Your Containers Should be WYSIWYG

Mathieu Bacou, Alain Tchana, Daniel Hagimont

Your Containers should be WYSIWYG

Mathieu Bacou*‡, Alain Tchana†, Daniel Hagimont*
*IRIT, Université de Toulouse, CNRS, Toulouse, France — {first.last}@enseeiht.fr
†I3S, Université Nice Sophia Antipolis, CNRS, Nice, France — alain.tchana@univ-cotedazur.fr
‡Atos Integration, Toulouse, France

***Abstract*—Modern cloud platforms rely on containers in order to deploy applications and allocate resources to them. Users of Container-as-a-Service platforms interact with another layer of abstraction, container orchestrators, to set resource allocations. Regarding the CPU allocation, orchestrators can use one of two strategies to apply the specified allocation: (1) the allocation of cores, reserved for one application; or (2) the allocation of quotas, which can be provided by any of the available processors. However current orchestrators only use the quota strategy.**

We benchmark both, demonstrating that the quota strategy can show up to 68% of degradation in our experiments when compared to the first strategy. We identify that this degradation comes from violating what we call the *What You See Is What You Get* (WYSIWYG) principle: a container's view of its available resources is wrong under the quota strategy.

We state that a better trade-off can be found in combining these two strategies, and we design a hybrid resource allocation algorithm that can be integrated into any container orchestrator. Our evaluations show that it prevents resource management problems that come from allocating cores, while canceling the performance overhead associated with the quota allocation strategy that violates the WYSIWYG principle.

***Index Terms*—container, container orchestrator, performance, predictability, CPU allocation**

I. Introduction

Cloud services are now established as an important part of the expenses of many companies. Managing applications on virtual machines (VMs) as imposed by the IaaS (Infrastructure as a Service) [1] cloud model is a difficult task [2], [3]. The user is the one that has to provide fault tolerance, scalability on workload increase, etc.; even the deployment process can be arduous. Thus many companies adhere to the PaaS (Platform as a Service) model [4] where most of these tasks are provided by the platform. To this end, PaaS providers mainly rely on *containers* [5] (LXC [6], Docker [7]...) and *orchestrators* [8] (Swarm [9], Ansible [10], Kubernetes [11]...). Containers ease application packaging while orchestrators automate both the deployment and the reconfiguration (fault handling, scalability and more) for the entire application lifetime. Such a PaaS is often referred to as *Container as a Service* (CaaS). Examples of CaaS are Amazon Elastic Container Service [12], Google Kubernetes Engine [13] or Microsoft Azure Container Service [14].

Scope: performance predictability is the ability for an application to always reach the same performance level under the same workload. It has recently been highlighted as one of the main issues in the cloud [15], [16]. Our work is focused on performance predictability for CaaS. To enforce performance predictability, the user who deploys containers assigns to each of them a fixed amount of computation capacity. Computation capacity allocation to a container is then generally implemented by the orchestrator using two parameters: (1) *request*, which is the minimum capacity to guarantee; and (2) *limit*, which is the maximum capacity that the container can use. In the context of a predictable CaaS, *request equals limit*.

Problem: there are two ways to enforce a computation capacity to a container: CPU sets and CFS quota.[1] The CPU sets method restrains a container to a given set of CPU cores. Regarding the CFS quota method, containers share all the machine's cores and the OS scheduler ensures that the total CPU time used by each container is kept under its quota (seen as an amount of CPU time). The CFS quota method is easier to implement, because it relies solely on the OS scheduler instead of managing sets of cores as in the CPU sets method. This is why it is used by almost all orchestrators to enforce a computation capacity to a container. The issue with CFS quota is that it can *shatter the predictability guarantee* for some types of applications. Indeed in practice, many applications auto-configure themselves [17], [18] based on the perceived available resources: they scan the system for resources, and then determine their own settings. For instance, the number of spawned worker processes depends on the number of cores.

Therefore, when deployed in a container placed under an arbitrary computation resource limit with CFS quota, *the application's view of the available resources is wrong:* the WYSIWYG principle is broken.

What You See Is What You Get. *The WYSIWYG principle states that whenever a containerized application probes its environment, it obtains a faithful view of its available resources.*

For example, in a container limited to 200% of CPU usage — i.e. gets two cores worth of CPU time on each scheduling period — and deployed on a 56 cores server, an application will believe it may use 56 cores and will auto-configure itself with 56 worker processes. As we demonstrate in this paper, this misconfiguration results in both performance degradation and unpredictability. For such types of applications, we see that CPU sets is an appropriate method to enforce computation capacity.

The violation of the WYSIWYG principle also affects other resources, such as memory [19]. This paper focuses only on the effect on CPU and proposes a solution specifically crafted for this resource.

[1]Please refer to section II-B1 for a more thorough description.

DOI 10.1109/SCC.2019.00022

Contributions:

- an analysis and comparison of computation capacity enforcement using CFS quota and CPU sets, based on micro- and macro-benchmarks, at the performance and resource management levels;
- an algorithm for orchestrators to manage both methods and best use them; this algorithm is adapted to Kubernetes [11], a widely used orchestrator;
- a discussion on how to determine the appropriate method depending on the containerized application;
- an evaluation of our prototype.

Section II gives related background information. Section III presents the motivations and an assessment of the problem. Section IV describes the smart allocation algorithm and how it can be integrated with Kubernetes. Section V shows evaluation results. A review of the related work is given in section VI. Finally we draw our conclusion in section VII.

II. Background

In this section we give relevant background information on the technologies of containers and orchestrators, as well as their resource allocation systems.

