Mind the Gap: Broken Promises of CPU Reservations in Containerized Multi-tenant Clouds

Li Liu
George Mason University
Fairfax, VA, USA
lliu8@masonlive.gmu.edu

Haoliang Wang
Adobe Research
San Jose, CA, USA
hawang@adobe.com

An Wang
Case Western Reserve University
Cleveland, OH, USA
axw474@case.edu

Mengbai Xiao
Shandong University
Qingdao, Shandong, China
xiaomb@sdu.edu.cn

Yue Cheng
George Mason University
Fairfax, VA, USA
yuecheng@gmu.edu

Songqing Chen
George Mason University
Fairfax, VA, USA
sqchen@gmu.edu

ABSTRACT

Containerization is becoming increasingly popular, but unfortunately, containers often fail to deliver the anticipated performance with the allocated resources. In this paper, we first demonstrate the performance variance and degradation are significant (by up to 5×) in a multi-tenant environment where containers are co-located. We then investigate the root cause of such performance degradation. Contrary to the common belief that such degradation is caused by resource contention and interference, we find that there is a gap between the amount of CPU a container reserves and actually gets. The root cause lies in the design choices of today's Linux scheduling mechanism, which we call Forced Runqueue Sharing and Phantom CPU Time. In fact, there are fundamental conflicts between the need to reserve CPU resources and Completely Fair Scheduler's work-conserving nature, and this contradiction prevents a container from fully utilizing its requested CPU resources. As a proof-of-concept, we implement a new resource configuration mechanism atop the widely used Kubernetes and Linux to demonstrate its potential benefits and shed light on future scheduler redesign. Our proof-of-concept, compared to the existing scheduler, improves the performance of both batch and interactive containerized apps by up to 5.6× and 13.7×.

SoCC '21, November 1–4, 2021, Seattle, WA, USA
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8638-8/21/11...$15.00
https://doi.org/10.1145/3472883.3486997

CCS CONCEPTS

• **Social and professional topics → Pricing and resource allocation**; • **Software and its engineering → Scheduling**; *Virtual machines*; • **Computer systems organization** → *Cloud computing*.

KEYWORDS

Multi-tenancy, CPU scheduling, Resource management, Containers, Cloud

ACM Reference Format:
Li Liu, Haoliang Wang, An Wang, Mengbai Xiao, Yue Cheng, and Songqing Chen. 2021. Mind the Gap: Broken Promises of CPU Reservations in Containerized Multi-tenant Clouds. In *ACM Symposium on Cloud Computing (SoCC '21), November 1–4, 2021, Seattle, WA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3472883.3486997

1 INTRODUCTION

Containers are becoming increasingly popular for software development and operations (DevOps) because container tooling (e.g., the widely used Docker [51]) greatly simplifies the deployment and testing procedures. Containers can be managed by an orchestration system such as Kubernetes [2, 16, 24]. Developers specify the amount of resources for containers and deploy them through the orchestration system, which then places containers onto the host machines based on the specified resource configuration [10, 18].

Container resources are often configured empirically—typically based on developers' estimation of workload demands and expectations of performance. It has been observed that the performance of containers running in multi-tenant clouds varies significantly and is difficult to predict [8, 11–13, 15, 41–43]. For example, [13] reported that container co-location might lead to more than 66× higher tail latency. Such performance degradation can cause severe bottlenecks such as stragglers for batch workloads [38] and violation of end-to-end Quality-of-Service (QoS) guarantees for latency-sensitive, interactive applications [47].

It is well-known that performance variations and degradations often come from *noisy neighbors* in multi-tenant cloud environments. However, the specific sources and mechanisms that cause variations remain unclear. Context switching overhead and contention at the hardware level, such as cache, bus, I/O devices, are among the commonly believed causes. Nevertheless, here we argue that the primary source in a container environment lies in the OS and specifically its CPU scheduling mechanism, whose design goal fundamentally conflicts with that of a container environment.

Our investigation started with a series of experiments to reproduce the performance variations of containers running in a multi-tenant environment. We discovered that the container's actual CPU usage is significantly less than the requested, even though there was enough workload to saturate all its requested CPUs. Motivated by the observations, we further investigated Linux's CPU scheduling scheme and discovered that, despite promises to users, CPU resource reservations cannot always be fully honored in a multi-tenant cloud. Also, such an issue is persistent through the latest LTS Linux kernel (v5.10). Our investigation yields two root causes which we call *Forced Runqueue Sharing*, where threads from one container are forced to share the same CPU with others regardless its CPU reservation; and *Phantom CPU Time*, where containers with less threads cannot utilize seemingly available CPU time which is fragmented across multiple cores. Those two causes are the result of trying to keep CPUs busy, i.e., the work-conserving nature [7, 30, 49], regardless of the reservation from containers via cgroups, which directly contradicts a container's requirement and user expectations.

Although task scheduling and particularly Completely-Fair-Scheduling (CFS) have been studied extensively and a few related issues have been discovered [33, 40, 46, 49, 52], the need to keep CPU resource promises makes the problem unique in the context of containers. Many-core CPUs are common in modern data centers, where a commodity server can easily host many containerized applications. This exacerbates the problem of performance variation and degradation caused by neighbor interference. Furthermore, this issue leads to violations of the service level agreement (SLA) between users and cloud providers who charge users based on resources they *requested* rather than consumed [3].

We use these findings to further explore various options in Linux kernel scheduling. Due to the conflict of the work-conserving scheduler and the resource reservation requirement, a redesign of the scheduling mechanism is mandated. Given the complexity and the broad impact of a redesign, in this paper, we instead implement *rKube*, a proof-of-concept that augments the scheduling mechanism of Kubernetes [16] and Linux as a quick remediation. *rKube* utilizes cpuset which enables allocations of actual CPUs instead of CPU time shares to improve isolation of container workloads. The prototype, albeit simple, shows promising performance improvements that bridge the gap between the CPU resource promise and the resource and performance one can actually obtain, and at the same time, sheds light on future scheduler developments.

In summary, this paper makes the following contributions.

- We discovered significant performance variations in container-based multi-tenant environments with impacted CPU utilization, causing SLA violations.
- We investigated the root causes and revealed that Linux's CFS often does not comply with Linux cgroups' CPU resource promise. We explored various options in the current scheduler design and concluded that a redesign of CFS is mandated to fully solve this problem.
- Alternatively, as a proof-of-concept, we designed and implemented a prototype called *rKube*, demonstrating one potential isolation solution. Evaluation using both benchmarks running in containers and Alibaba data center traces shows that: compared to the Kubernetes, 1) *rKube* can deliver a 2.1×–5.6× speedup for batch tasks, and 1.2×–1.7× higher throughput and 12.9× and 13.7× lower tail latency for interactive tasks, and 2) *rKube* can reduce the tail latency of high-priority, interactive tasks by as much as 10×.

The rest of the paper is organized as follows. Sec. 2 presents the background on the container orchestration systems. Sec. 3 shows the experimental results demonstrating the significant performance degradation observed, which motivates us to investigate the underlying reasons in Sec. 4. We present our solution in Sec. 5, followed by its evaluation in Sec. 6. We discuss related work in Sec. 7 and conclude in Sec. 8.

2 BACKGROUND

Container orchestration systems. Modern container orchestration systems, such as Borg [55], Kubernetes (k8s) [24], and Docker Swarm [29], automate the deployment, management, and scaling of containers. When deploying a containerized application, users typically submit a spec file to the orchestration system, which takes care of the entire life cycle including application placement (on which hosts), resource allocation, and scaling. Take k8s as an example, the basic deployment unit of a k8s application is called a pod [23]. A pod can host one or multiple containers. A user can configure the resource requirements, such as CPU and memory, for the containers of a pod via k8s' specification file. K8s schedules and places the application's pod(s) on one (or multiple) hosts.

Container runtime. When a pod's container starts on a host, the CPU requests and CPU limits are passed to the container runtime (e.g., Docker) and will be enforced by the host

OS. The container runtime is responsible for local container management, including setting up the cgroups, namespace isolation, and starting the containerized processes. In a multi-tenant environment, multiple containers are co-located on the same host and share the same resources of the host.

Linux uses cgroups for container resource allocation and isolation. Cgroups limit the resource consumption of the containers and control how different containers share resources. Specifically, the CPU requests and limits for a container are enforced by the cgroup values of *cpu.shares*, *cpu.cfs_period_-us* and *cpu.cfs_quota_us*.

