Client-Side Scheduling Based on Application Characterization on Kubernetes

Víctor Medel[1(✉)], Carlos Tolón[1], Unai Arronategui[1], Rafael Tolosana-Calasanz[1], José Ángel Bañares[1], and Omer F. Rana[2]

[1] Computer Science and Systems Engineering Department, Aragón Institute of Engineering Research (I3A), University of Zaragoza, Zaragoza, Spain
{vmedel,unai,rafaelt,banares}@unizar.es, ctolon@hotmail.com

[2] School of Computer Science and Informatics, Cardiff University, Cardiff, UK
ranaof@cardiff.ac.uk

Abstract. In container management systems, such as Kubernetes, the scheduler has to place containers in physical machines and it should be aware of the degradation in performance caused by placing together containers that are barely isolated. We propose that clients provide a characterization of their applications to allow a scheduler to evaluate what is the best configuration to deal with the workload at a given moment. The default Kubernetes Scheduler only takes into account the sum of requested resources in each machine, which is insufficient to deal with the performance degradation. In this paper, we show how specifying resource limits is not enough to avoid resource contention, and we propose the architecture of a scheduler, based on the client application characterization, to avoid the resource contention.

Keywords: Containers · Scheduling · Resource contention · Resource management

1 Introduction

With the advent of the cloud computing paradigm and the emergence of its technologies, computational power can be adjusted on demand to the processing needs of applications. Developers can currently choose among a number of cloud computational resources such as virtual machines (VMs), containers, or bare-metal resources, having each their own characteristics. A VM can be seen as a piece of software that emulates a hardware computing system and typically multiple VMs share the same hardware to be executed. Nevertheless, VM utilization can sometimes be difficult to achieve, e.g. when the applications to be run do not consume all the resources of a VM.

Containers are rapidly replacing Virtual Machines (VM) as virtual encapsulation technology to share physical machines [8,16,19,20]. The advantages over VMs are a much faster launching and termination time overheads, and an improved utilization of computing resources. Indeed, the process management origin of container-based systems allows users to adjust resources in a fine-grained fashion more closely with the granularity of many applications enabling single or groups of containers to be deployed on-demand [4]. Finally, container-based platforms, such as Kubernetes, also provide automating deployment and scaling of containerized applications, simplifying the scaling of elastic applications.

Nevertheless, as happened with VMs, containers also exhibit resource contention, which leads to unexpected performance degradation. In general terms, resource contention arises when the computing demand from the applications being executed exceeds the overall computing power of the shared host machine. In particular, resource contention appears in containers, when the demand of multiple containers in the same host machine exceeds the supply, understood in terms of CPU, memory, disk or network. This phenomenon can happen in spite of the isolation mechanisms integrated with container technologies, namely Linux namespaces and Linux Control Groups, which isolate the view of the system and limit the amount of computational resources, respectively. Therefore, the development of applications on these platforms requires new research on scheduling and resource management algorithms that reduces resource contention while maximizes resource utilization. Existing platforms like Kubernetes already incorporate a reservation mechanism in order to reduce resource contention. However, such mechanism is only for CPU and for the maximum amount of memory, and can decrease resource utilization.

In this paper, we propose a client-side scheduling approach in Kubernetes that aims at reducing the resource contention phenomenon in container technologies. Our approach makes use of application characterization in terms of the usage of resources, and extends the Kubernetes scheduler so that it can take better allocation decisions on containers based on such characterization. Our application characterization consists of dividing applications in two categories, namely high and low usage of resources and then, in this early stage work, we delegate the classification process of applications to the client or developer: He or she needs to provide the category which fits better into his/her application. Then, we extend the Kubernetes scheduler behaviour, in essence, we try to avoid that containers wrapping applications with high usage of a resource (i.e. CPU or disk) coincide in the same host machine. Finally, we conducted experiments with real-world applications, such as WordCount and PageRank, in operational stream processing frameworks, such as Thrill [6] and Flink [3], and compared the results with the standard Kubernetes scheduler.

The rest of this paper is structured as follows. In Sect. 2 a brief overview of related work is presented. Section 3 describes basic technological aspects of Docker and Kubernetes, and Sect. 4 shows the effects of resource contention. Section 5 presents our proposed architecture to deal with interference, and shows how an application characterization can help the scheduler to improve overall performance. Finally, our paper ends with conclusions and future work in Sect. 6.