A. Containers

Containers are the embodiment of Operating System-level virtualization. They are *isolated representations of the host OS,* at the level of different resources, and embed software to which they show a reduced view of the host's capabilities. For instance, a container can provide an application with a different filesystem hierarchy or a different process hierarchy, in such a way that the first process in a container gets the program identifier (PID) 1. It has access only to the network interfaces (NICs) that are also inserted into the container. Moreover, a process in a container may have lower memory limits, or PID limits, or CPU usage quota, etc. For instance, when limiting a process to a CPU set, the scheduler makes sure to only schedule the process on the allocated CPUs. Container isolation and limitation are implemented using *namespaces* and *control groups* respectively, which are both mainstream Linux kernel features. This paper studies container resource limitation, thus we focus on control groups.

In this picture, *container engines* such as Docker [7] manage processes along with their layers of isolation and their resource limits. They may add facilities to instantiate containers: for instance, building a special filesystem image to populate the isolated filesystem of a container, or managing virtual NICs for containers. Besides, *orchestrators* such as Kubernetes [11] are container managers: they provide another layer of abstraction to represent applications as an architecture of containers and manage their life-cycle.

B. Computation capacity limitation

This paper studies computation capacity limitation applied to containers in a CaaS. This is managed at two levels: (1) the container engine (2) and the orchestrator. For illustration, we consider Docker and Kubernetes, respectively.

1) Allocation at the container engine level: the container engine allocates CPU resources by two means: *CFS quota* or/and *CPU set*. Using the CFS quota mechanism, the container is assigned an amount of CPU time that the host OS scheduler — which for Linux is by default the Completely Fair Scheduler [20] (CFS) — allocates to the container's processes on every scheduling period. With this allocation mechanism, the container sees all cores present on the machine. As for the CPU set mechanism, it limits the container to a given set of cores, so it can only see those cores. It can also be used to pin a container to specific NUMA nodes or specific cores provided by Simultaneous multithreading (SMT).

As the reader can deduce, the CFS quota mechanism is more flexible than CPU set in the perspective of resource management. Indeed, it allows a fine-grained allocation in the sense that a container can request a portion of a core capacity. Moreover, CFS quota simplifies the work of top-level resource managers such as orchestrators because most of the work is done by the OS scheduler.

In addition, *CPU shares* can be used to set a minimal resource allocation *relatively to other containers*. However we do not use this feature (see below) because it is unpredictable — a container may use more than its share if there are free resources — and is not meant to allocate a definite amount of resources — it is relative to the shares of other containers.

2) Allocation at the orchestrator level: Kubernetes (as about any orchestrator) allows to statically specify the CPU needs of a container in terms of *requests* and *limits*. As argued in section I, these two parameters are equal in the context of a performance predictable CaaS, which is our research scope. Thus we ignore the *requests* part and its implementation that uses the mechanism of CPU shares described above.

To enforce a container's booked computation capacity, almost all orchestrators rely on the CFS quota mechanism because it is both easy to implement and perfect for efficient resource utilization. Of particular interest is a beta feature of Kubernetes to statically allocate CPUs to containers, that is to say allocate CPU resources using CPU sets rather than CFS quota. We describe in the following section how using CPU sets indeed addresses the performance and predictability issue; but we also argue why it cannot be used as-is because of its main inherent drawback of only allocating whole CPU cores.

III. Motivation and problem assessment

We show in this section that relying on CFS quota as current orchestrators do, is not the best way of providing performance predictability [15], [16] to all types of applications.

To this end, we compare the two mechanisms using benchmarks: Stream [21] and the PARSEC [22] benchmark suite. We also evaluate the in-memory analytics benchmark from CloudSuite [23], [24], a real-world Apache Spark [25] application that computes movie recommendations. The testbed is a 12 non-HyperThreaded, NUMA Intel® Xeon® E5–2420 v2 cores Dell machine running ArchLinux (Linux 4.15.15) and Docker 18.01-CE. However in order to avoid effects of the NUMA architecture, we set the benchmarks to only run on the

Figure 1: Distribution of Stream execution over 100 runs.

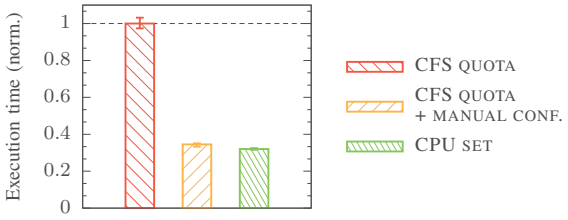


Figure 2: In-memory analytics benchmark execution time, normalized over its execution time under CFS QUOTA.

same NUMA node, i.e. on 6 cores. The computation capacity allocated to the container under test is 350%, meaning that the container is allowed to use the equivalent of the computational power of 3.5 cores. The container under test is executed under three separate setups:

- CFS QUOTA: the container has access to 6 cores, corresponding to the capacity of an entire NUMA node. However, CFS QUOTA is used to limit its CPU shares to 350%. This is the common practice;
- CFS QUOTA + MANUAL CONF: same as above, with the only difference that the application is manually configured to spawn a maximum of 4 threads (i.e. the closest integer number of threads greater than the allocation);
- CPU SET: CPU set is used to limit the container to 4 cores of the same NUMA node, and a CFS quota is also applied to actually limit the container to 350%. No manual configuration is applied.

Figure 1 shows the evaluation results for the Stream benchmark; each of its 100 individual runs is represented, to highlight the unpredictability issue. We can see that under CFS QUOTA, the benchmark exhibits two performance levels, illustrated by the hourglass-like shape of the points. This is not the case when using the CPU set mechanism, where points are mainly around the same performance level and are more clustered. This unpredictability issue with CFS QUOTA stems from the self-configuration feature of the evaluated benchmark when it decides the number of threads to spawn based on the number of cores (i.e. all the NUMA node's cores with CFS QUOTA). However, due to the use of CFS quota *that the application is not aware of*, its actual computing capacity is much lower. This mismatch, this violation of the WYSIWYG principle, is devastating for such hardware-dependent applications. This analysis is confirmed by executions under the CFS QUOTA + MANUAL CONF setup.

