level of service. As a result of employing core pinning as explained in Section II, scaling decisions do not affect response time. A schematic diagram of our system is shown in Fig. 2. It comprises three components: Load Balancer (LB), Admission Controller (AC), and Auto-Scaler (AS). As suggested in [12], the LB component directs an admitted request to the active container reporting the most recent and lowest utilization rate. The AC component leverages CPU utilization stats from the `cgroup` file-system to decide whether an incoming request should be handled. The AS spawns new containers or removes active ones as appropriate, depending on demand.

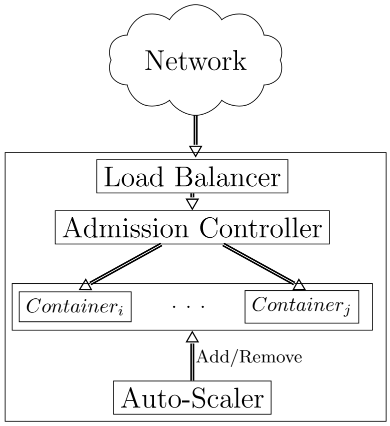



Fig. 2. Proposed container auto-scaling architecture.

We now delve into our design of the AC and AS components. Our LB implementation is inherited from [12].

### A. Admission controller

To remain profitable, cloud providers need to limit the amount of resources they allocate to each tenant, while still offering a sufficient amount to honor SLAs. When resources are constrained in this way, an AC is necessary to limit accepted requests, thus keeping the service time predictable and reducing SLA violations.

The behaviour of a container (which is CPU core pinned and thus restricted to one core) is shown in Fig. 3. As an exemplary cloud application, we employ again the iterative double 256-bit bitcoin hashing algorithm. Each request to this application triggers a different number of iterations. In Fig. 3, the response time is normalized to the number of iterations, and we show the mean iteration time, which is independent from the request's characteristics. The figure emphasizes that the relationship between the amount of allotted resources and SLA terms of service (such as the minimum response time) is not necessarily linear. In particular, there exists a discrepancy between CPU utilization (as reported by the operating system) and the actual occupied capacity of the core due to the use of hyper-threading [5], as the operating system considers two threads of the same core as two independent cores. Bearing this in mind and using the operating system metrics, a container observes a tri-stable CPU behavior: 0% CPU resource utilization when idle, 50% while continuously busy on a single core, 100% when continuously hyper-threading on the same physical core. The latter case points to saturation and unpredictable service times. The service time remains predictable only when a single thread (50% of a hyper-threaded core) is busy, as shown in Fig. 3. The above suggests that new requests should be admitted only when the container is not busy (0% utilization).

We finally remark that transients have a non-negligible impact on the correctness of admission decisions. Specifically, an admission error may happen in two cases: $(i)$ a request was just assigned to the container, but the reported CPU usage value is still close to 0% triggering the admission of an (otherwise undesirable) additional request; and $(ii)$ the container just finished to serve a request, but the reported CPU usage value is still close to 50%, triggering the rejection of a request that could instead be admitted. These transients in reported CPU utilization are typically short-lived such that the inter-arrival time of requests, even at peak time, is much longer in comparison. However, in order to further reduce the likelihood of the first event, a new request is admitted if the reported value is $\leqslant 25\%$. This solution works also for the second case.

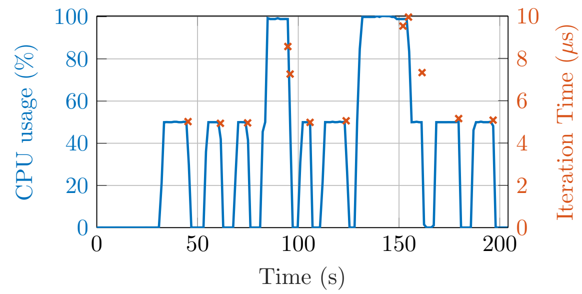



Fig. 3. Relationship between CPU utilization and service time. When reported utilization is $\leqslant 50\%$ iteration time is predictable.