Container CPU Allocation. The user specifies the value of $c_i.requests.cpu$ for the container c_i. When Kubernetes deploys the container to a host machine, at least a requested amount ($c_i.requests.cpu$) of CPU should be reserved for the container c_i [18]. Upon receiving the CPU requests from the user, Kubernetes performs the following two tasks in order to fulfill the requested CPU resources:

- Kubernetes chooses one (assuming the application is deployed on a single host) host node n_k in the data center where there are enough CPU resources available. In other words, the following constraint needs to hold when the new container is deployed in the host:

$$\sum_{c_i \in C_k} c_i.requests.cpu \leq n_k.allocatable, \tag{1}$$

  where C_k is the set of containers in n_k, and $n_k.allocatable$ is the total amount of CPUs that is allocatable to containers in that host.
- Kubernetes translates the CPU requests $c_i.requests.cpu$ to a value that can be understood and enforced by the host OS. Cgroups uses an integer value $g_i.cpu.shares$ to indicate the amount of CPU resources that a group g_i can consume. Kubernetes then maps each container c_i to its associated control group g_i in n_k. Converting $c_i.requests.cpu$ to $cpu.shares$ goes as follows:

$$g_i.cpu.shares = c_i.requests.cpu \times 1024.$$

Note that the value of $cpu.shares$ used by cgroups is a relative weight for allocating CPUs [9], as opposed to an absolute amount of CPU cores. Such discrepancy may cause a host to break its resource promises if it becomes over-committed. Therefore, to keep its promise to the user application, Kubernetes imposes Eq. 1 to each host so that intuitively such a case would never happen [18, 25]. As will be shown in Section 3.4, despite such effort, a co-located container still cannot fully utilize all its requested (allocated) CPU resources.

3 MOTIVATION

This section demonstrates that significant performance degradation exists in a multi-tenant containerized environment. In a nutshell, our motivational study shows that: 1) the significant performance degradation is due to neighbor activities, 2) hardware contention may not be the main contributing factor of the degradation, and 3) a surprisingly low CPU allocation is a clear cause of the performance degradation.

3.1 Experiment setup and methodology

3.1.1 Environment. We use Dell PowerEdge R420s, equipped with 2 Intel Xeon E5-2420 CPUs (6C12T) and 24GB RAM, as the hosting machines. We experimented mainly in Debian 9.12 with Linux kernel v4.9 but also verified the problem persists in the latest LTS Linux kernel v5.10. Docker 18.09.7 is used as the container runtime at each host machine, and Kubernetes 1.17.3 is used as the container orchestration system. We reserve 2 CPUs for Kubernetes and OS system services on each host and leave 22 available CPUs for hosting containers. For simplicity, we configure one container per pod. Hence, we will use container and pod interchangeably hereafter.

3.1.2 Workload. Our study focuses on two major types of data center applications: batch and interactive [55].

Batch application. We use PARSEC (Princeton Application Repository for Shared-Memory Computers) benchmark suite [32] and SPLASH-2 (Stanford ParalleL Applications for SHared memory) [56] as our batch workload, including data analytics, computer vision, and scientific simulation. PARSEC contains implementations of several threading models including pthread, OpenMP and TBB with different behaviors e.g., dynamic task handling vs. static. Here, we choose the default one for each application. We use the application *completion time* and *CPU utilization* as the major evaluation metrics for batch applications.

Interactive application. We use `Memcached` [19] as an example of interactive applications. `Memcached` is a high performance, networked in-memory key-value (KV) cache system. We use YCSB (Yahoo Cloud Serving Benchmark) [37] to evaluate `Memcached`'s performance in terms of *throughput* and *latency*. We use an update-heavy YCSB workload with 50% reads and 50% updates for 1 million KV records. Read-heavy workloads show a similar trend and thus are omitted.

Neighboring application. `stress-ng` [45] is a tool to generate synthetic workloads that stress test various subsystems and kernel interfaces. We use it as the neighboring application to generate CPU-intensive or memory-intensive workloads (Table 1). Both run with 32 worker threads.

Burstable and capped neighbors. Kubernetes allows developers to configure an application's CPU resources using CPU *requests* and *limits*. Not setting a limit or setting a limit larger than the CPU requests enables *burst mode*, where a

Table 1: Summary of containerized target and neighbor applications. R denotes the value of CPU requests (and CPU limits) for the target applications. $22 - R$ cores are then allocated to neighbor applications as in total 22 CPU cores are available for all containers. "-" means the neighbor application's CPU limits is not set (i.e., burstable).

Target Application	Threads	CPU Requests (R)	CPU Limits	Description
bodytrack	6	6	6	Tracks a human body through space
fluidanimate	8	8	8	Physical simulation of a fluid
streamcluster	12	12	12	Online clustering of an input stream
ocean_cp	8	8	8	Computes the cholesky factorization of a sparse matrix
volrend	4	4	4	Computes the cholesky factorization of a sparse matrix
Memcached	4	4	4	High-performance, distributed memory object caching system
Neighboring Application				
Capped & CPU-Intensive	32	22-R	22-R	CPU-intensive stress-ng w/ CPU limits set to CPU requests (capped)
Burstable & CPU-Intensive	32	22-R	-	CPU-intensive stress-ng w/o CPU limits set (burstable)
Capped & Memory-Intensive	32	22-R	22-R	Memory-intensive stress-ng w/ CPU limits set to CPU requests (capped)
Burstable & Memory-Intensive	32	22-R	-	Memory-intensive stress-ng w/o CPU limits set (burstable)

container is allowed to use more CPU resources than its requested CPU resources [4] when idle CPUs become available. We call a container under burst mode a `burstable` container. Otherwise, when a container's CPU limits is the same as its CPU requests, We call it a `capped` container.

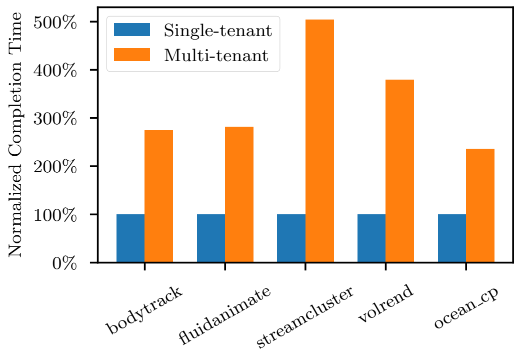
We use the publicly available Docker images [1, 5, 28] to run the applications mentioned above. Table 1 summarizes the batch and interactive applications that we use.

3.1.3 Methodology. A host that deploys multiple containers is called a *multi-tenant* host, where we have a *target* container and *neighbor* containers. We measure the performance metrics achieved by the target container's under interference from its neighboring containers.

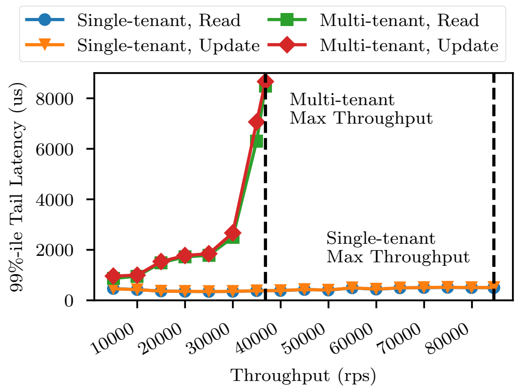
Table 1 specifies the CPU configurations of target containers. For neighbor containers, we set their CPU requests to be the remaining CPUs available in the host, i.e., $22 - CPU_{target}$. Depending on whether the neighbor is burstable or capped, its CPU limits will be set to default (empty) or the same requested value. We vary the number of threads for each application so that they take a similar amount of time (60s) to complete. For `Memcached`, we gradually increase the throughput (requests per second or rps) by 1000 until it is fully saturated; each `Memcached` test runs for 300s. Each experiment is repeated 10 times. The variance of the result among the repeated runs is small and thus omitted in the figures.

3.2 Impact of neighbors

We first quantify the impact of neighbor applications in a multi-tenant environment. Figure 1a plots the completion time of batch applications which is normalized against the completion time measured in a single-tenant environment. We observe that the performance degradation is severe under multi-tenancy— e.g., `bodytrack` and `streamcluster` take nearly 3× and 5× longer to complete in a multi-tenant host.



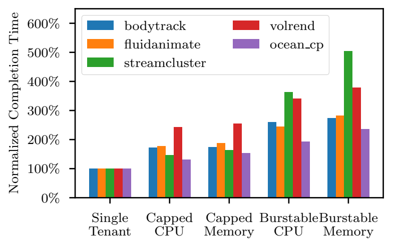
(a) Batch applications.



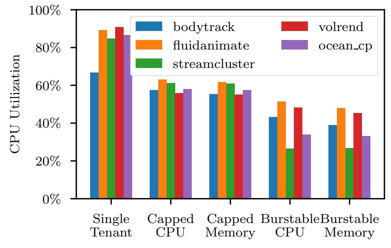
(b) Memcached.

Figure 1: Performance comparison: single-tenancy vs. multi-tenancy.

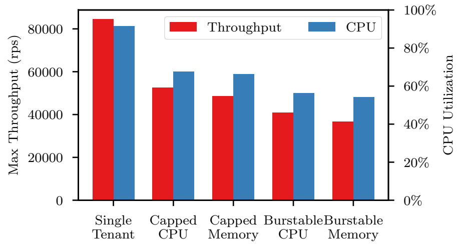
Similar performance degradation can also be observed for `Memcached`. As shown in Figure 1b, the 99^{th}-percentile (99%-ile) latency is much higher when `Memcached` is co-located with other apps, compared with running alone. Similar trend holds for the throughput metric: 1) the maximum throughput



(a) Batch applications, performance.