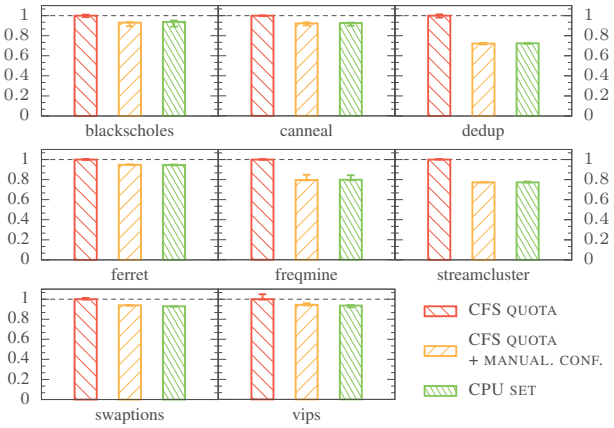


Figure 3: PARSEC benchmarks execution times, normalized over their respective execution time under CFS QUOTA.

Indeed we observed that this setup provides predictable results, as shown in the center plot of fig. 1. These results are similar to those obtained under the CPU SET setup, reported in the last plot, which suggests that CPU set is a suitable solution performance-wise. In addition to the unpredictability issue, we also observed that CFS quota lowers the performance of the tested applications in comparison with the two other setups: Stream performs 31.8% better under CPU SET. Further, results for CloudSuite's in-memory analytics benchmark are reported in fig. 2 (normalized median, 10% and 90% over 10 runs). The computation resource limit is set on the Spark worker container. For this example of a real application, the results are similar: the CPU SET setup outperforms the CFS QUOTA setup by 68%, and shows a better predictability.

Finally, results for the PARSEC benchmark suite are represented in fig. 3 (same data as for in-memory analytics). Performance improvements under the CPU SET setup range from 5.5% (FERRET) to 27.6% (DEDUP). Indeed, for a few benchmarks the violation of the WYSIWYG principle does not translate into a significant performance degradation.

Synthesis: a containerized application that performs auto-configuration is misguided by the difference between the real allocation, and the results of its scan of the system. This results in over-threading, which is characterized by thread management overhead, cache dirtying, etc. [26]–[30]. However, certain types of applications are agnostic about the allocation mechanism; some containers also embed background workloads like logging, for which performance degradation is not as big an issue. For such applications, it is better to use CFS quota in order to benefit from its efficient resource utilization capabilities. Indeed, CPU set has a coarse resource allocation granularity (a whole core) and is therefore inefficient for achieving fractional CPU limits. Furthermore, allocating sets of CPUs works against the scheduler because it is more constrained in making scheduling decisions: by definition, each container can only ever be scheduled on its CPU set.

Backed by our experiments, we advocate for the use of the

CPU set method when applicable. The next section presents a hybrid allocation solution which takes benefit of each allocation mechanism as much as possible.

IV. Hybrid CPU limit enforcement

In this section, we present our solution to the WYSIWYG compliance problem of the container engines and orchestrators.

A. Motivation

We identified that the problem comes from the fact that the container lies to its application in the number of cores it can use. Thus the straightforward solution is to fix this behavior. While this is very possible, it requires to modify the host kernel so when a process in a cgroup asks for the number of cores in any manner, the kernel responds with the true value. This is not a good solution in a cloud environment. Another solution is to incorporate in the containerized application, a library that can detect the true number of cores. This is undesirable, as the philosophy behind containerization for the deployment is to leave the application unchanged; inclusion of a library would require a special compilation path for a container version.

Finally, the experiments in section III show that a correct configuration is a solution. However, it requires a static modification of the application configuration embedded into the container. This static modification must be done for each variation of resource allocation, defeating the purpose of generic container images. Thus, setting a correct configuration to get the expected performance level is not a practical solution.

We designed an orchestrator-level container scheduler, because it only requires modification of the higher level of management (often a custom middleware developed by the cloud provider) and is completely application-agnostic, while fixing the problem for every affected application.

B. Resource allocation to containers

We propose a hybrid CPU limit enforcement strategy, relying on the two enforcement mechanisms: CPU set and CFS quota. From the analysis conducted in the previous section, we organize containers in two categories: $\mathcal{C}_1$ includes containers which are processor sensitive (i.e. that require adherence to the WYSIWYG principle) and $\mathcal{C}_2$ gathers other containers.

1) Overview: our algorithm is based on managing the cores of a server in three groups:

- $\mathcal{P}_1$: cores exclusively allocated to one $\mathcal{C}_1$ container, each of them is included in exactly one container's CPU set;
- $\mathcal{P}_2$: cores used to allocate a fraction of core to at least one $\mathcal{C}_1$ container, they are included in many CPU sets;
- $\mathcal{P}_3$: cores shared between all $\mathcal{C}_2$ containers, they are included in exactly one CPU set which is associated with all $\mathcal{C}_2$ containers.

$\mathcal{P}_3$ acts as a pool of available cores. The principle is to give a $\mathcal{C}_1$ container its own exclusive CPU set with the correct number of cores from $\mathcal{P}_3$ (thus moving them to $\mathcal{P}_1$), e.g. 4 cores for an allocation of 350%. If the allocation is not a whole number of cores, a core from $\mathcal{P}_2$ (or from $\mathcal{P}_3$ if needed, moving it to $\mathcal{P}_2$) is used to provide the remaining allocation; anyways, a CFS quota is used to schedule the usage of shared $\mathcal{P}_2$ cores. As for $\mathcal{C}_2$ containers, they all share the same CPU set made from $\mathcal{P}_2$ and $\mathcal{P}_3$ cores; again, their respective CFS quota arbitrates CPU usage. Therefore, $\mathcal{C}_1$ containers are allocated whole processors from $\mathcal{P}_1$ and may have fractional cores from $\mathcal{P}_2$; and $\mathcal{C}_2$ containers are allocated cores from $\mathcal{P}_2$ and $\mathcal{P}_3$.