2 Related Work

Container scheduling in cloud environments is an emergent research topic. Google has developed and used several schedulers for large scale infrastructures over past years based on a centralized architecture [5,21]. They are oriented for internal global use or as a global service provider. Some works have been proposed to improve the algorithms available as a standard in practical cloud infrastructures, such as Kubernetes[1], Mesos[2] [10] and Docker Swarm[3]. However, in [7], the authors point out the lack of works about resource management with containers, and they propose a scheduling framework that provides useful management functions that can be used to apply customized scheduling policies, mainly, in local environments. We can find few more works that complements to our approach: In [9], the authors propose a generational scheduler to map containers to different generations of servers, based on the requirements and properties learned from running containers. It shows an improved energy efficiency over Docker Swarm built-in scheduling policies. The work in [11] uses an ant colony meta-heuristic to improve application performance, also over Docker Swarm base scheduling policies. In contrast, in our paper, we consider both, resource utilization and application performance. The authors in [2] address the problem of scheduling micro-services across multiple VM clouds to reduce overall turnaround time and total traffic generated. Finally, in [1], the authors introduce a container management service which offers an intelligent resource scheduling that increase deployment density, scalability and resource efficiency. It considers an holistic view of all registered applications and available resources in the cloud. The main difference from our approach is that we focus on the client side requirements to optimize a subset of applications and resources.

Outside the container technologies, similar approaches exist. For instance, CASH [12] is a Context Aware Scheduler for Hadoop. It takes into account the heterogeneity of the computational resources of a Hadoop cluster as well as the job characteristics, whether they are cpu or I/O intensive. In [17], the authors use k-means as a classification mechanism for Hadoop workloads (jobs), so that jobs can be automatically classified based on their requirements. They plan to improve the performance of their scheduler by separating data intensive and computation intensive jobs in performing the classification. On the other hand, job interference was also studied in Hadoop, and acknowledged as one of the key performance aspects. In [23], the authors analyse the high level of interference between interactive and batch processing jobs and they propose a scheduler for the virtualization layer, designed to minimize interference, and a scheduler for the Hadoop framework. Similarly, the authors in [22] analyse the interference occurring among Apache Spark jobs in virtualized environments. They develop an analytical model to estimate the interference effect, which could be exploited for improving the Apache Spark Scheduler in the future.

[1] https://kubernetes.io/.
[2] http://mesos.apache.org/.
[3] https://docs.docker.com/swarm/, https://github.com/docker/swarmkit.

3 Kubernetes Architecture

Kubernetes is a platform to facilitate the deployment and management of containerized applications abstracting away from the underlaying infrastructure (see footnote 1). The system is composed of the Kubernetes *Control plane*, i.e. the master node and any number of *worker nodes* that execute the deployed applications. Applications are submitted to the control plane, which deploys them automatically across the worker nodes. Components of the control plane can reside on a single master node, or can be split across multiple nodes. The components of the control plane are: (1) The API server, implemented with a RESTful interface, which gives an entry point to the cluster. The API service is used as a proxy to expose the services which are executed inside the cluster to external clients. (2) The Scheduler that assigns a worker node to each component of your application. (3) *etcd*, a key-value distributed storage system, used to coordinate resources and to share configuration data of the cluster. And (4) The *Controller manager*, a process that combines and coordinates several controllers such as the replication controller, the node controller, the namespace or the deployment controller.

Each worker, which runs containerized applications, has the following components: (1) Docker, or any other container runtime. (2) The *kubelet* service that communicates with the master and the containers on the node. (3) *Kube-proxy* that balances client request across all containers that configure a service. The basic components considered on this paper are shown in Fig. 1. For a more detailed description of the chain of events triggered when a pod is created see [13].

The deployment unit in Kubernetes is a pod, an abstraction of a set of containers tightly coupled with some shared resources (the network interface and

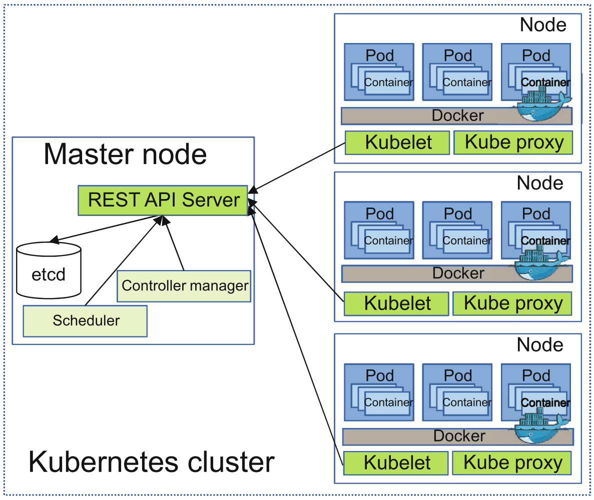


Fig. 1. Kubernetes architecture.

the storage system). Each pod, and all its containers, are executed on one allocated machine, and has a IP address that is shared by all containers. Therefore, two services listening on the same port cannot be deployed in a pod.

Developers can specify pod resource requests and limits. A pod *resource request* is the minimum amount of resources needed by all containers in the pod, and the pod *resource limits* is the maximum resources allocated to the containers in a pod. Once the pod is running on a node, it consumes as much CPU time as it can. CPU time is distributed between pods running on the pod in the same ratio than the pod request specifications. CPU is considered a compressible resource, which means that is acceptable a performance degradation due to a CPU resource contention. However, memory is incompressible, and it is not admissible for a pod to be running if it has not enough memory as requested. Consequently, developers should limit the amount of memory a container can consume.