### B. Auto-scaler

The purpose of the AS is to allocate the minimum number of containers that is commensurate to the demand, while still minimizing the number of dropped requests as reported by the empirical blocking probability. The scaling mechanism we employ is similar to the Q-learning paradigm used in [13]. Compared to that approach, we simplify the mechanism in order to expedite learning.[1] Furthermore, we quantize the state space into nine levels, from 0% to 45% in steps of 5%. The last level encompasses the range from 45% to 100% utilization. The latter detail is required because the admission control function ensures that utilization never exceeds 50%.

We designate the permissible scaling actions as: $-n$ (remove $n$ containers), $+n$ (add $n$ containers) and 0 (maintain the existing number of containers). For instance, for $n = 1$, we can thus describe the action space as set of possible options to increase, decrease or leave untouched the number of containers, as follows:

$$A = \begin{cases} [-1; 0; +1] & \text{if } 1 < N_t < M; \\ [0; +1] & \text{if } N_t = 1; \\ [-1; 0] & \text{if } N_t = M; \end{cases} \tag{2}$$

where $N_t$ is the current number of active containers and $M$ is the maximum number of containers that can be allocated.

## IV. Experiment Setup

We implement our provisioning scheme using Docker containers on a Dell T640 server with 20 hyper-threaded cores running Ubuntu 18.04. In order to expedite learning we choose $n = 1$ (cf. (2)) which effectively reduces the size of the action space. We implement CPU pinning and limit the maximum

[1] In particular, we set $\theta = 1$, the penalty of dropping a request in the *reward* function described in [13], so that only $\beta$, the cost of resources, is adjusted. The ratio of the two influences the policies learned.

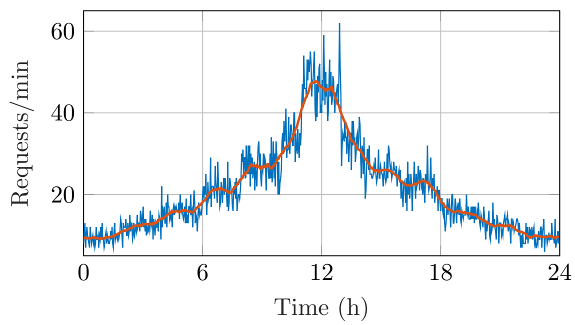


Fig. 4. Traffic profile observed during the Docker experiments. Traffic rates are taken over two-minute windows. The red line is the moving average over 30 samples.

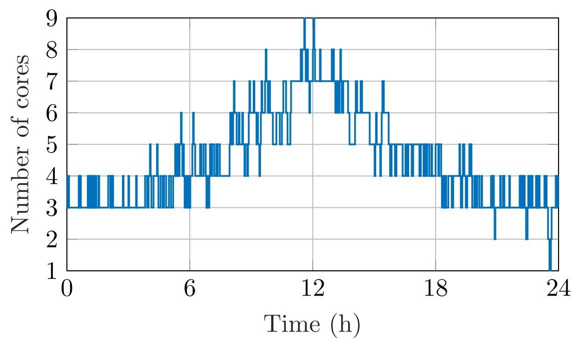


Fig. 5. Scaling decisions taken by Auto-Scaler algorithm during the Docker experiment at 20% of convergence (using $\beta$ = 0.02).

number of containers that can be provisioned, $M$ (cf. (2)), to 9. This ensures that server capacity is never exceeded and that the host processes run on an independent core. The AC and LB functions are implemented as python applications and run on the host core.

The server is connected to client PCs in an isolated LAN via a high-speed switch. Bash scripts on the client PCs spawn requests to the server with varying frequency at different hours of the day, so to mimic peak and off-peak periods typical of real-world traffic profiles.

A 24-hour cycle is split into hourly periods as shown in Fig. 4. Each period has inter-arrival times following a discrete distribution $\lambda \backsim U(0, \lambda_{\max})$. By varying $\lambda_{\max}$ we create a suitable peak/off-peak profile. The use of a uniform distribution ensures high entropy in order to evaluate the robustness of the schemes in challenging conditions. As mentioned in Section III-A, we deploy the double 256-bit bitcoin hashing algorithm as our cloud application. Admitted requests trigger different numbers of iterations to mimic the varying complexity of cloud applications.