2) Implementation: the implemented algorithm is described in alg. 1. It configures a container c with a CPU allocation r, expressed in millicores (mCPU) as is done with Kubernetes. Its goal is to allocate, as best as possible, a minimal set of cores to each $\mathcal{C}_1$ container while avoiding core sharing between $\mathcal{C}_1$ containers.

Algorithm 1 Hybrid CPU limit allocation.

Require: host has enough CPU resources in total to allocate r to c

procedure ALLOCATE(c, r)
  if host empty **then**
    $\mathcal{P}_1 \leftarrow \{\}$; $\mathcal{P}_2 \leftarrow \{\}$; $\mathcal{P}_3 \leftarrow$ {all cores}
  end if
  if c is type $\mathcal{C}_2$ **then**
    SET_CPUSET(c, $\mathcal{P}_2 \cup \mathcal{P}_3$)
  else
    $whole \leftarrow \text{floor}(r/1000)$; $frac \leftarrow r \bmod 1000$
    $p_1^c \leftarrow \{\min(whole, \text{size}(\mathcal{P}_3))$ cores from $\mathcal{P}_3\}$
    $\mathcal{P}_3 \leftarrow \mathcal{P}_3 \setminus p_1^c$; $\mathcal{P}_1 \leftarrow \mathcal{P}_1 \cup p_1^c$
    $p_2^c \leftarrow \{\}$
    if $\text{size}(p_1^c) \neq whole$ **then** ▷ missing whole cores
      if $frac = 0$ **then**
        abort allocation: cannot ensure WYSIWYG
      else ▷ can still ensure WYSIWYG (e.g. 150% on 2 cores)
        $rem \leftarrow (whole - \text{size}(p_1^c)) \times 1000 + frac$
        $p_2^c \leftarrow$ CHOOSE_PROCS($\mathcal{P}_2$, ceil($rem/1000$), rem)
        if $p_2^c = \{\}$ **then**
          abort allocation: cannot ensure WYSIWYG
        end if
      end if
    else if $frac \neq 0$ **then**
      $p_2^c \leftarrow$ CHOOSE_PROCS($\mathcal{P}_2$, 1, $frac$)
      if $p_2^c = \{\}$ **then**
        $p_2^c \leftarrow$ {1 core from $\mathcal{P}_3$}
        if $p_2^c = \{\}$ **then**
          abort allocation: cannot ensure WYSIWYG
        else
          $\mathcal{P}_3 \leftarrow \mathcal{P}_3 \setminus p_2^c$; $\mathcal{P}_2 \leftarrow \mathcal{P}_2 \cup p_2^c$
        end if
      end if
    end if
    SET_CPUSET(c, $p_1^c \cup p_2^c$)
  end if
  SET_QUOTA(c, r)
end procedure
function CHOOSE_PROCS(s, n, r)
  choose at most n CPUs from s to allocate r, returns {} if impossible
end function

3) Principle-hard and best-effort policies: this algorithm enforces that sensitive containers ($\mathcal{C}_1$) are WYSIWYG, i.e. only see a set of cores which corresponds to their allocated CPU resources, thus preventing misconfigurations. We call it "principle-hard". Another sensible "best-effort" policy is to accept allocating $\mathcal{C}_1$ containers even if they cannot be WYSIWYG, because the node actually has enough resources to host the container. Instead of aborting in multiple places, we would allocate the necessary amount of CPU on more cores from $\mathcal{P}_2$ by removing the limit n on CHOOSE_PROCS.

C. Container type determination

We propose to integrate container type identification in the Continuous Integration (CI) [31] step of performance tuning. This step usually involves determining the resource allocation needed by the application to perform at the expected performance level, under a characteristic workload. The application is evaluated under workloads w_i and with different resource allocations r_j. For each evaluation (w, r), we perform two sub-evaluations where the CPU allocation is enforced by either CFS quota or CPU set. The application is of type $\mathcal{C}_1$ (i.e. requires the WYSIWYG principle) if its performance under CPU set is better than under CFS quota in most evaluations. The evaluation in section III shows that the performance gap between both setups is significant when the application is sensitive, thus this method is accurate. Furthermore, it can be easily integrated in an existing CI process.

D. Integration in Kubernetes

Here we consider the Kubernetes resources allocator. In Kubernetes, containers are logically organized in pods. All the containers of a pod must be deployed on the same machine. Kubernetes chooses a machine with enough available resources to host all the pod's containers and instantiates each container c_i with the specified amount of CPU resources r_i on that machine. As Kubernetes exclusively relies on CFS quota, each container is constrained by its configured CPU quota, but it can see all cores on its host if it probes the system.

The first integration level in Kubernetes is to replace its node-level, quota-based allocation by the resource allocation strategy described in section IV-B. The second integration level is a modification of the Kubernetes orchestrator, more precisely the service which chooses the machine where a pod is deployed. The goal of the optimized orchestrator-level allocation algorithm is to deploy $\mathcal{C}_1$ containers on machines so that we maximize the number of processors allocated from $\mathcal{P}_1$, and minimize the shared processor time in $\mathcal{P}_2$ — while taking into account classic criteria such as resource availability. On deployment, Kubernetes computes the list of machines that have enough available resources to host the pod's containers. Then it simulates the execution of the hybrid allocation algorithm (shown in alg. 1) to find the machine where allocations from $\mathcal{P}_1$ are maximized, and shared CPU time from $\mathcal{P}_2$ is minimized. When the best machine is found, the simulated allocation can be reused to actually allocate the container.

E. Limits and improvements

The algorithm in its current state is a prototype, to show that a hybrid solution exists and can address the issue of this paper. It can nonetheless be improved.