(b) Batch applications, CPU util.



(c) Memcached, performance & CPU util.

Figure 2: Performance and CPU utilization of target container when running alone vs. with neighboring apps.

drops from 85k (single-tenant) to 37k rps (multi-tenant), as depicted by the vertical dashed lines, and 2) under the same throughput (e.g., 37k rps), the 99%-ile tail latency increases from 383 to 8479 μs, an increase of more than 22×!

Implication. Severe performance variation makes it extremely difficult to predict application performance and accurately estimate application resource requirements.

3.3 Hardware contention is not the sole cause

One might wonder if hardware contention is the culprit. To verify that, we quantify the performance degradation of the target applications by varying the neighbor application behaviors. We experiment with four configurations of the neighboring stress-ng, namely, capped & CPU-intensive, capped & memory-intensive, burstable & CPU-intensive, and burstable & memory-intensive. We intentionally create contention on two shared hardware resources—CPU and memory. A burstable (with no CPU limits set) stress-ng poses pressure on CPUs, while a memory-intensive stress-ng poses pressure on the shared memory subsystem of the host.

As shown in Figure 2a and 2c, burstable & memory intensive neighbors cause the most significant performance degradation for target compared to the single-tenancy case. We also observe notable performance degradation for capped & CPU-intensive neighbors, where we have the least aggressive contention for hardware since the application is only CPU-intensive, and its CPU usage is being capped. It suggests that hardware contention might be the only or even the major cause of the significantly dropped performance. Further reasoning based on the observed CPU utilization will be provided in the next section.

Implication. The performance degradation is due to factors other than the hardware contention from the experiments.

3.4 CPU capping and low CPU utilization

Next, we examine the impact of CPU capping on the performance of target applications. One may expect that a capped container would not be able to "steal" any CPU resources from the target application. However, as shown in Figure 2a and 2c, this is not the case—by switching from a burstable neighbor to a capped one, interestingly, the performance degradation does still exist, albeit mildly mitigated. The evidence is, capped & CPU-intensive, though imposing a minimum level of hardware contention as described earlier, still sees up to 243% performance reduction.

To further investigate this counter-intuitive phenomenon, we measure the CPU utilization of target applications. The CPU utilization is calculated by the percentage of CPU time the application consumes over the experiment period:

$$CPU\_utilization = CPU\_consumption/requests.cpu$$

As shown in the Figure 2b and 2c, for both batch and interactive applications, the CPU utilization decreases as the neighbor's contention level and burstiness increase. This is consistent with the performance degradation of the target application shown in Figure 2a and 2c, indicating the resulted performance degradation is highly correlated with the insufficient CPU consumed by the target application. Note that such phenomenon should be distinguished with the performance degradation caused by context switch overhead and hardware contentions, in which case containers generally have high CPU utilization but do less real work.

What remains unclear is *why* the container is getting insufficient CPU, especially for the capped & CPU-intensive case, given that: 1) no over-commitment on the host - each container should theoretically get its requested CPU, 2) the container CPU usage is capped - no containers can use more than it requested, 3) minimal hardware contention, and 4) there is clearly enough workload left to be done as indicated by the single-tenant case.

Implication. The orchestration system, together with the underlying host OS, fails to satisfy the container's CPU requests, which should have been able to use all its CPUs.

4 ROOT CAUSE ANALYSIS

In this section, we further investigate how a CPU request is fulfilled in a multi-tenant containerized environment and present the two root causes of the insufficient CPUs problem showed in the previous section, namely, Forced Runqueue

Sharing (FRS), and Phantom CPU Time (PCT). These two are by no means an exhaustive analysis of all possible causes, but we believe they have a major impact on the observed CPU utilization issue and reflect a fundamental conflict between modern scheduling design and the need from containerized environment, which will be discussed in Section 4.3.

4.1 Forced Runqueue Sharing

CFS on a single-core system is reasonably straightforward. However, when it comes to a multi-core system, the scheduling decision making becomes a much more complicated optimization process. In a multi-core system, each physical CPU core has its individual *runqueue*. The processes will first be assigned to a runqueue and then be selected to run on that CPU. Ideally, when a user requests X CPUs for the container, all the processes spawned inside that container should be scheduled exclusively onto the runqueues of those X CPUs. However, due to CFS's load balancing activities, it may not be the case, and one container may be forced to share the runqueue with one or more neighboring containers, reducing its available CPU time and increasing the potential interference. We call this such scenario as *Forced Runqueue Sharing*, and we will show specifically how it happens.

Load balancing between runqueues is key to the overall system performance and CPU utilization. CFS periodically balances the processes of each runqueue based on multiple factors and metrics. One of these metrics is the *load* of a runqueue, which is derived from the weight (share) and nature of its tasks (i.e., processes and threads) [17]. The nature of a task can be derived from its CPU usage history using a load tracking scheme called per-entity load tracking (PELT) [22]. For example, a lower-weight CPU-bound task may contribute more load to the runqueue than a higher-weight, I/O-bound task due to different historical CPU usage. Hence, the processes in a container with a large amount of requested CPUs (hence a high CPU share) are not guaranteed to occupy a runqueue exclusively. To make things even worse, there are multiple other factors, including cache locality, the Non-Uniform Memory Access (NUMA) topology, CPU affinity, CPU bandwidth control, and CPU capacity in heterogeneous platforms, which are not related to CPU shares but affects how processes are allocated to the runqueues.

Once the processes are assigned to their runqueues, for each CPU at every scheduling tick, the scheduler will pick the next process in that runqueue to run on the associated CPU core. In a consolidated multi-tenant environment, it is likely that the processes of one container share the same runqueue with other containers. During a "scheduling period" all processes in the runqueue would have a chance to run; the amount of CPU time is proportionally divided into "time slices" for all the processes in the runqueue based on their weights. For each process, the time slice is a time interval it expects to run on a CPU core or its CPU timeshare during this scheduling period. So a task weight only indicates a *relative* CPU share, not an absolute CPU share that the users expect their containers to get based on the reservations. Since the weight is only relative, once a container shares the runqueue with another container, its promised amount resources can no longer be guaranteed, and hence the previously observed insufficient CPU usage in Section 3.4

Table 2: Containers' *requests.cpu* and their corresponding share and weight.

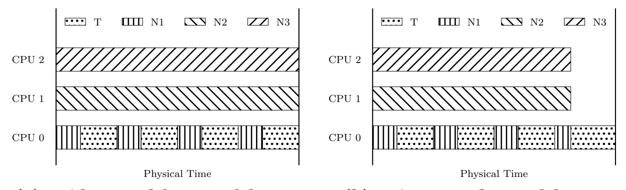
Container	*requests.cpu*	*cpu.shares*	Task (thread)	Task weight
Target (T)	1	1024	T	1024
Neighbor (N)	2	2048	N1, N2, N3	683, 683, 683

4.2 Phantom CPU Time

Currently, in Linux scheduler design, there is another design consideration to keep a "minimum granularity" for each time slice, which determines the minimum amount of time that a process needs to run before it can be preempted. It is set to prevent the slices from becoming too short, calling the scheduler too frequently, and increasing the overhead. The minimum granularity is determined by several scheduler parameters, including *sched_min_granularity_ns*, *sched_latency_ns* and *sched_wakeup_granularity_ns*, which is to balance the interactivity and the overhead. However, the interaction between them and the fact that processes from multiple containers share the same runqueue can lead to a phenomenon we call *Phantom CPU Time* - the available CPU time that containers are seemingly able to utilize but actually cannot, which further contributes to the previously observed performance degradation.

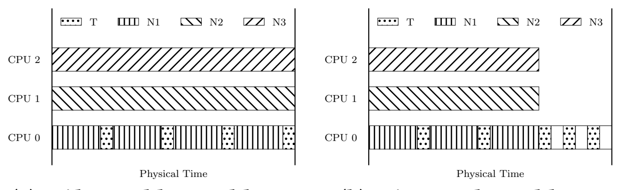
We now examine the activities of each CPU during one scheduling period with either batch or interactive applications. Assuming we have two containers and their configurations are listed in Table 2. Here two containers T and N are running on a host with three CPUs. Container T has only one thread, while container N has three threads: $N1 - N3$. The requested CPUs for T and N are 1 and 2, respectively. Because of the nature of the applications, load balancing and other reasons mentioned in the previous section, T and $N1$ are assigned to the same runqueue of CPU0. $N2$ and $N3$ are assigned to CPU1 and CPU2, respectively.

Let us first look at the case when T is a batch application. When the neighbors are burstable, as shown in Figure 3a, CPU1 and CPU2 are fully consumed by the neighboring threads $N2$ and $N3$. The container T only gets 60% of CPU0, as its CPU weight from its cgroup is 1024, and the CPU weight of $N1$ is 683. This is only 60% of its requested CPUs.