The Kubernetes scheduler allocates pods into nodes taking into account factors that have a significant impact on the availability, performance and capacity – e.g. the cluster topology, individual and collective resources, service quality requirements, hardware and software restrictions, policies, etc. The scheduler uses request and limits to filter the nodes that have enough resources to execute a pod, and from them, it chooses the best one. Pods can be categories in three Quality of Service (QoS) classes: *Best effort* (lowest priority), *Burstable*, and *Guaranteed* (highest priority). The QoS classes is inferred from the request and limits manifests. A *Guaranteed* pod has all containers with limits equal to requests; a *Best effort* pod has not request or limit manifest for any container; and the rest of pods are *Burstable*. Once a pod is deployed in a node, if *pod request manifest* < *pod request limits* resource requested are guaranteed by the scheduler, but it is possible to use resources beyond the request manifest if they are idle resources.

4 Resource Contention on Kubernetes

When several containers are running on the same machine, they compete for the available resources. As the container abstraction provides less isolation than virtual machines, sharing physical resources might lead to a degradation in the performance of the applications running inside the containers.

To avoid this situation, Kubernetes provides a resource reservation mechanism. That mechanism has two main restrictions. The first one is that the reservation is only for CPU and for the maximum amount of RAM. However, the resources that are shared in a machine which might degrade the performance are not restricted to those ones. For instance, the network bandwidth is shared among all containers in the same machine, and the network access is shared for all containers inside a pod [15]. Other shared resource is the memory bandwidth. The second issue is that a reservation mechanism can lead to unused resources in the cluster. An application might reserve an entire core – CPU limit in Kubernetes terminology – but it only uses the resource sporadically.

Table 1. Reference applications used as a background workload with the resource which they use intensively and with the chosen execution parameters.

Application	Resource	Notes
Pov-ray	CPU	Version 3.7 with default parallelism
STREAM [14]	Memory bandwidth	- DSTREAM_ARRAY_SIZE=100000000 - DNTIMES=100
dd	Disk I/O bandwidth	dd if=/dev/zero of=/root/testfile bs=1G count=1 oflag=direct > dev/null

We executed several applications on the same machine to characterise how the performance degrades. The machine has two E6750 cores and 8 GB of RAM. The chosen applications are a map-reduce application, *WordCount*, and a webgraph application, *PageRank* [18], expecting that PageRank makes a higher usage of CPU than WordCount. Additionally, we ran both applications inside two different frameworks for data stream processing: Flink [6] and Thrill [3]. We chose both of them because they are implemented in different programming languages – Flink is implemented in Java, whereas Thrill is implemented in C++. We ran each experiment ten times, and we plotted their mean values.

The first set of experiments consists in running one application per experiment – WordCount inside Flink, WordCount inside Thrill, PageRank inside Flink and PageRank inside Thrill – along with a background execution caused by another application which makes an intensive usage of a certain resource: a ray tracing program, Pov-ray[4], the STREAM benchmark [14], and a file transfer and conversion Unix command, dd[5], are used as workload background applications. These three applications were executed in a continuous loop. A summary of the parameters used, their version, as well as the resource they use intensively is depicted in Table 1.

For the experiments, we ran WordCound and PageRank and varied the input size in order to observe how their performance degrades for long executions. For PageRank applications, we selected the Barabasi-Albert graph which was generated using the NetworkX package[6]. As a reference time, we take the execution time of each application in isolation, without the background application, App_0. Given the execution time of that application with a certain background workload, for instance App_{pv}, we calculate the performance degradation as $\frac{App_{pv}}{App_0}$. Results are shown in Fig. 2. We can see that: (i) the implementation of Thrill is much more efficient than Flink in all cases; (ii) there is a significant performance degradation when we execute WordCount in all cases for big input sizes – 1 thousand million –, which is caused when dd is very high (about four times). The explanation is that the size of the input is 6.76 GB, so there are a lot of

[4] Persistence of vision raytracer (version 3.7) [computer software], http://www.povray.org/download/.

[5] dd(1) linux user's manual (2010).

[6] https://networkx.github.io/.