The management of allocation fractions on $\mathcal{P}_2$ cores is a complex problem. It is a bin-packing problem, that is well known in cloud computing research because it has to be solved when managing virtual resources, in order to host as many VMs in a datacenter as possible [32]. However, the current consensus for VM placement is to *consolidate*, i.e. to use as few servers as possible to host VMs (and to maximize individual server resource usage). Reaching this goal provides the best power usage throughout the datacenter. This is not valid in our case: we want to *avoid colocation of fractional allocations*. The reason is that, as will be explained in section V, two $\mathcal{C}_1$ containers that share a core will see a performance improvement that is less than expected. Anyway, the current implementation in our algorithm uses a first-fit strategy, tweaked to prefer the emptiest $\mathcal{P}_2$ cores.

Moreover, an important feature that must be taken into account when talking about CPU allocation, is Non-Uniform Memory Access (NUMA). Essentially, it means that not all CPUs are equal for an application, and the placement of an application's threads and processes has a strong impact on its performance. Our algorithm does not currently integrate this constraint.

V. Evaluations

In order to manage each application with the appropriate resource allocation mechanism, our contribution is composed of two modules: (1) the container type identification system, and (2) the hybrid resource allocation system. The effectiveness of the former is obvious because it relies on benchmarking, as described in section IV-C. The set of experiments realized in section III prove that benchmarking is efficient. The experiments of section III also *prove the effectiveness of the CPU set allocation mechanism* in canceling the performance degradation induced by CFS quota. Therefore, this section focuses on the evaluation of the hybrid resource allocation algorithm described in section IV: we want to evaluate how effective it is in providing correct CPU sets to $\mathcal{C}_1$, as well as its overhead.

A. Experiment description

Remember that the goal of this system is to allocate, as best as possible, a minimal set of cores to each $\mathcal{C}_1$ container while avoiding core sharing between $\mathcal{C}_1$ containers. We also want to minimize resource waste, that could lead to rejecting containers for which the WYSIWYG principle could not be guaranteed. It follows that we are interested in two metrics:

- s is the proportion of total CPU time across the datacenter allocated to $\mathcal{C}_1$ containers and given on shared CPUs, i.e. cores in more than one CPU set;
- r is the reject rate of $\mathcal{C}_1$ containers because we could not guarantee the WYSIWYG principle (see section IV).

We evaluate both the principle-hard version, that rejects container allocations if it cannot guarantee the WYSIWYG principle, and the best-effort version that does not reject containers unless it simply cannot allocate enough resources.[2]

Note on s: it is an important metric because CPU sharing among $\mathcal{C}_1$ containers leads to performance degradation. For instance, in a setup where two containers request 350% CPU, each of them is assigned to a set of 4 CPUs under a quota of 350% (like in the CPU SET setup from section III), but the sets may have one common core equally shared between both

[2]That is to say, with the best-effort version $r = 0\%$.

containers. This resource sharing on one core leads to resource and scheduling interference, and thus performance degradation and unpredictability — which the user expects to avoid with a CPU set. We evaluated that the Stream benchmark presented in section III executes on average 1.7 times slower with a core shared with another instance of itself, than without any shared core. We do not expand on this interference because it is a well-known problem, independently of our use of CPU sets, especially for memory-intensive applications such as Stream [33], [34]. Nonetheless, *joined CPU sets are a better alternative than CFS quota* performance-wise: the benchmark remains on average 12% faster. To summarize, this metric is representative of the loss of resource flexibility inherent to allocating CPU set and restricting the scheduler, as explained in section III.

Moreover, our algorithm is two-fold (see section IV-D): (1) the hybrid allocation at the machine level; and (2) the optimized orchestrator scheduler that chooses a deployment machine by simulating the allocation to find the best one. Thus we also evaluate the impact of choosing the best deployment machine at the orchestrator level, on s and r. We further check the scalability of our algorithms with the proportion of $\mathcal{C}_1$ containers, i.e. the containers that need special handling.

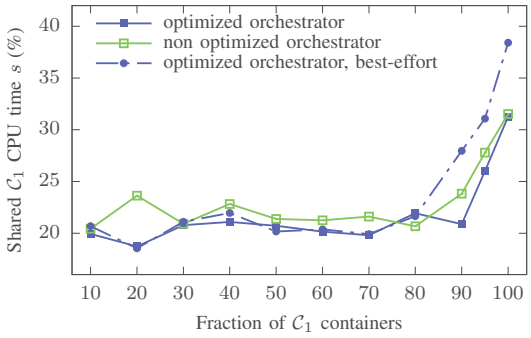
Our evaluation is a simulation[3] of allocations taken from the Google cluster traces [35] composed of 12.5k machines and lasting over about one month. Each container is given a certain chance to be $\mathcal{C}_1$ (i.e. to require a CPU set) or $\mathcal{C}_2$, that we varied through our experiments. We highlight the fact that Google's datacenter overcommits resources, which is not the case for Kubernetes, and neither is it for our allocation system. It means that the simulated datacenter is under a heavy load and cannot allocate all the containers from the trace. Understand that r only counts rejections due to a WYSIWYG principle violation, and ignores rejections due to a direct lack of resources.

B. Results

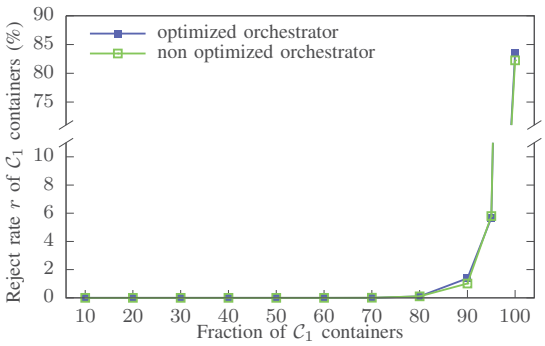
1) Optimized orchestrator-level allocation algorithm: fig. 4 shows a comparison of s and r for the principle-hard node-level allocation mode, with the optimized orchestrator-level allocation algorithm and without it (i.e. the default scheduler). s is stable for both cases between 10% and 90%: around 20.4% with the optimized algorithm, and 21.8% without. Indeed, the optimized algorithm has a beneficial impact on s, which averages at reducing shared CPU time by about 1.4 points (6.8%) for fractions of $\mathcal{C}_1$ containers between 10% and 90%. The behavior above 90% is commented in a paragraph below.