(a) w/ burstable neighbors. (b) w/ capped neighbors.

Figure 3: Illustration of a scheduling period where the target container T is running batch applications.



(a) w/ burstable neighbors. (b) w/ capped neighbors.

Figure 4: Illustration of a scheduling period where the target container T is running interactive applications.

Even if we cap the neighbors, as shown in Figure 3b, the situation does not get better for T. When $N1-N3$ consume all its CPU quota (i.e., 2), all these three threads will be suspended during the current scheduling period, and CPU0-CPU2 will become available. If T were to utilize all the remaining time of CPU1 and CPU2, it would be able to consume the time of one full CPU it has requested. However, since T only has one thread, it can only run on one CPU at a time and, therefore, cannot take advantage of those *phantom* available CPUs. Hence, no matter whether the neighbors are capped or burstable, the target container will always receive significantly lower actual CPU time than its requests amount, despite no over-commitment in the system. Due to the load balancing and the way the notion of CPU shares is defined (without considerations of parallel consumption), the more threads one container has, the more disruptive it will be when mixed with other containers with fewer threads, and the more CPUs it can unfairly utilize in case of burstable.

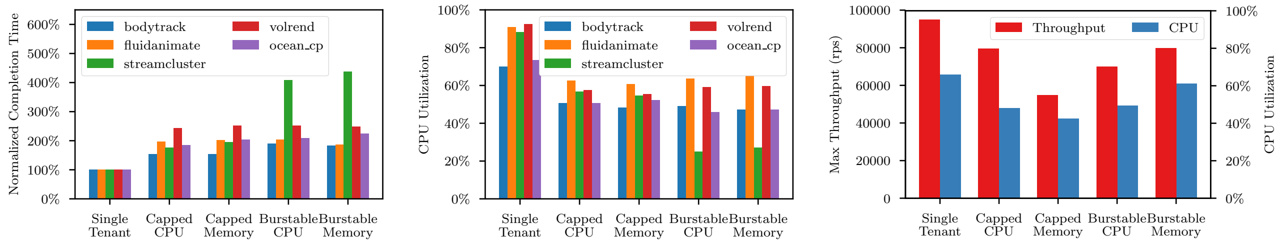
Now let us examine the case when T is an interactive application. Unlike a batch application, an interactive application may wake up and only do a relatively small amount of work before going back to sleep. For example, Memcached receives a request, processes it, replies to the client, and then goes back to sleep. However, it needs to react quickly, i.e., wake up as soon as a user request arrives. When the neighbors are burstable, as shown in Figure 4a, T cannot wake up in time because of the enforcement of the minimum granularity. Theoretically, T can be scheduled to run on CPU1 or CPU2. However, frequent migrations between cores are often discouraged due to cache affinity and other factors, and hence rarely happen. T therefore has to continue sharing CPU0 with $N1$, resulting in an increased requests processing latency due to limits on frequent preemption and context switches. Similar to the batch application scenario, capping the neighbors does not help. As shown in Figure 4b, only when the neighboring application's threads exhaust all their CPU quota, T can finally wake up more frequently than before. Nevertheless, T still cannot take advantage of the phantom CPU time on CPU1 and CPU2, simply because T has only a single thread.

4.3 Our key finding

As we have demonstrated, due to FRS and PCT, target containers are unable to fully utilize all reserved CPU resources.. One may wonder if these two are scheduler bugs that are overlooked and can be easily fixed. However, we argue that neither FRS nor PCT is simply a bug. Rather, they are the result of interactions between various design considerations in modern schedulers. When it comes to a system running containerized workload, the resource reservation requirement from containers becomes incompatible with those design choices made in modern schedulers.

Modern schedulers (e.g., Linux's CFS, FreeBSD's ULE and Xen's Credit scheduler) have been designed with at least two goals in mind: to maximize resource utilization and to maximize interactive performance. To support the interactivity, jobs run periodically with time slices assigned by the scheduler. Time slices are often lower-bounded, otherwise, the resulting excessive number of context switches will hurt utilization and performance. Given a lower-bounded time slice, the scheduler may not effectively preempt a running job any time a higher priority job needs to run. This directly contradicts the common expectation of resource reservation (e.g., CPU bandwidth control of cgroups [7]) in a containerized environment: an ideal containerization design would expect a container, which has a certain amount of CPUs reserved but has not fully utilized the reserved resource yet, to be able to run in order to fulfill the reservation before the reservation expires. The two causes we found are therefore inevitable results of such conflicts in design goals.

Specifically, despite that the target container can reserve the amount of a whole CPU so that its threads can run anytime, whenever such a thread yields or blocks, OS will move other threads to the reserved CPU to keep it busy. Thus, FRS is the result of load balancing mechanisms of the schedulers, which is a result of the goal to maximize utilization. Similarly, PCT is a result of the combination of two design considerations: 1) threads being unable to unconditionally preempt others, which in turn is the result of having a lower-bounded time slice in order to reduce scheduling overhead; and 2) threads being the scheduling entity, which effectively biases



(a) Batch applications, performance. **(b) Batch applications, CPU util.** **(c) Memcached, performance & CPU util.**

Figure 5: Performance and CPU utilization of target container with Linux kernel v5.10 running on CloudLab.

towards containers with more threads, and in turn is the result of enforcing system-wide fairness across threads.

To summarize, we argue that there is a fundamental mismatch between the design goals of modern schedulers and the need to reserve CPU for the containerized environment. Such a mismatch strongly demands a new mechanism to fulfill the promise. Hoping the problem may have been mitigated by the recent Linux kernel development, we tested the latest LTS kernel v5.10 with a similar configuration as in Sec. 3. As shown in Figure 5, the same problem persists for batch and interactive applications, albeit with slightly different behaviors from different neighboring workloads. This leads to our further investigation on the design of a new resource configuration scheme.

5 *rKube* AS A MAKESHIFT SOLUTION

Here we discuss the challenges in fundamentally solving the problem and then propose a proof-of-concept design called *rKube*, as a quick yet practical rescue. *rKube* demonstrates what can be achieved, shedding light on future designs.

5.1 Rationale of *rKube*

Given the fundamental conflicts discussed in Section 4, solving the problem within the existing scheduler is challenging without a redesign. For example, an intuitive way to achieve CPU reservation is to allow the target container to regain the CPU as soon as it is ready (e.g., after I/O completion). This requires unconditional preemption, so the target container will not wait in the queue (and compete with others). By constantly assigning a higher priority to the target container, this might be (approximately) achievable. However, the consequence of such an approach is that 1) there are a lot (more) context switches than without such a change. With an excessive number of context switches, the overhead increases, the effective CPU utilization inevitably decreases; 2) while the target's CPU utilization is improved, other concurrent processes suffer because the more time the OS scheduler uses to serve the target, the less remaining time it has to serve the other concurrent processes [7]. This is even without counting the increased overhead of context switches.

Based on our investigation and experiments, we concluded that changing the scheduler's existing configuration parameters could not effectively mitigate the issue. Even worse is that other unexpected conflicts may be introduced when fiddling with the parameters. Hence, we argue that a redesign of modern schedulers is mandated to solve the problem. However, a redesign of a fundamental component such as CFS is likely to take years to achieve. As a quick and CFS transparent solution, instead, we propose *rKube*, which augments the Kubernetes by introducing the currently missing parameters that allow users to express their demand to the kubelet. It relies on CPU reservation (instead of dynamic weight adjustment) to implement resource allocation and scheduling.

Specially, CPU reservations could be achieved by setting CPU affinity for individual containers. For this purpose, Linux provides a control feature, called cgroups, that could be utilized by *rKube* via setting *cpuset.cpus*. While *cpu.shares* provides relative CPU shares among groups, *cpuset.cpus* limits CPU usages in absolute values that are independent of the CPU speed and the scheduling of neighboring containers. In our design, *rKube* first selects a set of CPUs to be reserved for the target container based on their *requests.cpu*; then, it sets *cpuset.cpus* for all the containers to guarantee that the runqueues of the selected CPUs are exclusive for the target container. In this way, no other neighboring containers will be scheduled on these CPUs. Eventually, the system and the other containers will not be affected by the scheduling overhead of the reserved CPUs, the cost of which will be paid by the users who are in demand.

5.2 Implementation of *rKube*

We built our *rKube* prototype based on Kubernetes v1.17.3. To enable a user to require specific scheduling for a container explicitly, we add a new field named "policy" to the Kubernetes pod template. It refers to if the CPU request is `strict` or `standard`. Based on this field's value, a container can have its *requests.cpu* fulfilled by *cpuset.cpus* instead of the original *cpu.shares* if the request is `strict`, i.e., using *rKube*. Otherwise, it falls back to the original implementation. Thus, this option provides backward compatibility.

