Optimizing Kubernetes Resource Requests

Oleksandr Kovalchuk
National Technical University of Ukraine
"Igor Sikorsky Kyiv Polytechnic Institute"
Kyiv, Ukraine
me.olexandr.kovalchuk@gmail.com

Yuri Gordienko
National Technical University of Ukraine
"Igor Sikorsky Kyiv Polytechnic Institute"
Kyiv, Ukraine
yuri.gordienko@gmail.com

Sergii Stirenko
National Technical University of Ukraine
"Igor Sikorsky Kyiv Polytechnic Institute"
Kyiv, Ukraine
sergii.stirenko@gmail.com

***Abstract*—Due to how Kubernetes scheduler works, improper CPU requests can lead to uneven load of servers and poor performance of workloads. This study focuses on how to set the optimal CPU request for Kubernetes workload. The presented solution allowed to increase CPU utilization and free more resources in cluster. Whereas it performed well for constantly-loaded workload, but for oddly-loaded ones applying the recommended values led to SLA degradation, while no significant CPU throttling was noticed. In order to apply the results in production, further research is required.**

***Index Terms*—Kubernetes, containers, Prometheus, CPU usage**

I. Introduction

Microservices attract more attention from industry and academia as a new service-oriented architecture. It is believed [1] that 91% of companies are using or have plans to use microservices. This means that instead of having a single monolithic application, companies deal with a lot of smaller services, which are deployed separately. According to Dimensional Research, 99% of companies which use microservices face difficulties operating those. And 74% of questioned companies plan to increase their microservice performance management investment.

It is known that large companies usually have a large number of microservices to support their business operations. Therefore, the task of optimizing costs for computing resources required to run microservices instances is a goal for platform providers.

Lately, Kubernetes (k8s) platform has spread as de-facto standard for deploying microservices in production. The platform provides such features as observability, healing, and scaling [2]. This is why k8s is chosen as a subject for the research.

While the overall aim of the conducted research is to build artificial intelligence assisted method to optimize resource utilization for k8s, this study focuses on ways to assist operators (developers, system administrators) in properly defining requests for CPU usage.

II. Glossary

CPU	Central processor unit
CPU Usage	Measurement of how much CPU time is used by running process.
CPU Utilization	Relation between how much CPU is used to how much CPU is available.
CPU Throttling	Adjusting the amount of CPU time for the process as part of resource management in order to distribute CPU time between all running processes.
SLA	Service Level Agreement. Defines allowed response times and error rate from service.

III. Literature Review

As the final objective of the conducted research is to develop a custom scheduler for Kubernetes platform, and beforehand implementing, it is necessary to understand how the default scheduler works. The main steps of pod (related group of processes) scheduling process are described in [3]. These steps are: filtering out unsuitable nodes, ranking nodes by priority functions, and finally, scheduling pod to the node. Most important predicates for node filtering as well as priority functions are mentioned. However, this book chapter [3] gives only a broad overview on scheduling strategy of the default k8s scheduler. Neither scheduler extension nor the usage of custom schedulers is described in the book, as well as no measures of the scheduler effectiveness are provided.

While Kubernetes scheduler is responsible for assigning processes to nodes, process scheduling within that node (server) is done using native Linux instruments. Interface, which controls resource allocation and isolation among groups of tasks is called control groups (cgroups), which can manage CPU, memory, network, disk I/O and various combinations of these resources. Resource configuration in control groups is done by editing cgroups subsystems parameters via synthetic cgroup filesystem [4]. One of the subsystems, responsible for CPU resource allocation is cpu subsystem. It operates different parameters, which can be later used by Linux Fair Scheduler (CFS). In order to ceil CPU usage, users can define cpu.cfs_quota_us parameter. After process or processes group used its CPU time defined in this parameter, they will be throttled and will not be allowed to use CPU time during given CFS period [5]. Kubernetes CPU limits are enforced via cpu.cfs_quota_us mechanism. For Kubernetes CPU requests another mechanism and another parameter is used, which is called cpu.shares. Parameter defines relative share of CPU time available to the processes in cgroup. For example, processes in two cgroups which have similar cpu.shares value will receive equal CPU time, but if cgroup A has $cpu.shares = 1024$ and cgroup B has $cpu.shares = 2048$, then processes in cgroup B will receive twice as much CPU time as ones in cgroup A [5]. Taking that into account, we can conclude, that improper usage of CPU requests in Kubernetes can lead to some processes have too small share and will be throttled.