As for r, both optimized and non-optimized versions experience no container rejection due to impossible WYSIWYG allocation between 10% and 60% of $\mathcal{C}_1$ containers. Starting with 70%, the optimized broker increases the reject rate by about 0.3 points (1.7%). This behavior is expected because the optimized algorithm tries its best to avoid CPU sharing (i.e. tries to reduce s), which leads to greater CPU fragmenting;

[3]Source code of the simulator can be found here: https://git.bacou.me/?p=NestedVirt/KubernetesCPUSets.git.



(a) Shared $\mathcal{C}_1$ CPU time s (%). The lower the better.



(b) Reject rate r (%) of $\mathcal{C}_1$ containers for non WYSIWYG allocation.

Figure 4: Efficiency of the optimized orchestrator-level allocation algorithm under principle-hard and best-effort node-level allocation, depending on the fraction of allocated $\mathcal{C}_1$ containers.

with more fragmented CPUs, there are fewer whole cores available for CPU sets (see section IV for the hybrid node-level allocation algorithm). This effect is however very weak.

2) Best-effort node-level allocation algorithm: fig. 4a also shows a comparison of s for the best-effort and principle-hard node-level allocation modes (with the optimized orchestrator-level allocation algorithm). Remember that by nature, $r = 0\%$ for the best-effort mode (r for principle-hard mode is displayed in fig. 4b). Both allocation modes exhibit similar results between 10% and 80% of $\mathcal{C}_1$ containers. For greater fractions however, and in accordance with the increase of reject rate for the principle-hard mode, the best-effort mode allocates $\mathcal{C}_1$ containers with many shared CPU cores, leading to a 33.9% increase of s. Given that the principle-hard mode also shows null reject rate for $\mathcal{C}_1$ containers fraction under 60%, it is preferable to use the best-effort mode only when it is mandatory not to reject container allocations under heavy load.

With more than 90% $\mathcal{C}_1$ containers: values of s and r when allocating containers almost exclusively with CPU sets are very high with any setup ($s = 31.3\%$ and $r = 83.5\%$ with optimized orchestrator-level allocation and principle-hard node-level allocation). It shows why CPU set-only allocations is not a viable solution, and CFS quota must still be preferred when

the container is $\mathcal{C}_2$. If all containers require a CPU set, then the pool of cores available to $\mathcal{C}_1$ containers (see section IV) is quickly consumed, which means the system has no whole cores to allocate: it cannot guarantee the WYSIWYG principle, and the reject rate is very high. Similarly, s also rises because the only containers that can be allocated are forced to share at least one core. In summary, the absence of $\mathcal{C}_2$ containers leads to a starvation of whole cores on the nodes, which in turn leads to generally bad allocations, and a lot of rejected allocations. If non WYSIWYG allocations are allowed, as in best-effort mode (i.e. when not rejecting $\mathcal{C}_1$ containers because of the WYSIWYG principle), s shows an expected increase to 38.4%.

3) Scalability with the proportion of $\mathcal{C}_1$ containers: for a $\mathcal{C}_2$ container, our algorithms do nothing more than the classic allocator. However for a $\mathcal{C}_1$ container, i.e. that requires a CPU set, the best case for the node-level allocation algorithm is to find the correct number of cores from $\mathcal{P}_3$, which is an $O(1)$ operation. The worst case, where it has to find shared CPUs from $\mathcal{P}_2$, is directly linked to the number of $\mathcal{P}_2$ cores on a node, which can be bounded by c the number of cores on the node; thus the worst case is an $O(c)$ operation. So the node-level algorithm scales well with the number of containers on a node. However, a greater number of containers statistically means an increased scarcity in $\mathcal{P}_3$ cores, which favors occurrences of the worst case.

Results of scalability with the proportion of $\mathcal{C}_1$ containers for the optimized orchestrator-level allocation algorithm, are shown in fig. 5. We observe that the time needed to choose a node for one container increases with the proportion of $\mathcal{C}_1$ containers. It is because allocating a $\mathcal{C}_1$ container costs much more than a $\mathcal{C}_2$ container: as explained above, while the latter only requires normal orchestrator checks such as available resources, the former also needs a node-level allocation simulation. As the fraction of $\mathcal{C}_1$ increases, so does the average orchestrator-level allocation time per container.

Also note how there is a vast increase after the 90%-mark. It comes from the scarcity of $\mathcal{P}_3$ cores as explained above, that leads to the node-level allocation algorithm often hitting its worst case: almost every node-level allocation simulation becomes $O(c)$, with c the number of cores on a node.

Otherwise, the optimized orchestrator-level allocation algorithm scales with the number of nodes n because it simulates the node-level allocation on each node that can host the container. However this can be largely mitigated by offloading the simulation to each potential node: let each node simulate its own allocation and then communicate the result back to the orchestrator, that only has to compare the quality (WYSIWYG principle-wise) of each simulation. Many virtualized datacenter management frameworks [36] use a similar architecture by offloading node monitoring to the nodes themselves. Moreover, the node-level allocation algorithm for one container takes about $40\,\mu s$, which is negligible in the face of the creation time of a container that reaches hundreds of milliseconds [37]; so offloading will greatly reduce the node selection time upon container scheduling. Note that fig. 5 shows values from our prototype simulator that does not implement this offloading optimization.