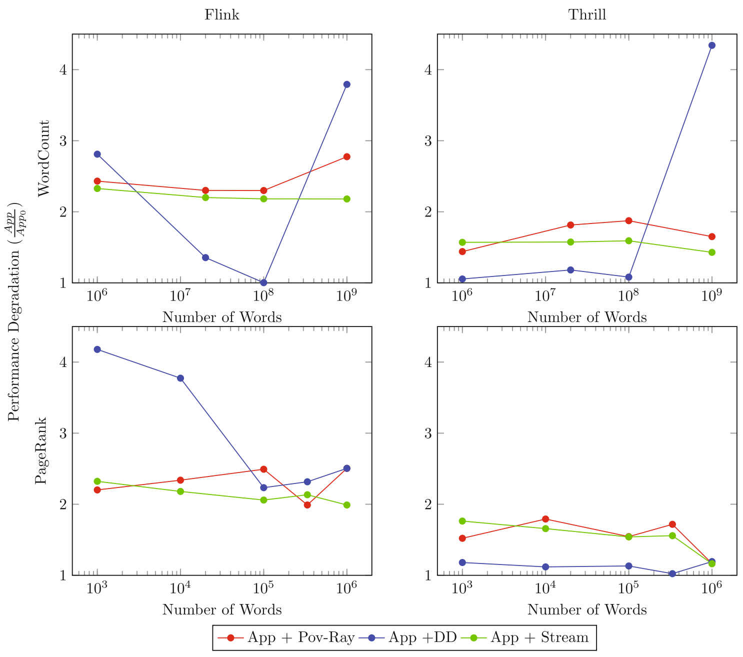


Fig. 2. Performance degradation for several Apps – WordCount inside Flink, Word-Count inside Thrill, PageRank inside Flink and PageRank inside Thrill – with a background workload.

page faults in the execution and the application is continuously accessing to the storage system. (iii) Finally, there is also a significant performance anomaly when executing WordCount and Flink and dd for small input sizes. This is due to internal implementation of Flink regarding I/O access, as for such small input sizes the computational times are reduced in comparison with the overheads for accessing disk.

In the second set of experiments, we have measured the degradation caused in real scenarios. We execute on the same physical machine the following scenarios for each application – WordCount and PageRank–: (i) one instance of Flink – Thrill–; two instances of Flink –Thrill– and four instances of Flink –Thrill. (ii) one instance of Flink + one instance of Thrill; two instances of Flink + two instances of Thrill. Results are shown in Fig. 3. We can observe that in Flink, the degradation is similar when there is another Flink or Thrill application. When there are four applications, the performance is degraded in a high degree with Flink applications in WordCount example. The results are very similar in Thrill experiments.

In general, we can see that the degradation is higher when two or more instances of the same container are scheduled in the same machine. The reason for this behaviour is that both applications make use of the same resources at the same time, so the contention is higher.

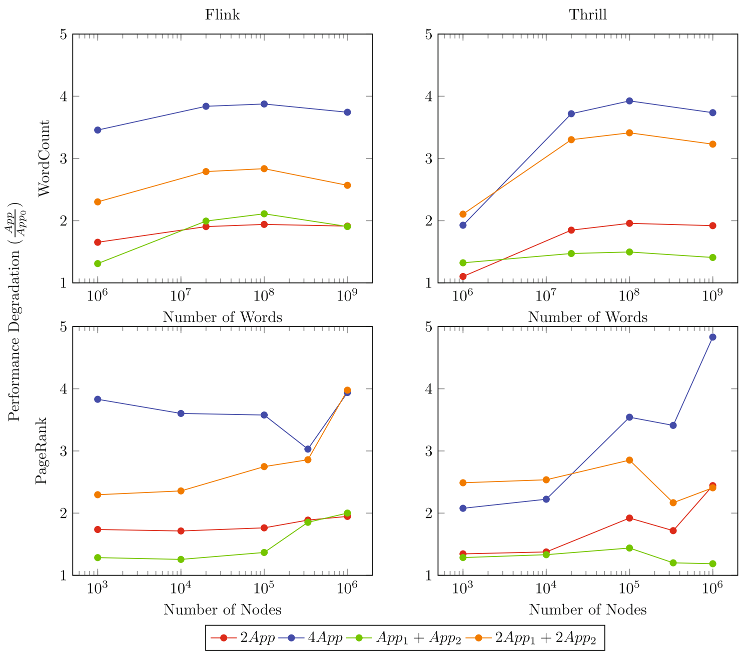


Fig. 3. Performance degradation for several Apps – WordCount inside Flink, WordCount inside Thrill, PageRank inside Flink and PageRank inside Thrill – when executed in different configurations.

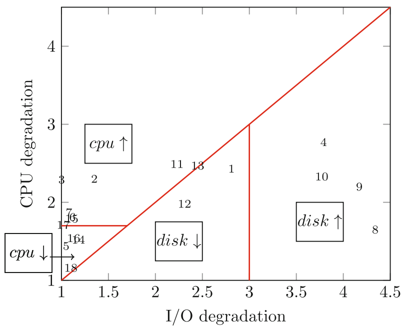
5 Client-Side Scheduling

As we presented in Sect. 3, pods are allocated into machines by Kubernetes Scheduler. Kubernetes provides a label mechanism, which allows it to place the pods in machines which satisfy certain conditions. For example, the application can request a machine with a solid disk. However, the client should know which kind of labels the cluster provides. This mechanism is insufficient to deal with the problem presented in Sect. 4. In this section, we introduce a methodology to characterize applications in an informal way. The implemented client-side scheduler uses the characterization as a guideline to allocate pods inside machines.