To implement *rKube*, we modify Kubernetes components `kubectl, apiserver, kube-scheduler` and `kubelet` to make sure the new field is correctly passed to `kubelet` in the host. In the host, once a container has strict CPU requests, the configured number of CPUs is selected. As we observed in Figure 2, contention on the hardware resources also impacts the target application. To reduce the interference with its neighboring applications in the shared memory hierarchy, the CPU topology is taken into consideration in the CPU selection. *rKube* aims to separate them from shared CPUs used by other containers. For example, when *requests.cpu* is larger than the CPU number in a socket, and all CPUs in that socket are not reserved yet, all these CPUs will be selected. They will be added to *cpuset.cpus* of the target and will be removed from *cpuset.cpus* of all others in the host.

There are other design choices in building *rKube*. Originally, in the host, `kube-reserved` and `system-reserved` can be used to requests CPU for Kubernetes and OS system daemons, such as the kubelet, container runtime, and sshd [25]. Similar to user containers, *rKube* applies the same CPU reservation for the requests of system services to prevent them from being starved and becoming bottlenecks.

Furthermore, since containers may fail unexpectedly, like other Kubernetes components, our subsystem that manages cgroup should also be resilient to container crashes. For this purpose, *rKube* does not rely on Docker to update `cgroup` values, but instead, it maintains host status and writes to cgroup filesystem directly. The subsystem also checks for any inconsistency on all pod life cycle events. Hooking pod life cycle events rather than container life cycle events makes the CPU reservation simple, stable, and complete.

Note that our implementation of *rKube* did not change the underlying host's default scheduling, e.g., CFS. We chose not to directly modify CFS to make it fit for containerized environments because 1) we strive to make the implementation transparent to the underlying OS, and 2) CFS is widely used for other purposes, and modification of that would impede the adoption of *rKube*. For similar considerations, *rKube* keeps the default approach for requesting resources, but with the new option added, it is backward compatible.

As a makeshift solution, *rKube* also comes with several limitations. For example, at the moment, it does not support fractional CPU shares, and by allocating CPUs exclusively, it could impact the overall host resource utilization.

6 EVALUATION

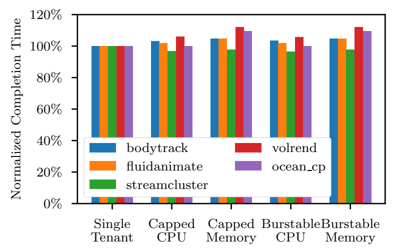
The performance of *rKube* is evaluated from two perspectives: 1) its effectiveness in reducing the neighbor interference and keeping the CPU resource promise, and 2) how does a good resource promise translate into the direct benefits for developers and cloud service providers, i.e., its practical value. For effectiveness, *rKube* is compared with the default scheduling strategy, referred to as *standard* in the corresponding figures. We evaluate with both synthetic workload (Section 6.1) and realistic workload adapted from real-world data center traces (Section 6.5). For the practical value, we study three common use cases in cloud performance tuning and evaluate the benefits of using *rKube* against the typical best practices in production [14], including "resource scaling" (Section 6.2 and 6.3) and "resource under-commitment" (Section 6.4).

6.1 Effectiveness of *rKube*

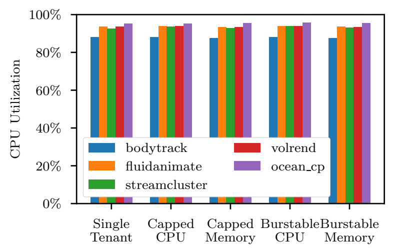
First, we study whether the containerized applications' performance can be improved with *rKube* and how much it can help, if any. For this purpose, we compare the CPU consumption and performance of different containerized applications in the multi-tenant environment where different neighboring containers are running with *standard* and *rKube*. The evaluation is conducted using the same setup as in Section 3.1.

Figure 6 shows the result under *rKube*, while the result under *standard* is shown in Figure 2 in Section 3. The results show that *rKube* effectively increases the CPU utilization and performance of the target applications, regardless of what type of settings a neighboring container uses. For example, for `streamcluster`, when its neighbor runs a memory-intensive workload and does not have *limits.cpu* set, as shown in Figure 2b and Figure 6b, *rKube* increases its CPU utilization from 27% to 93%; correspondingly, as shown in Figure 2a and Figure 6a, its completion time is reduced from 482.0 seconds to 86.4 seconds, a speedup by more than 5x. For the batch applications, the speedup of task completion time ranges from 2.1x to 5.6x depending on the different neighboring workloads. For `Memcached`, as shown in Figure 2c and Figure 6c, *rKube* increases its CPU utilization from 55% to 95%, while its maximum serving throughput increases from 37k to 62k rps. Figure 8 further shows that the corresponding 99%-ile tail latency for read operations is reduced by over 13x, from 8479μs to 621μs. We observe that the improvement of `Memcached` happens across all the situations, with a throughput boost from 1.2x to 1.7x and a reduction of tail latency between 12.9x and 13.7x.

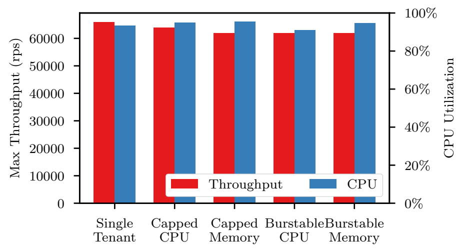
As a makeshift solution, *rKube* slightly scarifies overall host CPU utilization in certain cases. For example, as shown in Figure 7b, for the burstable & memory-intensive case, *rKube* decreases host CPU utilization from 100% to 91%, since user (neighbor) containers are prevented from using CPU reserved for system services. In this figure, the CPU below the lower dashed line is reserved for the target container, the CPU between two dashed lines is reserved for the neighbor container, and the CPU above the upper dashed line is reserved for other system services. However, for the capped & memory-intensive case, *rKube* instead increases host CPU utilization from 88% to 92%.



(a) Batch apps, performance.

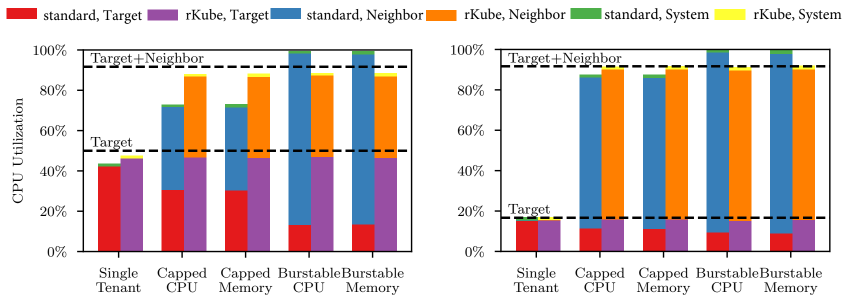


(b) Batch apps, CPU utilization.



(c) Memcached, performance & CPU util.

Figure 6: Performance and CPU utilization of target container with *rKube*. (*standard* result in Figure 2)



(a) Batch app (streamcluster).

(b) Memcached.

Figure 7: Composition of overall host CPU utilization with *rKube* and *standard*. Dotted lines indicate the CPU allocated to target/neighbors.

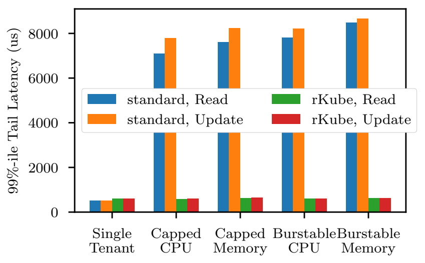


Figure 8: rKube vs. standard, Memcached tail latency.

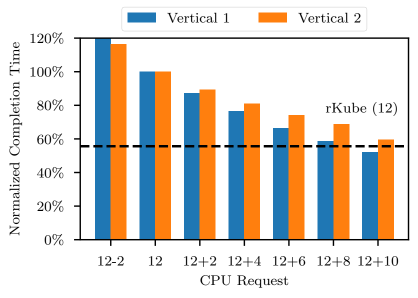
6.2 Vertical scaling vs. *rKube*

Our evaluation so far shows that, with correctly enforced CPU requests, *rKube* can effectively improve the application's CPU utilization and performance in the multi-tenant containerized environment. In current practices, when a user deploys an application to the cloud, and the application's performance is not satisfying, the user often has to invest more resources to improve its performance, often referred as vertical scaling. As most cloud providers charge users based on the resources requested , an accurate estimate of resource demand for the application is not only critical for predicting its performance (e.g., when the task can complete), but can also help reduce the user's cost. We next study how *rKube* can effectively avoid unnecessary resource wastage with improved cost-effectiveness.