We compare our provisioning scheme with Kubernetes, a widely used container management tool. Concretely, we use the Google Horizontal Pod Autoscaler for Kubernetes. We also benchmark our scheme with static over-provisioning and under-provisioning.

## V. Results And Discussion

We consider as comparative measures the following metrics: $(i)$ the saved cost, cf. (1), is the difference in cost between

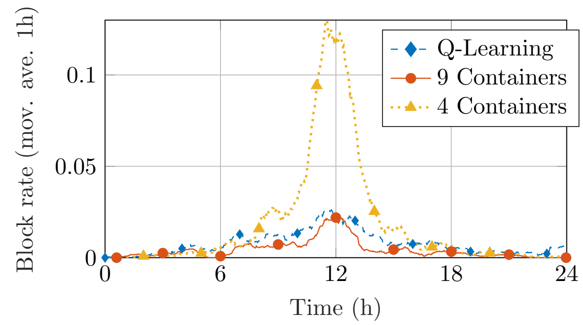


Fig. 6. The blocking rate observed over the time during Docker experiments in terms of rejected requests per second.

employing the maximum amount of resources throughout and using an auto-scaling algorithm to provision variable amounts of resources; $(ii)$ the service time, i.e., the time taken to process a request normalized by the number of iterations triggered by it, measured at the server side in order to exclude network effects; and $(iii)$ the blocking rate, a measure of service availability defined as the percentage of dropped requests with respect to those received by the server.

*1) Docker experiments:* We initially test our scheme in a Docker environment. With reference to the offered load presented in Fig. 4, our scheme generates the scaling profile shown in Fig. 5. Note that the goodness of scaling decisions is to be learnt, so that in Fig. 5 we show a snapshot, corresponding to 20% *convergence* (i.e., the proportion of states visited by the Q-learning provisioning agent for which the policy is fully learned is 20%). With the parameters we have adopted ($\theta = 1$ and $\beta = 0.02$ in the Q-learning mechanism described in [13]) the system learns to act quite conservatively with respect to blocking and encourages the provisioning of additional containers even with modest rises in the traffic profile. With this setting, our solution achieves 51% saved cost. The comparative performance in terms of blocking rate is shown in Fig. 6. Our scaler achieves blocking rates that are comparable to the over-provisioned case with 9 containers throughout the 24 hours. It outperforms by far the under-provisioned case in which only 4 containers are statically deployed and no adaptation is enforced over time. The saved cost for the under-provisioned case stands at 55% which is only marginally higher than that of our auto-scaler but with much poorer service availability, especially at peak traffic, as shown in Fig. 6.

From Fig. 7, it is clear that the number of containers deployed has an effect on the service time. This is due to the fact that container processes share the L2 and L3 cache memory. The greater the number of active containers the more pronounced the impact. For this reason, the over-provisioned case with 9 containers exhibits slightly higher service times, whereas the under-provisioned case with 4 containers yields the lowest service time. Our scaling solution suffers a small deviation from the low service times for about half of the cases owing to the instances when it provisions more than the benchmark 4 containers at peak traffic. However for all cases considered, the maximum difference in service times is

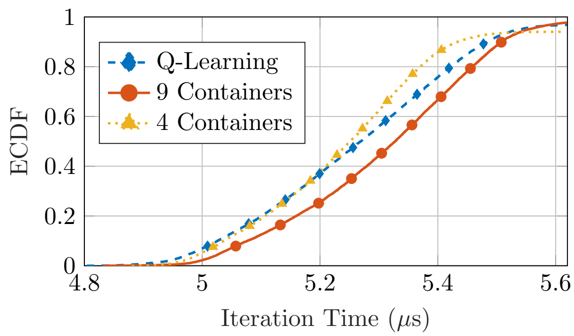



Fig. 7. Empirical CDF of the service time for Docker experiments.