5.1 Application Characterization

In certain cases, applications can be classified depending on which resource they use more intensively – CPU, I/O disk, network bandwidth, or memory bandwidth. An application which is writing in disk continuously has a different behaviour from another one which makes an intensive use of CPU. For the lack of space, in this paper, we only consider applications that make an intensive use of CPU or an intensive use of I/O disk. In our previous experiments, `pov-ray` was the application which exemplifies a high CPU application and the `dd` command exemplifies a high I/O disk utilisation. Real applications might have an intensive usage of a resource, but with different degrees. For example, the bzip application, used to compress large files, has an I/O intensive behaviour that is less than the usage made by the `dd` example. This behaviour can be modelled which the definition of several intensity usage grades. For the sake of simplicity and as we want to propose a general methodology, we are going to use here only two grades of resource usage, a high usage of the resource – with a $\uparrow$ notation – and a low usage of the resource – with a $\downarrow$ notation. Nevertheless, we acknowledge that the number of grades is a determinant aspect for the scheduling performance that needs to be addressed, and there is a number of approaches in the literature that can be exploited for better determining it, such as classification and clusterization data mining algorithms.

Therefore, in our approach, we have a total of four categories: High cpu utilisation (*cpu* $\uparrow$), low cpu usage (*cpu* $\downarrow$), high I/O disk usage (*disk* $\uparrow$), and low I/O disk usage (*disk* $\downarrow$). The characterization of an application in one of these categories is going to allow the scheduler to take better allocation decisions. As the simplest method, the client or the developer should provide the category which better fits better his/her application. Although the categories are very intuitive, alternative sophisticated methods can be developed to classify applications automatically. In order to illustrate our methodology, in Fig. 4, we show a possible characterization. We have plotted the I/O degradation – the number of times the application is slower when it is scheduled in the same machine along with `dd`– vs the CPU degradation – the same procedure using `pov-ray`. We used `dd` and `pov-ray` as benchmarking applications, however, other applications which make a high usage of a single resource can be used. The values were taken from the experiments shown in Fig. 2. The red lines split the four categories, and they were obtained with qualitative criteria. Then, we classified each application



(a) CPU degradation vs. I/O Degradation.

App	Input Size	Id	Categoría
FlinkWC	$1 \cdot 10^6$	1	*cpu* $\uparrow$
FlinkWC	$20 \cdot 10^6$	2	*cpu* $\uparrow$
FlinkWC	$100 \cdot 10^6$	3	*cpu* $\uparrow$
FlinkWC	$1000 \cdot 10^6$	4	*disk* $\uparrow$
ThrillWC	$1 \cdot 10^6$	5	*cpu* $\downarrow$
ThrillWC	$20 \cdot 10^6$	6	*cpu* $\uparrow$
ThrillWC	$100 \cdot 10^6$	7	*cpu* $\uparrow$
ThrillWC	$1000 \cdot 10^6$	8	*disk* $\uparrow$
FlinkPR	1000	9	*disk* $\uparrow$
FlinkPR	10000	10	*disk* $\uparrow$
FlinkPR	100000	11	*cpu* $\uparrow$
FlinkPR	334863	12	*disk* $\downarrow$
FlinkPR	$1 \cdot 10^6$	13	*cpu* $\uparrow$
ThrillPR	1000	14	*cpu* $\downarrow$
ThrillPR	10000	15	*cpu* $\uparrow$
ThrillPR	100000	16	*cpu* $\downarrow$
ThrillPR	334863	17	*cpu* $\uparrow$
ThrillPR	$1 \cdot 10^6$	18	*cpu* $\downarrow$

(b) Application Identification and Characterization.

Fig. 4. Application characterization based on the CPU degradation and the I/O degradation. Numbers in subfigure a are application identifiers in table b. (Color figure online)

Algorithm 1. Client-Side Scheduler

```
1: procedure Client-Side Scheduler($l_{app}, W$)
2:     $\boldsymbol{S} = GetClusterState()$
3:     $minValue := \infty$
4:     $bestNode := 0$
5:     for $N$ in $\boldsymbol{S}$ do
6:         if $|N| \leq min\{|M|, \forall M \in \boldsymbol{S}\}$) then
7:             if $minValue > \sum_{j}^{|N|} w_{j,app}$ then
8:                 $minValue := \sum_{j}^{|N|} w_{j,app}$
9:                 $bestNode := N$
10:            end if
11:        end if
12:    end for
13:    $Allocate(l_{app}, bestNode)$
14: end procedure
```

Table 2. Example weight matrix W for two resources and two usage grades.

$App_1 \backslash App_2$	$cpu \uparrow$	$cpu \downarrow$	$disk \uparrow$	$disk \downarrow$
$cpu \uparrow$	5	4	2	1
$cpu \downarrow$	4	3	1	0
$disk \uparrow$	2	1	5	4
$disk \downarrow$	1	0	4	3

taking as criteria the resource which caused more contention. The plotted numbers are the identifier of the corresponding application, which are shown in Fig. 4b.