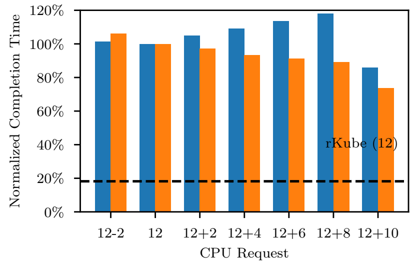
Figure 9a and Figure 9b show the effect of vertically scaling `streamcluster` vs. when *rKube* is applied alone, respectively. Two strategies are available when performing vertical scaling: 1) the number of threads equals to the CPU requests, thus increasing along with the CPU requests; and 2) the number of threads remains fixed while the CPU requests increase. We denote these two strategies as `Vertical 1` and `Vertical 2` in the figure. By default, the number of threads and the CPU requests are both set to 12 for `streamcluster`. We then vary the CPU requests between 10 and 22, as shown by the figure's x-axis. In this case, strategy "Vertical 2" keeps the number of threads 12. Figure 9a and Figure 9b show the normalized completion time, which is calculated as the ratio of the completion time of various scaling strategies to the baseline. Thus, the smaller the ratio is, the less the completion time it takes.

The results show that by requesting more CPUs, the application performance for `streamcluster` improves slowly with both scaling strategies. As a comparison, *rKube* (dashed lines) requires no additional CPU resources but reduces the application's completion time by 40%, as shown in Figure 9a (when the neighbor application is capped and memory-intensive), and by 80%, as shown in Figure 9b (when the neighbor application is burstable and memory-intensive). Furthermore, vertical scaling causes further performance drop with a burstable neighboring application since the CPU contention increases as more threads are spawned by vertical scaling. This trend can be observed from the results of "Vertical 1" in Figure 9b.

The results indicate that *rKube* is more effective than vertical scaling when guaranteeing a predictable performance. Solely increasing resources does not prevent the target applications and neighbor threads from sharing the same CPU runqueue, thus "stealing" CPU resources from the target.

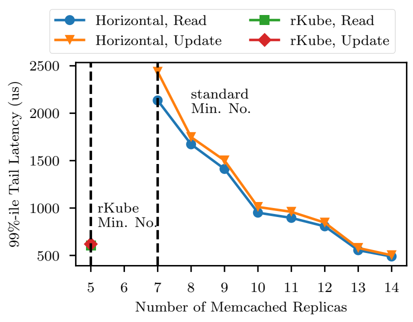


(a) Capped & Mem-Intensive neighbors.

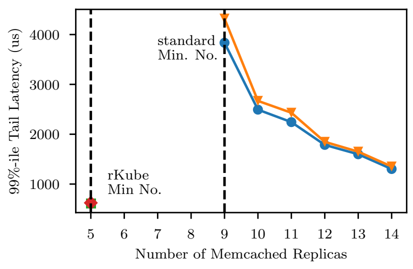


(b) Burstable & Mem-Intensive neighbors.

Figure 9: `streamcluster` performance improvement when using vertical scaling and *rKube*.

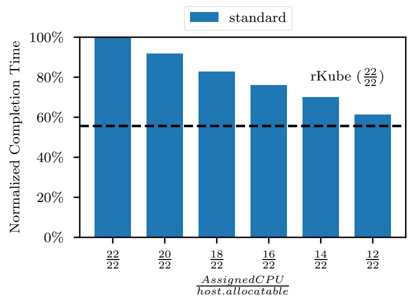


(a) Capped & Mem-Intensive neighbors.

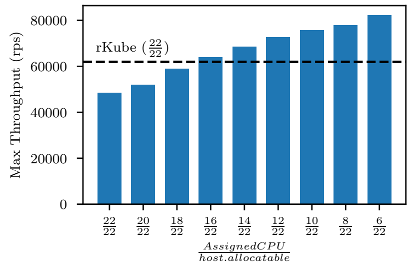


(b) Burstable & Mem-Intensive neighbors.

Figure 10: `Memcached` performance improvement when using horizontal scaling and *rKube*.



(a) Batch application (`streamcluster`).



(b) `Memcached`.

Figure 11: Application Performance by under-commitment strategy and *rKube*.

6.3 Horizontal scaling vs. *rKube*

For interactive applications, "horizontal scaling" or "scaling out" is often used when the applications hit a performance bottleneck. Unlike vertical scaling, horizontal scaling typically requires provisioning and committing additional infrastructure capacity to achieve the desired performance. For example, to enable the `Memcached` cluster to serve at higher throughput, one can increase the number of `Memcached` replicas and distribute the load across all the replicas.

Figure 10a and Figure 10b show the effect of horizontally scaling `Memcached` vs. when *rKube* is applied alone. For both tests, the deployed `Memcached` observes a target throughput of at least 300k rps. The vertical dashed lines denote the minimum number of replicas needed to achieve the target throughput. The minimum number of replicas is 7 and 9 in Figure 10a and Figure 10b, respectively, due to the different settings of neighboring containers. We observe that by increasing the number of replicas, the tail latency decreases gradually. On the other hand, with the help of *rKube*, the minimum number of replicas is 5 for both types of neighbors, with a tail latency of 606 and 613 ms. Finally, by comparing the results in Figure 10a and Figure 10b, we find that *rKube* is incredibly helpful when having burstable neighbors than when having capped neighbors. Capped neighbors will be throttled after their CPU consumption meets their demand, while burstable neighbors can be more aggressive in competing resources.

Overall, the results show that while scaling out can improve Memcached's throughput, the latency is also increased due to neighbor containers. *rKube* can mitigate such effects, delivering better service quality to users.

6.4 Resource under-commitment vs. *rKube*

By default, Kubernetes assigns all the allocatable CPUs of the host to containers. When a host node in a Kubernetes cluster runs out of resources, the kubelet (the primary "node agent" that runs on each host node) will be triggered to reclaim resources by evicting pods until the resource usage is under a pre-defined threshold again. Therefore, an admin or the cloud provider may intentionally leave some resources unassigned (i.e., *under-commitment*) to guarantee that there is enough resource headroom when the load of the (higher-priority) applications spikes. We call such a strategy "resource under-commitment". We study the effectiveness of resource under-commitment and compare it with that of *rKube*. We gradually reduce the number of CPU requests of the neighbor containers (so that more allocatable CPUs become available and can be utilized if needed) to simulate the over-supplied resource scenario. Figure 11 shows the performance improvement of `streamcluster`, when we reduce the number of CPUs assigned to the neighboring containers that are capped and running memory-intensive workloads.

In Figure 11a, the x-axis represents the number of assigned CPUs (to both the target and neighboring applications) vs. total allocatable CPUs. Throughout, the target application `streamcluster` always requests 12 CPUs, as listed in Table 1. For example, 20/22 means 20 are assigned to the applications (where 12 is assigned to `streamcluster` and 8 is to the neighbor), and 2 are allocatable but not assigned. For *rKube*, the number of assigned CPUs for the target application is fixed at 12. By default, all CPUs (22/22) are assigned to the target and neighboring containers. The performance of the target application in the default setting is used as our baseline for comparisons. As shown by the y-axis (normalized completion time), the target application's performance improves when the neighboring container requests fewer CPUs.

rKube is more effective and efficient; *rKube* correctly enforces the CPU shares and provides a performance guarantee for the target application. For example, the completion time is reduced by 40% when we decrease the number of allocated CPUs from 22/22 to 12/22. In this case, *rKube* outperforms the under-commitment strategy even when there are 10 additional unassigned CPUs (and could be used by the target).

Figure 11b compares under-commitment and *rKube* for `Memcached`. As shown, the throughput improves (while the 99%-ile tail latency decreases) as more CPUs become available (unassigned). We also observe similar trends for the latencies of `Memcached` update operations (omitted due to space limitations). Overall, when serving at the same throughput level, *rKube* significantly outperforms under-commitment strategies in terms of the 99%-ile tail latency. Furthermore, with the help of *rKube*, `Memcached` achieves higher maximum throughput (the horizontal dashed line in the figure), which is comparable with that of under-commitment. Again, unlike under-commitment, which sacrifices the overall CPU utilization (the CPU utilization drops from 22/22 to 16/22), *rKube* is able to maintain a high CPU utilization of 22/22.

With *rKube*, a higher level of performance improvement is achieved. More importantly, *rKube* maintains the highest level of CPU utilization without needing to sacrifice the resource share of the neighboring application. This is desirable for modern data centers that have long been suffering a notoriously low overall resource utilization [34, 35, 53, 54].

6.5 *rKube* with Alibaba traces

We evaluate *rKube* using production data center traces from Alibaba [31]. The original traces contain a 12-hour long workload of co-located long-running, interactive jobs and transient, batch jobs. The whole workload spans ~1300 machines.

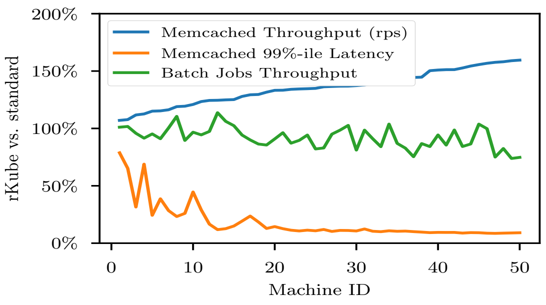


Figure 12: *rKube*'s performance with Alibaba traces, normalized against *standard*.