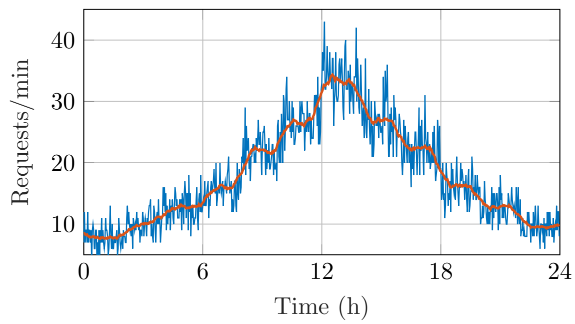



Fig. 8. Traffic profile observed during the Kubernetes experiments. Traffic rates are taken over two-minute windows. The red line is the moving average over 30 samples.

small with respect to the minimum values observed, i.e., the difference is less than 0.1 $\mu s$ out of $\sim$5 $\mu s$, for more than 93% of the cases. This is because admission control ensures that the system is not saturated.

*2) Kubernetes experiments:* We now compare the performance of our scaler against the commercial horizontal pod autoscaler (HPA) of Kubernetes. We re-run the experiments with the traffic profile shown in Fig. 8 using two computers with different specifications. Our scaler autonomously learns the appropriate operating conditions for each computer to trigger the addition or removal of containers. HPA however requires that the threshold be set as an external input. Such a setting is often a trial and error process and is both application and configuration dependent. To obtain comparable results to our scaler, we set this threshold as 28%. The comparison between scaling decisions of our algorithm and the ones made by HPA are shown in Fig. 9 (with our algorithm at 25% of convergence achieved). In these experiments we use two different servers which can guarantee different service times. In this situation our algorithm can be used to make the service time predictable, which is our goal. While observing the servers behavior, the service level that can be guaranteed by the system is the one guaranteed by the slowest machine, in this case the slave node.

While the two algorithms save the same amount of cost and achieve similar results in terms of service time, as depicted in Fig. 10, their blocking performance differs. Our approach

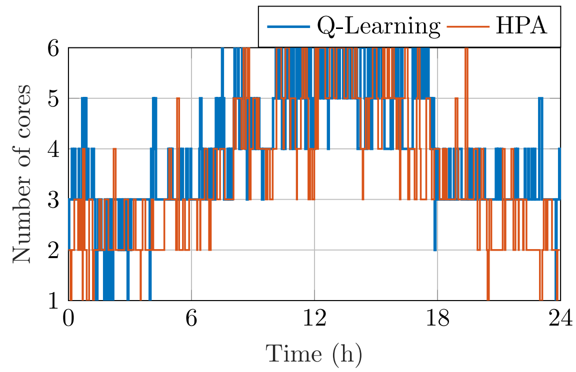
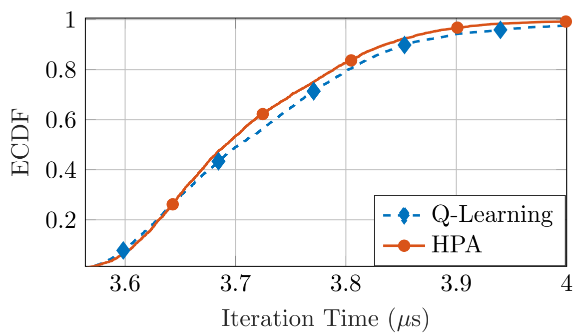
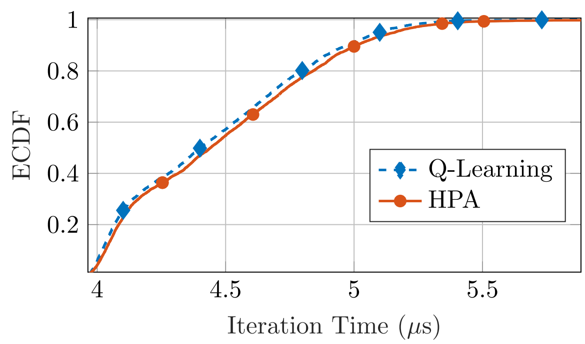



Fig. 9. Scaling decisions taken by the AS algorithm during the Kubernetes experiment at 25% of convergence (using $\beta$ = 0.02).