5.2 Client-Side Scheduling

We propose a scheduler design which has two criteria: (i) Balancing the number of applications in each node; (ii) minimising the degradation in a machine caused by the resource competition. Formally, let us define a node N as a multi-set of labels. Each label represents an application that is running inside that node. In our example, we have four kind of labels – l_0 equivalents to *cpu* $\uparrow$; l_1 equivalents to *cpu* $\downarrow$, and so on. In a certain moment, the state of the cluster $\boldsymbol{S}$ can be modelled as a set of nodes. Given a new application whose label is l_{app}, the best node to allocate l_{app} is given by:

$$\underset{i \in 0}{\overset{|E|}{\operatorname{argmin}}} \sum_{j} w_{E_{i,j},app}$$

where $w_{k,l}$ is the weight of the *k-th* row and *l-th* column of a weight matrix W. Each $w_{k,l}$ models the penalty to schedule a new application labelled as l, if in that node is running an application labelled as k. $E_{i,j}$ is the j-th application label of i-th node in E set. E is defined as $E = \{N \in \boldsymbol{S} \wedge \forall M \in \boldsymbol{S}, |N| \leq |M|\}$. The E set contains the nodes with less applications. Algorithm 1 implements the previous formalisation.

In order to obtain the weight matrix W, we provide the following rules: For each element $w_{k,l}$, we observe if the labels are associated with the same resource. If that is the case, then we set high values of penalty: 3, 4, or 5. Then, we observe the grade of usage. From the previous values, if both grades are high we set the highest penalty value (5); if only one is high, then we associate the medium value (4), and if both are low, then the lowest penalty value is set (3). In case the labels are not associated with the same resource, we repeat the same process to associate the low values (0, 1, or 2) if i and j correspond to different resources. From the experiments made in Sect. 4, we can build a weight matrix W, as depicted in Table 2.

5.3 Experiments

We made some experiments to compare our client-side scheduler with the default Kubernetes scheduler. The proposed scheduler was implemented in Python. The experiments were run in a Kubernetes cluster with 8 machines (each machine has four i5-4690 cores and 8 GB of RAM). One of the machines acts as a dedicated Master Node. In the proposed scenario, we ran six applications three times – `dd` and `pov-ray` with parameters from Table 1; PageRank in Thrill and Flink with 1 million nodes and WordCount in Thrill and Flink with 1,000 million words– with the default Kubernetes scheduler. The scenario was executed ten times. As the Kubernetes scheduler has a non-deterministic behaviour, we show three reference cases in Fig. 5. Each bar represents the execution time of the application, and its colour indicates the machine where the scheduler placed the application. The vertical line shows the total time measured for the experiment (time to create the pods plus execution time plus time to delete the pods). Case number 1 represents the worst case. Kubernetes allocated `WCFlink1` and `WCFlink2` in the same machine, with an execution of `dd` and `PRFlink1`. As a result, the execution time of `WCFlink1` is more than 10 min, due to the degradation caused by sharing the machine with `WCFlink2`. Case number 2 represents a balanced case, with an execution time about 10 min. The Scheduler placed again `WCFlink3` and `WCFlink2` in the same, so there is certain degradation in the performance. The best case corresponds to Case 3. In this situation, Kubernetes allocated `WCFlink1`, `WCFlink2`, `WCFlink3` in different machines and the result is better (about eight minutes). From these experiments, we can conclude that, as Kubernetes has a non-deterministic behaviour, the execution time of the applications has a high variance. If the scheduler splits the applications with an intensive CPU usage along different machines, the results are better; however the decision is taken randomly. Additionally, we can see in Case 2 that the default Scheduler does not try to balance the number of applications along the number of machines – the scheduler places four applications in node3 and only one application in node5.

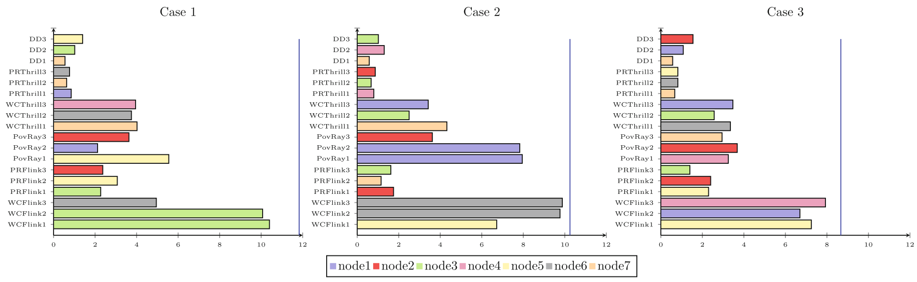


Fig. 5. Execution time and machine allocation with the default Kubernetes scheduler. The blue line shows the total measured time (Execution time + time to create pods + time to delete pods). (Color figure online)