Since the trace datasets lack detailed information about application types, we use Memcached to simulate online, interactive jobs and the PARSEC benchmark suite to simulate offline, batch jobs. For each job, we set the requested CPUs based on the values of `plan_cpu` in the traces, and we derive the number of threads based on the values of `cpu_avg` and `cpu_max`. We use a 40-core server with 2x Intel Xeon Silver 4114 CPUs (10C20T) from CloudLab [39] for this experiment. We reserve 2 cores for systems and the rest of 38 cores are used for containers. We randomly choose 50 machines from the traces and replay the workload assigned to that particular machine using our settings for a fixed period of time.

Figure 12 reports the normalized performance of *rKube*. We observe that both the throughput and the 99%-ile tail latency of Memcached get improved compared to those of *standard* by an average of 35% and 451%, respectively, with the tail latency being as much as 10× faster. The throughput of batch jobs, defined the number of batch jobs completed per minute, however, shows some level of variance across machine IDs—batch jobs run 8% slower on average under *rKube* than under *standard*. The result is expected—by reserving CPUs for specific containerized applications, *rKube* may lead to some level of CPU under-utilization. Different from previous experiments, here the resource of the batch jobs are uncapped and not isolated by *rKube*, given their lower priority in data centers. The results demonstrate the efficacy of *rKube* in sustaining real-world, production data center workloads, as data center operators or cloud providers typically assign higher priorities for online, interactive, directly-user-facing applications while providing best-effort, temporary resource over-commitment for short-lived batch jobs with lower priorities [54, 55, 58]. We believe this is mostly acceptable considering that *rKube* can significantly reduce SLA violations and most batch jobs have low priorities.

7 RELATED WORK

The Kubernetes and Docker community has long been suffering from the performance issues related to Linux scheduling and OS virtualization [8, 11–13, 15]. Kubernetes provides alternative host CPU management policies [42], e.g., *static* policy, which utilizes *cpuset.cpu* to pin the pods for better performance. However, the *static* policy is intended for better cache affinity and lower scheduling overhead, not reducing neighbor interference. It also comes with its own limitations. For example, it can be applied if and only if it is a *Guaranteed* pod and has integer CPU *requests*. Those pods will still share the CPU with system services such as the container runtime and the kubelet. Also, a host cannot have mixed static and sharing containers at the same time. As a comparison, *rKube* is specifically designed to reduce interference and can be enabled on selected pods on a host with other CPU sharing pods running together.

In the meantime, the Linux community has been actively investigating new cases where Linux fails to provide proper CPU isolation support for containerized applications [6, 20, 21, 26, 27]. However, despite the efforts. Many of these performance issues still persist with the latest software stack, stressing that improvement for the containerized environment is imperatively needed.

Researchers have identified drawbacks of the CPU schedulers. For example, Lozi et al. [49] reported that the Linux scheduler sometimes fails to make good use of the CPU resources and causes performance degradation for some applications. Kim et al. [44] found that CFS with distributed runqueues fails to achieve a global fare share scheduling, and proposed a global virtual time fair scheduling as a solution. Bouron et al. [33] analyzed the impact of two OS schedulers: FreeBSD's default ULE scheduler, and Linux's default CFS, on applications performance, and concluded there is no overall winner for complex use cases. Our work focuses on the context of containerized environments, where fulfilling the promise of CPU resource requests is crucial to users.

Considerable prior works have examined performance isolation in multi-tenant clouds. CPI^2 [57] used cycle-per-instruction data to identify and throttle misbehaved tasks in the shared hosts. VASE [48] claimed that the protection scope is erroneously used as the resource scope in the multi-tenant environment and proposed virtual CPUs as a container for CPU accounting and management. Bubble-Up [50] identifies contention on the shared resource as a major obstacle for high-priority, latency-sensitive tasks to share hosts with other tasks, and provides a characterization method that predicts the performance degradation due to the contention on the shared resource in the memory subsystem. Studies showed that small latency variation per microservice would result in significant (tail) latency increases, and severely impact the end-user experience [36, 41, 59]. *rKube* also targets the multi-tenant container cloud environment but is among the first to provide a solution to address this issue. Though contention on shared hardware is commonly known as a challenge for performance isolation, we demonstrate that, even without CPU over-commitment, performance isolation is non-trivial due to the Linux scheduler design.

Some other studies have explored container performance isolation. Iron [43] improves the network CPU isolation by accounting for the CPU time spent on the network stack on behalf of the co-located containers. Gao et al. [42] argued that Linux cgroups fails to achieve consistent and fair resource accounting, and show that the resource consumption is not correctly charged to the specified cgroup configurations. In our work, we show that, in addition to those issues, the mismatched interaction between cgroups and Linux's default CFS scheduler is the main cause of severe performance interference.

8 CONCLUSION

Containerized cloud environments are becoming more and more popular in the production environment. While offering plenty of advantages, it has also been found that the application performance suffers from significant variations, making it difficult for resource requesting (for users) and resource planning (for cloud providers). In this paper, we have quantitatively evaluated the performance variations of batch and interactive applications, and showed that the application could suffer a slowdown of 5x, leading to SLA violations. These results motivated us to reason the underlying causes, which point to the forced runqueue sharing and the phantom CPU time due to the scheduling mechanism used in the underlying host. We have shown the problem is due to the misaligned design goals of the scheduler and the containers, which mandates a new kernel scheduler redesign. As a proof-of-concept, we have designed and implemented *rKube*, a CFS-transparent alternative, by augmenting the existing Kubernetes with an additional option, making it backward compatible. Our evaluation results show that *rKube* can effectively deliver the performance corresponding to a user's requests and outperform the common best practices for scaling up in the production environments for improving the application's performance, demonstrating what can and should be achieved in the future efforts.

rKube is open-sourced and is available at: https://github.com/njuliuli/kubernetes/tree/policy.

ACKNOWLEDGMENTS

We appreciate the constructive comments from the reviewers and our shepherd Michael Swift. This work has been supported in part by NSF grants CNS-2007153, CNS-2008468, CNS-2045680, and CCF-1919075. Some authors have also been supported in part by a Google Faculty Research Award, a Commonwealth Cyber Initiative grant, an unrestricted gift from Adobe Research, and the National Natural Science Foundation of China No.6210070716.

REFERENCES

[1] alexeiled/stress-ng - docker hub. https://hub.docker.com/r/alexeiled/stress-ng. (Accessed on 09/16/2020).

[2] Amazon eks - managed kubernetes service. https://aws.amazon.com/eks/. (Accessed on 05/29/2020).

[3] Amazon eks pricing | managed kubernetes service | amazon web services. https://aws.amazon.com/eks/pricing/. (Accessed on 01/01/2021).

[4] Assign cpu resources to containers and pods | kubernetes. https://kubernetes.io/docs/tasks/configure-pod-container/assign-cpu-resource. (Accessed on 09/16/2020).

[5] bitnami/memcached - docker hub. https://hub.docker.com/r/bitnami/memcached. (Accessed on 09/16/2020).

[6] Bkk19-tr06 - deep dive in the scheduler - youtube. https://www.youtube.com/watch?v=1xhK0cH2Dkg&ab_channel=LinaroOrg. (Accessed on 09/15/2020).

[7] Cfs bandwidth control – the linux kernel documentation. https://www.kernel.org/doc/html/latest/scheduler/sched-bwc.html. (Accessed on 01/01/2021).

[8] Cfs quotas can lead to unnecessary throttling . issue #67577 . kubernetes/kubernetes. https://github.com/kubernetes/kubernetes/issues/67577. (Accessed on 09/15/2020).

[9] Cfs scheduler – the linux kernel documentation. https://www.kernel.org/doc/html/latest/scheduler/sched-design-CFS.html. (Accessed on 09/06/2020).

[10] Containerdefinition - amazon elastic container service. https://docs.aws.amazon.com/AmazonECS/latest/APIReference/API_ContainerDefinition.html. (Accessed on 01/01/2021).

[11] Cpu considerations for java applications running in docker and kubernetes | by christopher batey | medium. https://link.medium.com/H3WcqAfND9. (Accessed on 09/15/2020).

[12] Cpu scheduler imbalance with cgroups | josef bacik's blog. https://josefbacik.github.io/kernel/scheduler/cgroup/2017/07/24/scheduler-imbalance.html. (Accessed on 09/15/2020).

[13] Cpu throttling - unthrottled: Fixing cpu limits in the cloud. https://engineering.indeedblog.com/blog/2019/12/unthrottled-fixing-cpu-limits-in-the-cloud/. (Accessed on 09/15/2020).

[14] Horizontal scaling - aws well-architected framework. https://wa.aws.amazon.com/wat.concept.horizontal-scaling.en.html. (Accessed on 09/15/2020).

[15] How to optimize i/o intensive containers on kubernetes - neuvector. https://neuvector.com/container-security/optimize-i-o-intensive-containers/. (Accessed on 09/15/2020).