Medel, Rana, Arronategui and Bañares in their research [6] mentioned that pods are scheduled under interference from other processes, and the interference must be considered. The authors mentioned such sources of interference as CPU usage, cache memory and memory bandwidth, network usage, and I/O file system access. However, the description of benchmarks the authors used is not definitive. Reference NET model developed allows simulating Kubernetes resource manager and determining application resistance to interferences, but it requires real-world usage data, which is collected by running benchmarks on the same node as application, and thus, is poorly suitable for predicting application behavior. In the study by Medel [7], the methodology to estimate the interferences was proposed.

While not being related with scheduling, service autoscaling provides possibility to react on workload changes and use the optimal amount of resources to sustain the surge. It allows keeping small amount of services running (thus, keeping resource usage as low as possible) and scaling up instances when the surge occurs. A set of key factors which should be considered in the development of auto-scaling methods is presented in [8]. The key factor analyzed in the article is CPU usage threshold. The authors experimentally showed that value between 80 and 100 for the threshold provides the best scaling performance while not triggering over-provisioning issue. Memory-based scaling is also discussed but proves to be more expensive than CPU-based one as the former usually requires the increased network bandwidth to transfer data. Thus, using high-memory machines in advance needs to be considered as a significant requirement in order to run memory-intensive applications. While the research is complete on these factors, they might not be suitable for all cases. For example, these criteria are not sufficient to effectively scale taskqueue workers, for such a scenario the different, custom metric is required.

IV. Methodology

In order to set proper resource requests for Kubernetes workloads we analyse workload's actual resource usage throughout last 14 days and calculate appropriate requests for the workload. In this research CPU usage request is chosen as the optimization target. Usage data are collected with the use of Prometheus monitoring, which was set up in the cluster by Prometheus operator [9]. Prometheus periodically scraps metrics from Kubernetes nodes and stores those, providing querying application programming interface.

The data is collected per-container for 14-days period. The following query template is used to collect the data:

```
kube_pod_labels{
    namespace="%(namespace)s",
    %(labels)s
}
* on (pod)
group_right
rate(
  container_cpu_usage_seconds_total{
    namespace="%(namespace)s",
    container="%(container)s"
  }[1m]
)
```

Collected data format is the array of pairs (timestamp, cpu usage). CPU usage is measured in seconds of single core usage per second.

Recommended CPU request is calculated as rectified linear unit (ReLU) of 95th percentile of the usage data over the observed period (1). Recommended values are then stored in the database.

$$N_{rec} = max(N_0, P95(d)) \tag{1}$$

where N_{rec} is the recommended value of CPU request, $P95$ is the 95th percentile, d is the usage data over the observed period (14 days), N_0 is the minimal request value, which is equal to 10 milliseconds of single CPU core time for process per second. Rectification is required in order to not set CPU request to the too small value, as this may cause performance issues. Value for rectification $N_0 = 10$ was chosen empirically, as with lower values, significant operational issues (applications were unable to respond even health check requests) were observed.

Recommended values are applied to the workloads during the subsequent workload creation or modification. For this purpose mutating admission webhook is used. When workload is created or modified, its specification is sent to webhook, configured in cluster, and can be modified, based on the webhook's response, and only after those modification, workload is created or modified in cluster [10]. Changes are only applied when recommended and actual requests differ more than minimal threshold (same as used in ReLU). Workloads are matched by namespace and name combination.

In order to analyse negative effects of the applied technique, CPU throttling time is analysed. High throttling times mean insufficient resources allocation for workload.