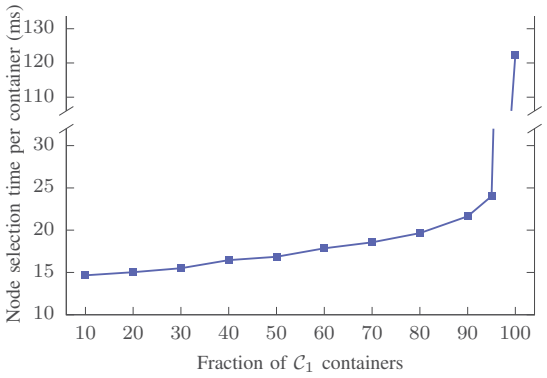


Figure 5: Scaling of the optimized orchestrator-level allocation algorithm with the fraction of allocated $\mathcal{C}_1$ containers.

4) Summary: Our hybrid allocation algorithm is able to fix the performance problem due to incorrect observed resources for about 79.6% of the simulated containers with minimal overhead on reject rate and container scheduling speed.

VI. Related work

Many papers have studied the performance benefits of containers when compared with virtual machines, showing that the former fare better than the latter [37]–[39]. In particular, Felter *et al.* [19] identified the lack of awareness of the resources limits as an fundamental limitation to containerization. Chaufournier *et al.* [37] also observed the difference of performance between the CFS quota and CPU sets, without further studying the phenomenon. Others compared both virtualization techniques from the point of view of the power consumption [40], showing no major difference. Finally, containers have been studied to provide high availability [41] or to actually replace VMs in PaaS [42].

There has been research on the resource management of containers. Paraiso *et al.* [43] modeled containers to ease their resource management. Hoenisch *et al.* [44] represented the joined problem of VM and container scaling as an optimization problem. Following this, Al-Dhuraibi *et al.* [45] proposed ElasticDocker to provide containers with vertical scaling in a fashion similar to VMs. Although it manages both the number of vCPUs (where the host is a VM) and the CFS quota, it does not investigate the effect of erroneous number of vCPUs. Both these works can be integrated with our hybrid resource allocator to help setting the resource limits. Baresi *et al.* [46] also developed a solution for vertical autoscaling of the containers and their VMs, and implemented a system of application-specific hooks to solve the problem of dynamically scaling the application with its container — thus addressing the problem of observed resource discrepancy. Our solution is application-agnostic, but handling dynamic scaling would require the ability to notify the application that the amount of allocated resources changed.

The problem of the discrepancy between a container's resources and the view of the containerized application has been noticed by Oracle, which proposed a solution under the form of a library [47] to abstract getting system resource information, and that aims at being container-aware. The drawback is that the application needs to be updated to use this library. Another remarkable initiative from the industry is the recent container-awareness of the Java Virtual Machine (JVM) [48]. Starting with Java 10, the JVM auto-configures itself based solely on memory and CPU configuration of its cgroup, which necessarily reports the correct amount from inside a container.

Finally, our hybrid CPU allocation mechanism is akin to vCPU pinning in the VM world. It has been shown [49] that vCPU pinning can be beneficial for power consumption, performance and in reducing resource interference. An advanced hybrid allocation mechanism could benefit from the works in this domain [50] to bring even better performance.

VII. Conclusion

Two mechanisms are available in Linux systems to enforce a computation capacity limit to a container: CPU set and CFS quota. Orchestrators such as Kubernetes generally rely on CFS quota for its ease of use and flexibility. However, the drawback of this approach is that all the machine's cores are made visible to each container independently of their allocated quota. This is particularly annoying for applications that autoconfigure according to the hardware environment they see (e.g. by instantiating a number of threads accordingly). We have shown that it may have a significant impact on performance and more importantly, predictability.

We advocate for an increased but rational use of CPU sets in addition to the CFS quota mechanism. To this end, we introduced a hybrid CPU limit enforcement strategy which relies on both mechanisms. The main principle is to detect for each application whether it requires the WYSIWYG principle, i.e. whether it is sensitive to the number of visible cores, and to schedule sensitive application containers on a reduced set of cores, thanks to the CPU set mechanism. We described how this allocation strategy could be integrated in the Kubernetes environment and evaluated it, showing that it can ensure optimal performance for the greatest fraction of CPU time while retaining some property of resource allocation flexibility.

Finally, the violation of the WYSIWYG principle affects more resources than the CPU, including memory and networking. Our work is focused on a solution tailored for the CPU but all resources will need to be accounted for.

References

[1] S. Bhardwaj, L. Jain, and S. Jain, "Cloud computing: A study of Infrastructure As A Service (IAAS)", *IJEIT*, vol. 2, pp. 60–63, Jan. 2010.

[2] L. M. Pham, A. Tchana, D. Donsez, N. de Palma, V. Zurczak, and P. Y. Gibello, "Roboconf: A hybrid cloud orchestrator to deploy complex applications", in *2015 IEEE CLOUD*, 2015.

[3] A. Tchana, B. Dillenseger, N. De Palma, X. Etchevers, J.-M. Vincent, N. Salmi, and A. Harbaoui, "A self-scalable and auto-regulated request injection benchmarking tool for automatic saturation detection", *IEEE TCC*, vol. 2, no. 3, pp. 279–291, 2014.

[4] *Google App Engine*, https : / / cloud . google . com / appengine/, Online, Google, 2018.

[5] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors", in *EuroSys*, 2007.

[6] M. Helsley, "LXC: Linux container tools", *IBM Dev. Works Technical Library*, 2009.

[7] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment", *Linux Journal*, vol. 2014, no. 239, p. 2, 2014.

[8] A. Tosatto, P. Ruiu, and A. Attanasio, "Container-based orchestration in cloud: State of the art and challenges", in *CISIS*, IEEE, 2015.

[9] *Docker swarm*, https://github.com/docker/swarm, Online, 2018.

[10] L. Hochstein and R. Moser, *Ansible: Up and Running: Automating Configuration Management and Deployment the Easy Way*. " O'Reilly Media, Inc.", 2017.

[11] E. A. Brewer, "Kubernetes and the path to cloud native", in *SoCC*, 2015.

[12] *Amazon Elastic Container Service (ECS)*, https://aws.amazon.com/ecs/, Online, Amazon, 2018.