[16] Kubernetes - google kubernetes engine (gke) | google cloud. https://cloud.google.com/kubernetes-engine. (Accessed on 05/29/2020).

[17] Load tracking in the scheduler [lwn.net]. https://lwn.net/Articles/639543/. (Accessed on 09/06/2020).

[18] Managing resources for containers | kubernetes. https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/. (Accessed on 08/27/2020).

[19] memcached - a distributed memory object caching system. https://memcached.org/. (Accessed on 06/03/2020).

[20] [ospm-summit-17] parameterizing cfs load balancing: nr_running/util/load - youtube. https://www.youtube.com/watch?v=JyA5MpVpAAM&ab_channel=RetisLab. (Accessed on 09/15/2020).

[21] [ospm-summit-17] tracepoints for pelt - youtube. https://www.youtube.com/watch?v=tyoFqxviXOY&ab_channel=RetisLab. (Accessed on 09/15/2020).

[22] Per-entity load tracking [lwn.net]. https://lwn.net/Articles/531853/. (Accessed on 09/06/2020).

[23] Pod overview - kubernetes. https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/. (Accessed on 05/29/2020).

[24] Production-grade container orchestration - kubernetes. https://kubernetes.io/. (Accessed on 05/29/2020).

[25] Reserve compute resources for system daemons | kubernetes. https://kubernetes.io/docs/tasks/administer-cluster/reserve-compute-resources/. (Accessed on 08/27/2020).

[26] Rework cfs load balance - youtube. https://www.youtube.com/watch?v=cfv63BMnIug&ab_channel=RetisLab. (Accessed on 09/15/2020).

[27] San19-220 deep dive in the scheduler - youtube. https://www.youtube.com/watch?v=_re97U8Vlzc&ab_channel=LinaroOrg. (Accessed on 09/15/2020).

[28] spirals/parsec-3.0 - docker hub. https://hub.docker.com/r/spirals/parsec-3.0. (Accessed on 06/05/2020).

[29] Swarm mode overview | docker documentation. https://docs.docker.com/engine/swarm/. (Accessed on 05/29/2020).

[30] Work-conserving scheduler - wikipedia. https://en.wikipedia.org/wiki/Work-conserving_scheduler. (Accessed on 01/01/2021).

[31] Alibaba. Alibaba cluster trace. https://github.com/alibaba/clusterdata/. (Accessed on 01/11/2021).

[32] Bienia, C. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[33] Bouron, J., Chevalley, S., Lepers, B., Zwaenepoel, W., Gouicem, R., Lawall, J., Muller, G., and Sopena, J. The battle of the schedulers: Freebsd {ULE} vs. linux {CFS}. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)* (2018), pp. 85–96.

[34] Cheng, Y., Anwar, A., and Duan, X. Analyzing alibabaâĂŹs co-located datacenter workloads. In *2018 IEEE International Conference on Big Data (Big Data)* (2018), pp. 292–297.

[35] Cheng, Y., Chai, Z., and Anwar, A. Characterizing co-located datacenter workloads: An alibaba case study. In *Proceedings of the 9th Asia-Pacific Workshop on Systems* (New York, NY, USA, 2018), APSys '18, Association for Computing Machinery.

[36] Cheng, Y., Gupta, A., and Butt, A. R. An in-memory object caching framework with adaptive load balancing. In *Proceedings of the Tenth European Conference on Computer Systems* (New York, NY, USA, 2015), EuroSys '15, Association for Computing Machinery.

[37] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (New York, NY, USA, 2010), SoCC '10, Association for Computing Machinery, pp. 143–154.

[38] Dean, J., and Ghemawat, S. Mapreduce: Simplified data processing on large clusters.

[39] Duplyakin, D., Ricci, R., Maricq, A., Wong, G., Duerig, J., Eide, E., Stoller, L., Hibler, M., Johnson, D., Webb, K., Akella, A., Wang, K., Ricart, G., Landweber, L., Elliott, C., Zink, M., Cecchet, E., Kar, S., and Mishra, P. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)* (July 2019), pp. 1–14.

[40] Fried, J., Ruan, Z., Ousterhout, A., and Belay, A. Caladan: Mitigating interference at microsecond timescales. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)* (2020), pp. 281–297.

[41] Gan, Y., Zhang, Y., Cheng, D., Shetty, A., Rathi, P., Katarki, N., Bruno, A., Hu, J., Ritchken, B., Jackson, B., Hu, K., Pancholi, M., He, Y., Clancy, B., Colen, C., Wen, F., Leung, C., Wang, S., Zaruvinsky, L., Espinosa, M., Lin, R., Liu, Z., Padilla, J., and Delimitrou, C. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud and Edge Systems. In *Proceedings of the Twenty Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (April 2019).

[42] Gao, X., Gu, Z., Li, Z., Jamjoom, H., and Wang, C. Houdini's escape: Breaking the resource rein of linux control groups. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications*

Security (2019), pp. 1073–1086.

[43] Khalid, J., Rozner, E., Felter, W., Xu, C., Rajamani, K., Ferreira, A., and Akella, A. Iron: Isolating network-based {CPU} in container environments. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)* (2018), pp. 313–328.

[44] Kim, C., Choi, S., and Huh, J. Gvts: Global virtual time fair scheduling to support strict fairness on many cores. *IEEE Transactions on Parallel and Distributed Systems 30*, 1 (2018), 79–92.

[45] King, C. I. Stress-ng. https://kernel.ubuntu.com/~cking/stress-ng/, 2020. Accessed: 05/20/2020.

[46] Leverich, J., and Kozyrakis, C. Reconciling high server utilization and sub-millisecond quality-of-service. In *Proceedings of the Ninth European Conference on Computer Systems* (New York, NY, USA, 2014), EuroSys '14, Association for Computing Machinery.

[47] Li, J., Sharma, N. K., Ports, D. R., and Gribble, S. D. Tales of the tail: Hardware, os, and application-level sources of tail latency. In *Proceedings of the ACM Symposium on Cloud Computing* (2014), pp. 1–14.

[48] Liu, L., Wang, H., Wang, A., Xiao, M., Cheng, Y., and Chen, S. Vcpu as a container: Towards accurate cpu allocation for vms. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (New York, NY, USA, 2019), VEE 2019, Association for Computing Machinery, p. 193âĂŞ206.

[49] Lozi, J.-P., Lepers, B., Funston, J., Gaud, F., Quéma, V., and Fedorova, A. The linux scheduler: A decade of wasted cores. In *Proceedings of the Eleventh European Conference on Computer Systems* (New York, NY, USA, 2016), EuroSys '16, Association for Computing Machinery.

[50] Mars, J., Tang, L., Hundt, R., Skadron, K., and Soffa, M. L. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture* (2011), pp. 248–259.

[51] Merkel, D. Docker: Lightweight linux containers for consistent development and deployment. *Linux J. 2014*, 239 (Mar. 2014).

[52] Ousterhout, A., Fried, J., Behrens, J., Belay, A., and Balakrishnan, H. Shenango: Achieving high {CPU} efficiency for latency-sensitive datacenter workloads. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)* (2019), pp. 361–378.

[53] Reiss, C., Tumanov, A., Ganger, G. R., Katz, R. H., and Kozuch, M. A. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing* (New York, NY, USA, 2012), SoCC '12, Association for Computing Machinery.

[54] Tirmazi, M., Barker, A., Deng, N., Haque, M. E., Qin, Z. G., Hand, S., Harchol-Balter, M., and Wilkes, J. Borg: The next generation. In *Proceedings of the Fifteenth European Conference on Computer Systems* (New York, NY, USA, 2020), EuroSys '20, Association for Computing Machinery.

[55] Verma, A., Pedrosa, L., Korupolu, M., Oppenheimer, D., Tune, E., and Wilkes, J. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems* (New York, NY, USA, 2015), EuroSys '15, Association for Computing Machinery.

[56] Woo, S. C., Ohara, M., Torrie, E., Singh, J. P., and Gupta, A. The splash-2 programs: Characterization and methodological considerations. *ACM SIGARCH computer architecture news 23*, 2 (1995), 24–36.

[57] Zhang, X., Tune, E., Hagmann, R., Jnagal, R., Gokhale, V., and Wilkes, J. Cpi2: Cpu performance isolation for shared compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems* (2013), pp. 379–391.

[58] Zhang, Z., Li, C., Tao, Y., Yang, R., Tang, H., and Xu, J. Fuxi: A fault-tolerant resource management and job scheduling system at internet scale. *Proc. VLDB Endow. 7*, 13 (Aug. 2014), 1393–1404.

[59] Zhou, H., Chen, M., Lin, Q., Wang, Y., She, X., Liu, S., Gu, R., Ooi, B. C., and Yang, J. Overload control for scaling wechat microservices. In *Proceedings of the ACM Symposium on Cloud Computing* (New York, NY, USA, 2018), SoCC '18, Association for Computing Machinery, pp. 149–161.