For the purposes of experiment, all workloads were changed by webhook almost simultaneously. In the real-world scenario it should be noted, that it might take some time for all workloads to be redeployed and modifications to apply.

Experiments were conducted at night under constant low load. Before and after making changes similar set of end-to-end (E2E) tests was run in order to create load and test the performance. Low load at the beginning of the experiment did not affect its result, as for calculations data for much longer period (14 days) were used.

V. Results

For each of observed namespaces, we managed to reduce requested CPU by 8-10 cores, which is 42 – 51% (Fig. 1) of the previously requested CPU cores. Slight decrease of the unused cores on the upper chart (timestamp: 05:40 – 05:45)

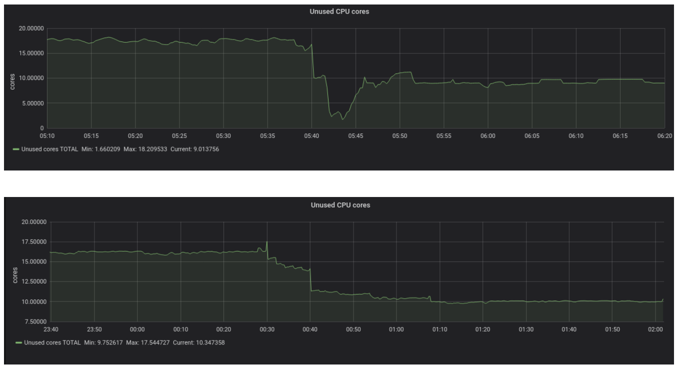

Fig. 1. Unused cores for tested namespaces. Lower is better.

which is latter restored is caused by shutdown and startup actions elasticsearch pods performed.

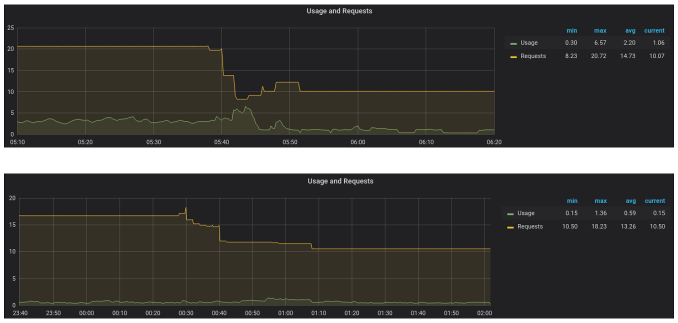

Fig. 2. CPU requests and CPU usage charts for tested namespaces. The closer requests (yellow) to actual usage (green) the better.

Application of calculated CPU requests increased CPU utilization by approximately 7% (Fig. 2). Rapid decrease in CPU requests on the charts corresponds to the moment of redeploying applications with modified values.

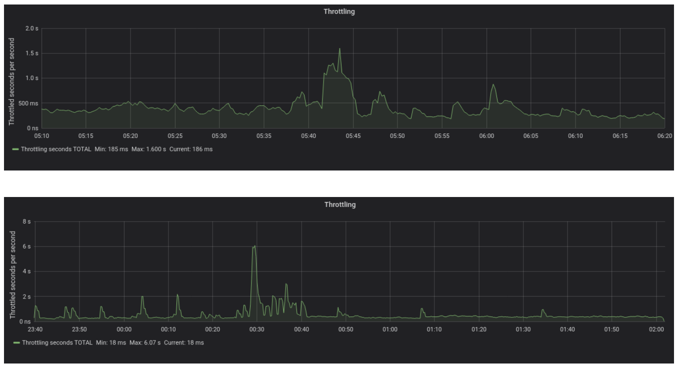

Fig. 3. CPU throttling for tested namespaces. Lower is better.

As for the throttling, we noticed the slight throttling increase in one of the tested namespaces, and the slight throttling decrease in the other (Fig. 3). This might be caused by either workloads tested (there were different kinds of workloads in different namespaces) or nodes performance (tested cluster is heterogeneous, and consists of nodes with different configurations). Despite experiment was conducted on servers with old kernel, which is affected by bug in CFS [11], CFS quotas were disabled on testing environment, thus results were not affected by this issue.