[13] *Google Kubernetes Engine*, https://cloud.google.com/kubernetes-engine/, Online, Google, 2018.

[14] *Microsoft Azure Container Service (AKS)*, https://azure.microsoft.com/en-us/services/container-service/, Online, Microsoft, 2018.

[15] C. Delimitrou and C. Kozyrakis, "Hcloud: Resource-efficient provisioning in shared cloud systems", *ACM SIGOPS Operating Systems Review*, vol. 50, no. 2, pp. 473–488, 2016.

[16] V. Nitu, P. Olivier, A. Tchana, D. Chiba, A. Barbalace, D. Hagimont, and B. Ravindran, "Swift birth and quick death: Enabling fast parallel guest boot and destruction in the Xen hypervisor", in *ACM SIGPLAN/SIGOPS VEE*, ACM, 2017.

[17] T.-I. Salomie, G. Alonso, T. Roscoe, and K. Elphinstone, "Application level ballooning for efficient server consolidation", in *EuroSys*, ACM, 2013.

[18] K. R. Lawrence, *Method and system for dynamically adjustable and configurable garbage collector*, US Patent 6,629,113, 2003.

[19] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers", in *ISPASS*, 2015.

[20] M. T. Jones, "Inside the Linux 2.6 Completely Fair Scheduler", IBM, Tech. Rep., 2009.

[21] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers", *IEEE TCCA Newsletter*, 1995.

[22] C. Bienia, "Benchmarking modern multiprocessors", PhD thesis, Princeton University, 2011.

[23] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: A study of emerging scale-out workloads on modern hardware", in *ASPLOS*, 2012.

[24] T. Palit, Y. Shen, and M. Ferdman, "Demystifying cloud benchmarking", in *ISPASS*, Apr. 2016, pp. 122–132.

[25] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica, "Apache spark: A unified engine for big data processing", *Commun. ACM*, vol. 59, no. 11, pp. 56–65, Oct. 2016.

[26] S. Eyerman and L. Eeckhout, "The benefit of smt in the multi-core era: Flexibility towards degrees of thread-level parallelism", in *ASPLOS*, 2014.

[27] J. Kwon, K.-W. Kim, S. Paik, J. Lee, and C.-G. Lee, "Multicore scheduling of parallel real-time tasks with multiple parallelization options", in *RTAS*, IEEE, 2015, pp. 232–244.

[28] A. Raman, H. Kim, T. Oh, J. W. Lee, and D. I. August, "Parallelism orchestration using dope: The degree of parallelism executive", in *PLDI*, 2011.

[29] M. E. Haque, Y. h. Eom, Y. He, S. Elnikety, R. Bianchini, and K. S. McKinley, "Few-to-many: Incremental parallelism for reducing tail latency in interactive services", in *ASPLOS*, 2015.

[30] M. Jeon, Y. He, S. Elnikety, A. L. Cox, and S. Rixner, "Adaptive parallelism for web search", in *EuroSys*, ACM, 2013, pp. 155–168.

[31] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Pearson Education, 2010.

[32] A. Beloglazov and R. Buyya, "Optimal online deterministic algorithms and adaptive heuristics for energy and performance efficient dynamic consolidation of virtual machines in cloud data centers", *Concurrency and Computation: Practice and Experience*, vol. 24, no. 13, pp. 1397–1420, 2012.

[33] O. Mutlu and T. Moscibroda, "Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems", in *ACM SIGARCH Computer Architecture News*, IEEE Computer Society, vol. 36, 2008, pp. 63–74.

[34] ——, "Stall-time fair memory access scheduling for chip multiprocessors", in *MICRO*, IEEE Computer Society, 2007, pp. 146–160.

[35] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Heterogeneity and dynamicity of clouds at scale: Google trace analysis", in *ACM SoCC*, 2012.

[36] A. Beloglazov and R. Buyya, "Openstack neat: A framework for dynamic and energy-efficient consolidation of virtual machines in openstack clouds", *Concurrency and Computation: Practice and Experience*, vol. 27, no. 5, pp. 1310–1333, 2015.

[37] L. Chaufournier, P. Sharma, P. Shenoy, and Y. C. Tay, "Containers and virtual machines at scale: A comparative study", in *Middleware*, 2016.

[38] R. Morabito, J. Kjällman, and M. Komu, "Hypervisors vs. lightweight virtualization: A performance comparison", in *IC2E*, 2015.

[39] 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", *Advanced Science and Technology Letters*, vol. 66, no. 105-111, p. 2, 2014.

[40] R. Morabito, "Power consumption of virtualization technologies: An empirical investigation", in *UCC*, 2015.

[41] W. Li and A. Kanso, "Comparing containers versus virtual machines for achieving high availability", in *IC2E*, 2015.

[42] R. Dua, A. R. Raja, and D. Kakadia, "Virtualization vs containerization to support paas", in *IC2E*, 2014.

[43] F. Paraiso, S. Challita, Y. Al-Dhuraibi, and P. Merle, "Model-driven management of docker containers", in *CLOUD*, 2016.

[44] P. Hoenisch, I. Weber, S. Schulte, L. Zhu, and A. Fekete, "Four-fold auto-scaling on a contemporary deployment platform using docker containers", in *ICSOC*, 2015.

[45] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, and P. Merle, "Autonomic vertical elasticity of docker containers with elasticdocker", in *CLOUD*, 2017.

[46] L. Baresi, S. Guinea, A. Leva, and G. Quattrocchi, "A discrete-time feedback controller for containerized cloud applications", in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016.

[47] *Libresource*, https://github.com/lxc/libresource, Online, Oracle, 2019.

[48] E. Kahraman, "Docker awareness in java", Tech. Rep., 2018.

[49] 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*, 2015.

[50] Z. Li, Y. Bai, H. Zhang, and Y. Ma, "Affinity-aware dynamic pinning scheduling for virtual machines", in *CLOUDCOM*, 2010.