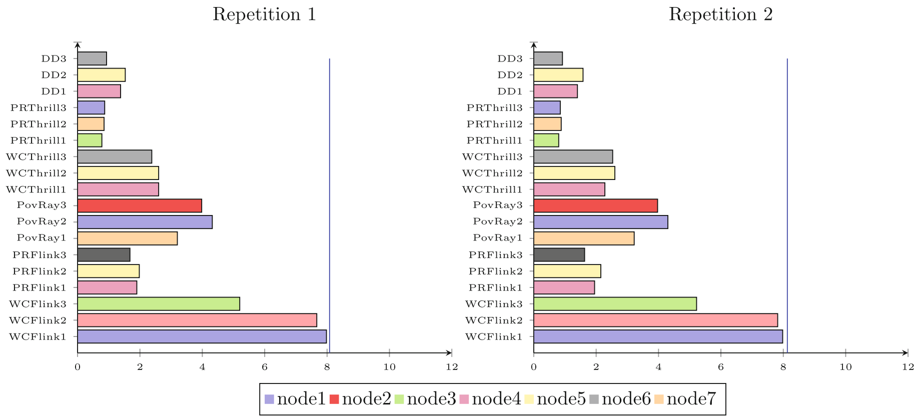


Fig. 6. Execution time and machine allocation with the proposed scheduler. The blue line shows the total measured time (Execution time + time to create pods + time to delete pods) (Color figure online)

The results of the same experiment with our scheduler are shown in Fig. 6. Its behaviour is deterministic, so under the same conditions, the scheduler allocates the applications in the same machine – in the figure we only show two of the ten executions, due to the low variance. The overall execution time of the experiment is about eight minutes. This value is 20% better than the mean time of the Kubernetes scheduler –about 10 min–, and it is significantly better –33%– than the worst case – about 12 min –. The total time is similar to the best case of the default scheduler. Additionally, the variance in the execution time is lower. The improvement is achieved splitting the application with a high CPU utilisation –`WCFlink1`, `WCFLink2`, and `WCFlink3`– in different machines.

In our last set of experiments, we executed the same batch of applications using the reservation mechanism available in Kubernetes. As `WCFlink1`, `WCFlink2`, `WCFlink3` have the highest execution time, we reserved two cores for them. For the rest applications, we reserved only one core. The results are dis-

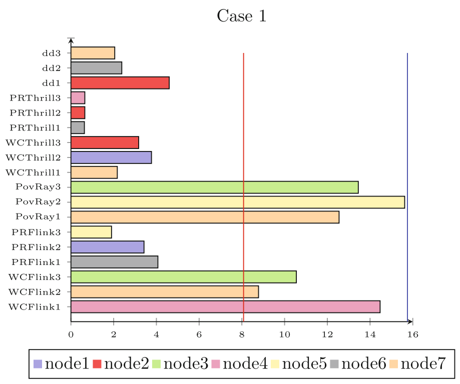


Fig. 7. Execution time and machine allocation with CPU limit mechanism. `WCFlink1`, `WCFlink2`, `WCFlink3` are executed with two cores and the rest application with one core. The blue line shows the total time obtained with the proposed scheduler and the red line shows the total measured time (Execution time + time to create pods + time to delete pods) (Color figure online)

played in Fig. 7. The red line compares the total time with our scheduler with the blue line. We can see that the total time is almost twice. The reason for this behaviour is that there are a lot of unused resources in the machines. Additionally, the variance of the execution time is very high – for instance, `pov-ray1` has an execution time about 12 min and `pov-ray2` has an execution time about 16 min–. It can be explained due to the fact that there are other resources that cause performance degradation which are not reserved.

6 Conclusions and Future Work

Container virtualization provides a quick and flexible mechanism to share computational resources in machines, while improving resource utilization as compared to other Cloud resource such as Virtual Machines. However, the low isolation between container based applications can lead to performance degradation in those applications. In our paper, we have shown that the default mechanism to isolate resources between containers in Kubernetes are not sufficient to lead with the performance degradation. Although CPU is the main source of degradation, the competition for other resources – I/O disk, memory bandwidth and network – should be included in the model. Moreover, our experiments show that the CPU reservation mechanism can lead to unused resources in the cluster, and the execution time of applications might have a high variance caused by degradation caused by other sources distinct than the CPU.

As a solution to deal with the competition of resources between containers, we propose a scheduling technique based on the characterization of applications. Clients or developers provide informal information about their applications – for instance, which resource the application uses more intensively – and in turn, the scheduler uses that information to allocate the applications using the same resource in different machines. In our experiments, we achieved about a 20 percent improvement in the execution time of a simple scenario compared with the default Kubernetes non-deterministic scheduler. The total execution time is about the half compared to a scenario were resources are reserved in Kubernetes. Additionally, the behaviour of our scheduler is deterministic, so it can be used for further analysis. As future work, for the classification stage of applications, machine learning algorithms can be exploited, which can even automate the classification process and can achieve more sophisticated classification results, while targeting more complex applications in order to improve our scheduling approach.

Acknowledgements. This work was co-financed by the Industry and Innovation department of the Aragonese Government and European Social Funds (COSMOS research group, ref. T93); and by the Spanish Ministry of Economy under the program “Programa de I+D+i Estatal de Investigación, Desarrollo e innovación Orientada a los Retos de la Sociedad”, project id TIN2013-40809-R. V. Medel was the recipient of a fellowship from the Spanish Ministry of Economy.