VI. Discussion

As it can be seen from charts, setting CPU requests per container to the values calculated from the previous CPU usage metrics did not cause negative effect on performance, whereas more free resources became available for other workloads. Additionally, after modifying workload definitions we preformed E2E testing to both generate load and check overall correctness. E2E testing showed increase of the network timeouts, meaning that lower requests actually caused increase of the response time. Whereas we run E2E testing regularly, it is either not often enough to impact overall CPU usage statistics. One of possible reasons of such performance degradation is that experiment was conducted on a limited set of processes (only two namespaces of approximately 70 were affected), thus making processes from other namespaces have bigger value of cpu.shares, then observed processes.

At the same time constantly loaded workloads showed no change in their performance, meaning that described method is suitable for constantly loaded workloads.

Throttling spikes on the charts happened at the moment of redeploying workloads with updated requests and correspond to the increased CPU load on shutdown/startup.

As the described method displayed different efficiency for different kind of workloads, the further investigation is required. Currently we investigate on different threshold values and percentiles when calculating recommended requests value. Nearest plans include analysing workloads type based on their CPU usage and applying different recommendation strategies for those. The last but not the least is investigating on different ways to modify requests and matching workloads by their characteristics instead of name.

VII. Conclusions

In this research we proposed a method of improving mechanics of setting requests for workloads based on the previous resource utilization of the workload. We conducted an experiment which showed that our approach is suitable only for constantly-loaded workloads. In order to properly configure workloads, whose load varies from time to time a lot a different approach is required. For constantly-loaded workloads effect is noticeable, and shows that in some cases it can save up to 50% of resources in Kubernetes cluster.

The further research will include modifying prediction formula (1) and investigating different ways to predict optimal CPU requests value for oddly-loaded workloads.

Acknowledgment

To V.H., for the support and involvement in testing the described method on their live Kubernetes cluster. To N.G., for providing support with E2E testing.

References

[1] D. Research. (2018) Global microservices trends. a survey of development professionals. [Online]. Available: https://go.lightstep.com/rs/260-KGM-472/images/global-microservices-trends-2018.pdf

[2] T. K. Authors, *Kubernetes Documentation — What is Kubernetes?*, 2020. [Online]. Available: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/

[3] D. Vohra, "Scheduling pods on nodes," in *Kubernetes Management Design Patterns*. Springer, 2017, pp. 199–236.

[4] P. Menage, *Linux Kernel Documentation — CGROUPS*, 2020. [Online]. Available: https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt

[5] R. Inc., *Resource Management Guide — CPU*, 2020. [Online]. Available: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/resource_management_guide/sec-cpu

[6] V. Medel, O. Rana, J. Á. Bañares, and U. Arronategui, "Adaptive application scheduling under interference in kubernetes," in *2016 IEEE/ACM 9th International Conference on Utility and Cloud Computing (UCC)*. IEEE, 2016, pp. 426–427.

[7] V. Medel, U. Arronategui, J. Á. Bañares, R. Tolosana, and O. Rana, "Modeling, characterising and scheduling applications in kubernetes," in *International Conference on the Economics of Grids, Clouds, Systems, and Services*. Springer, 2019, pp. 291–294.

[8] S. Taherizadeh and M. Grobelnik, "Key influencing factors of the kubernetes auto-scaler for computing-intensive microservice-native cloud-based applications," *Advances in Engineering Software*, vol. 140, p. 102734, 2020.

[9] B. Brazil, *Prometheus: Up & Running: Infrastructure and Application Performance Monitoring*. " O'Reilly Media, Inc.", 2018.

[10] T. K. Authors, *Kubernetes Documentation — Dynamic Admission Control*, 2020. [Online]. Available: https://kubernetes.io/docs/reference/access-authn-authz/extensible-admission-controllers/

[11] X. Pang, "sched/fair: Fix bandwidth timer clock drift condition," 2018. [Online]. Available: https://bit.ly/343D5uH