(a) Master node




(b) Slave node

Fig. 10. CDF of the service time for the Kubernetes experiments.

based on CPU core pinning and Q-learning largely outperforms HPA in terms of blocking rate, as shown in Fig 11.

## VI. Related work

The container provisioning problem is an evolving field of research given that this virtualization method is quite recent compared to virtual machines. The authors of [14] leverage the maturity of VM provisioning schemes and propose an Integer Linear Programming (ILP) optimal mapping between containers and the available VMs.

In [15], the authors consider containers hosted within VMs and propose a system able to coordinate the vertical scaling of both. The authors of [11] propose ELASTICDOCKER, a

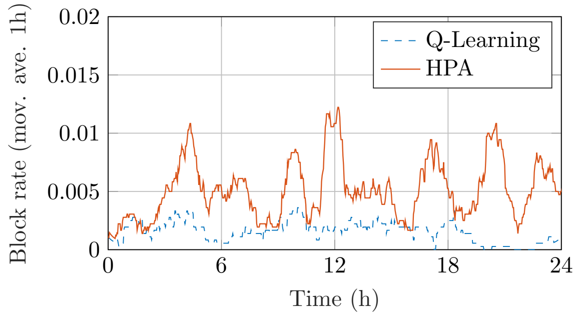



Fig. 11. Blocking rate observed over the time during Kubernetes experiments in terms of rejected requests per second.

vertical scaling system based on MAPE-K principles. Their scheme follows best-practices of setting thresholds to act as triggers for scaling operations. These works do not consider the effects of hyper-threading on vertical scaling and make assumptions based on queuing theory [16]. The prevalence of hyper-threaded CPUs therefore limits the performance of the proposed schemes [7].

In [17], Ye *et al.* propose a scheme which predicts the resource demands of an application and scales appropriately. An application-dependent, proactive resource provisioning scheme is proposed in [18]. The latter leverages the historical record of resource usage to trigger scaling actions.

The authors of [19] implement a horizontal scaler leveraging both reactive and proactive approaches. The amount of needed resources is calculated from a proactive term, based on a reactive threshold, and a proactive term, based on traffic forecasting. The latter is performed with a simple ARMA model and may be ineffective for rapidly varying traffic.

In [20] Sangpetch *et al.* carry out a comparative study of three auto-scaling algorithms based on either Q-Learning, artificial neural networks, or rules on thresholds. Q-Learning is found to achieve superior performance.

The preceding proposals do not consider CPU architecture and hence fail to capture the effect of hyper-threading on performance. Our approach takes this into account and curtails its pervasive effects by implementing CPU core pinning and leveraging horizontal scaling. These approaches coupled with our model free Q-Learning scaler result in a robust and configuration-agnostic scheme that ensures predictable response times regardless of the application under consideration.

## VII. Conclusions

We have presented a robust container provisioning system that leverages Q-Learning for autoscaling. We have demonstrated the consistent performance achieved by implementing CPU core pinning and horizontal scaling when compared to vertical scaling with hyper-threaded cores. We show that CPU core pinning simplifies the pricing models for cloud providers by facilitating an easy mapping between actual resources used and container resources assigned to tenants. Although our scheme curtails the application of hyper-threading and its advantages, the benefits of predictable and consistent high performance outweighs this disadvantage by far.

We have also demonstrated the superior performance of our scaling scheme when compared to the Kubernetes Horizontal Pod Autoscaler (HPA) widely adopted in container provisioning platforms. Our Q-Learning scheme attains predictable response times in the face of highly dynamic traffic without the need for manual threshold setting. It is able to autonomously learn the appropriate scaling triggers without prior knowledge of the system configuration or the cloud application.