References

1. Awada, U., Barker, A.D.: Improving resource efficiency of container-instance clusters on clouds. In: 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2017). IEEE (2017)
2. Bhamare, D., Samaka, M., Erbad, A., Jain, R., Gupta, L., Chan, H.A.: Multi-objective scheduling of micro-services for optimal service function chains. In: IEEE International Conference on Communications (ICC 2017). IEEE (2017)
3. Bingmann, T., Axtmann, M., Jöbstl, E., Lamm, S., Nguyen, H.C., Noe, A., Schlag, S., Stumpp, M., Sturm, T., Sanders, P.: Thrill: high-performance algorithmic distributed batch data processing with c++. arXiv preprint arXiv:1608.05634 (2016)
4. Brunner, S., Blochlinger, M., Toffetti, G., Spillner, J., Bohnert, T.M.: Experimental evaluation of the cloud-native application design. In: IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC), pp. 488–493 (2015)
5. Burns, B., Grant, B., Oppenheimer, D., Brewer, E., Wilkes, J.: Borg, omega, and Kubernetes. ACM Queue **14**, 70–93 (2016)
6. Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., Tzoumas, K.: Apache Flink: stream and batch processing in a single engine. Bull. IEEE Comput. Soc. Techn. Comm. Data Eng. **38**(4), 28–38 (2015)
7. Choi, S., Myung, R., Choi, H., Chung, K., Gil, J., Yu, H.: Gpsf: general-purpose scheduling framework for container based on cloud environment. In: International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData), pp. 769–772. IEEE (2016)
8. Felter, W., Ferreira, A., Rajamony, R., Rubio, J.: An updated performance comparison of virtual machines and linux containers. In: International Symposium on Performance Analysis of Systems and Software (ISPASS), pp. 171–172 (2015)
9. Havet, A., Schiavoni, V., Felber, P., Colmant, M., Rouvoy, R., Fetzer, C.: Genpack: a generational scheduler for cloud data centers. In: 2017 IEEE International Conference on Cloud Engineering (IC2E), pp. 95–104. IEEE (2017)

10. Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A.D., Katz, R.H., Shenker, S., Stoica, I.: Mesos: a platform for fine-grained resource sharing in the data center. In: NSDI, vol. 11, p. 22 (2011)
11. Kaewkasi, C., Chuenmuneewong, K.: Improvement of container scheduling for Docker using ant colony optimization. In: 2017 9th International Conference on Knowledge and Smart Technology (KST), pp. 254–259. IEEE (2017)
12. Kumar, K.A., Konishetty, V.K., Voruganti, K., Rao, G.V.P.: CASH: context aware scheduler for hadoop. In: 2012 International Conference on Advances in Computing, Communications and Informatics, ICACCI 2012, 3–5 August 2012, Chennai, India, pp. 52–61 (2012)
13. Lukša, M.: Kubernetes in Action (MEAP). Manning Publications, Greenwich (2017)
14. McCalpin, J.D.: Memory bandwidth and machine balance in current high performance computers. In: IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter, pp. 19–25, December 1995
15. Medel, V., Rana, O., Arronategui, U., et al.: Modelling performance & resource management in Kubernetes. In: Proceedings of the 9th International Conference on Utility and Cloud Computing, pp. 257–262. ACM (2016)
16. Morabito, R., Kjällman, J., Komu, M.: Hypervisors vs. lightweight virtualization: a performance comparison. In: 2015 IEEE International Conference on Cloud Engineering (IC2E), pp. 386–393. IEEE (2015)
17. Oskooei, A.R., Down, D.G.: COSHH: a classification and optimization based scheduler for heterogeneous hadoop systems. Future Generation Comp. Syst. **36**, 1–15 (2014)
18. Page, L., Brin, S., Motwani, R., Winograd, T.: The pagerank citation ranking: bringing order to the web. Technical report, Stanford InfoLab (1999)
19. Raho, M., Spyridakis, A., Paolino, M., Raho, D.: Kvm, Xen and Docker: a performance analysis for ARM based NFV and cloud computing. In: 3rd Workshop on Advances in Information, Electronic and Electrical Engineering (AIEEE), pp. 1–8. IEEE (2015)
20. Seo, K.T., Hwang, H.S., Moon, I.Y., Kwon, O.Y., Kim, B.J.: Performance comparison analysis of linux container and virtual machine for building cloud. Adv. Sci. Technol. Lett. **66**(105–111), 2 (2014)
21. Verma, A., Pedrosa, L., Korupolu, M.R., Oppenheimer, D., Tune, E., Wilkes, J.: Large-scale cluster management at Google with Borg. In: Proceedings of the European Conference on Computer Systems (EuroSys), Bordeaux, France (2015)
22. Wang, K., Khan, M.M.H., Nguyen, N., Gokhale, S.S.: Modeling interference for apache spark jobs. In: 9th IEEE International Conference on Cloud Computing, CLOUD 2016, USA, pp. 423–431 (2016)
23. Zhang, W., Rajasekaran, S., Duan, S., Wood, T., Zhu, M.: Minimizing interference and maximizing progress for hadoop virtual machines. SIGMETRICS Perform. Eval. Rev. **42**(4), 62–71 (2015)