## Acknowledgment

This work has received funding from the European Union's Horizon 2020 programme under GA No. 732667 (RECAP).

## References

[1] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in *ISPASS*. IEEE, 2015, pp. 171–172.
[2] K.-T. Seo, H.-S. Hwang, I.-Y. Moon, O.-Y. Kwon, and B.-J. Kim, "Performance comparison analysis of linux container and virtual machine for building cloud," *Adv. Sci. Tech. Lett.*, vol. 66, no. 105-111, p. 2, 2014.
[3] D. Bernstein, "Containers and cloud: From lxc to docker to kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, 2014.
[4] K. Hightower, B. Burns, and J. Beda, *Kubernetes: up and running: dive into the future of infrastructure*. " O'Reilly Media, Inc.", 2017.
[5] M. Tegtmeier, "CPU utilization of multi-threaded architectures explained," 2015. [Online]. Available: https://blogs.oracle.com/solaris/cpu-utilization-of-multi-threaded-architectures-explained-v2
[6] T. Leng, R. Ali, J. Hsieh, V. Mashayekhi, and R. Rooholamini, "An empirical study of hyper-threading in high performance computing clusters," *Linux HPC Revolution*, vol. 45, 2002.
[7] Y. Ding, E. D. Bolker, and A. Kumar, "Performance implications of hyper-threading," in *Int. CMG Conference*, 2003, pp. 21–29.
[8] C. D. Balaji Subramaniam. (2018) Future Highlight CPU Manager. [Online]. Available: https://kubernetes.io/blog/2018/07/24/feature-highlight-cpu-manager/
[9] C. Prakash, P. Prashanth, U. Bellur, and P. Kulkarni, "Deterministic container resource management in derivative clouds," in *IC2E*. IEEE, 2018, pp. 79–89.
[10] A. Podzimek, L. Bulej, L. Y. Chen, W. Binder, and P. Tuma, "Analyzing the impact of cpu pinning and partial cpu loads on performance and energy efficiency," in *CCGrid*. IEEE, 2015, pp. 1–10.
[11] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, and P. Merle, "Autonomic vertical elasticity of docker containers with elasticdocker," in *CLOUD*. IEEE, 2017, pp. 472–479.
[12] V. Cardellini, M. Colajanni, and P. S. Yu, "Dynamic load balancing on web-server systems," *IEEE Internet Computing*, vol. 3, no. 3, pp. 28–39, May 1999.
[13] C. Ayimba, P. Casari, and V. Mancuso, "SQLR: Short-Term Memory Q-Learning for Elastic Provisioning," 2019, arXiv:1909.05772 [cs.NI].
[14] M. Nardelli, C. Hochreiner, and S. Schulte, "Elastic provisioning of virtual machines for container deployment," in *SPEC*. ACM, 2017, pp. 5–10.
[15] Y. al dhuraibi, Z. Faiez, N. Djarallah, and P. Merle, "Coordinating vertical elasticity of both containers and virtual machines," 03 2018.
[16] W. Dawoud, I. Takouna, and C. Meinel, "Elastic virtual machine for fine-grained cloud resource provisioning," in *International Conference on Computing and Communication Systems*. Springer, 2011, pp. 11–25.
[17] T. Ye, X. Guangtao, Q. Shiyou, and L. Minglu, "An auto-scaling framework for containerized elastic applications," in *BIGCOM*. IEEE, 2017, pp. 422–430.
[18] H. Zhang, H. Ma, G. Fu, X. Yang, Z. Jiang, and Y. Gao, "Container based video surveillance cloud service with fine-grained resource provisioning," in *CLOUD*. IEEE, 2016, pp. 758–765.
[19] C. Kan, "Docloud: An elastic cloud platform for web applications based on docker," in *ICACT*. IEEE, 2016, pp. 478–483.
[20] A. Sangpetch, O. Sangpetch, N. Juangmarisakul, and S. Warodom, "Thoth: Automatic resource management with machine learning for container-based cloud platform." in *CLOSER*, 2017, pp. 75–83.