XX

Graduate Category

1 Introduction

Developers running services on cloud provders like AWS or Google, or containerization systems like Kubernetes, reserve the CPU and memory that service will run on. Workloads requiring such guarantees are *latency critical* (LC).

These reservations leads to a utilization problem: the load on most applications is variable, so developers choose reservations based on the expected peak load [6, 10, 11].

Best effort (BE) workloads do not have reservations and their completion time is not critical. Popular examples include long-running MapReduce jobs or background data analytics.

Running BE workloads on the same resources as LCs helps solve the utilization problem; the resource that this work focuses on is CPU. BE tasks can run on unused CPUs opportunistically, the challenge is to enforce resource reservations for LC services while achieving good utilization.

Current popular scheduling systems fail to properly enforce reservations. When running a small social web application on Kubernetes, we observe significant impacts on its latency when starting a BE image resize job on the same cores. *Figure* 1 shows the end-to-end latency of an endpoint that gets a users feed, and the throughput of the BE job. After the BE job starts running, the mean response latency of the server jumps from ~7ms to ~15ms.

2 Problem

The underlying mechanism enforcing CPU reservations for most modern containers running on Linux is `cgroups`' weight interface: all Open Container Initiative (OCI) compliant containers, including Kubernetes, Docker, CRI-O, and containerd use it [2, 8, 9], as do VM frameworks, including Firecracker, AFaas and libvirt [1, 3, 5]. Kubernetes for instance creates one top level group for all BE services and assigns it the lowest weight of 1, and pods with reservations are in separate groups with higher weights, e.g., in the experiment the server asked for 4 CPUs and Kubernetes assigned its pod a weight of 157.

A simplified experiment shows that `cgroups` weights are unable to enforce reservations. We run an open-loop client on a remote machine, and then start two best effort workloads doing image resizing. We put the LC server and the BE resize job each in their own group, and pin them to the same four cores. `cgroups` supports weights in the range of [1,10000], Figure 2 shows that running the server with different weights has no effect on the latency impact of the BE task.

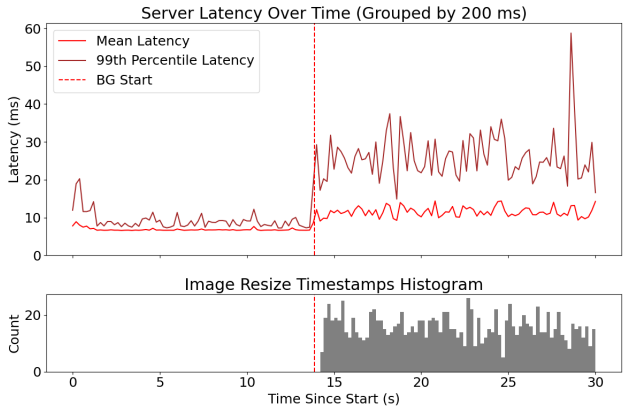
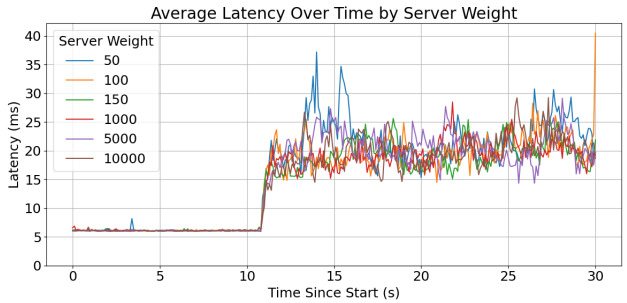
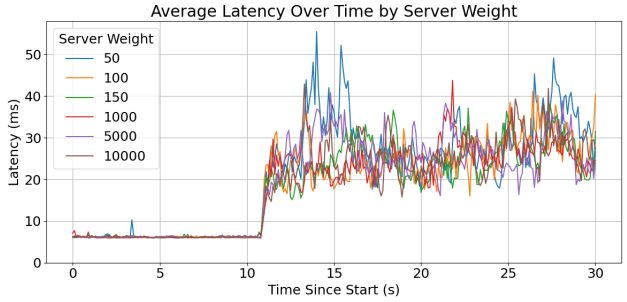


Figure 1: In Kubernetes, running a best effort application has a latency impact on server that has reserved resources



(a) Low load, utilization before starting the BE tasks is around 85%



(b) High load, utilization before starting the BE tasks is around 95%

Figure 2: Changing the weight of the server has little impact on how much the weight 1 BE task interferes with it

The spike in latency after starting the BE happens because the BE sometimes runs uncontended on one core, even while another has queued and waiting server threads. Figure 3 shows an outtake from the trace, where on core 6, the red

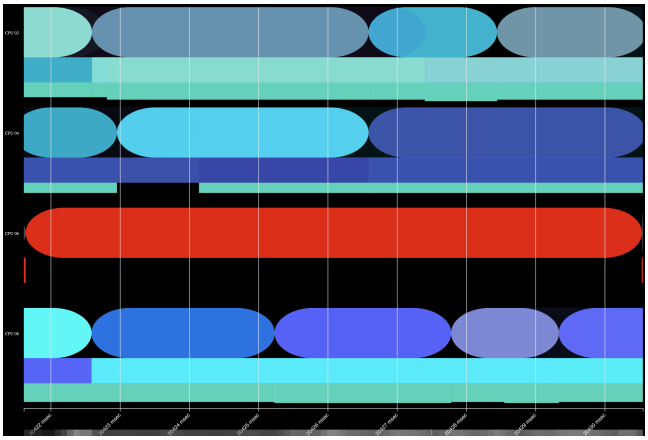


Figure 3: Core 6 runs an image resize process, unaware that cores 2, 4, and 8 all have runnable and queued server threads

process that is running for the whole 10ms is a thread of the BE job, while server threads, shown in varying shades of blue, are queued on the other cores.

This priority inversion happens because weights are only enforced within runqueues, which are per-CPU. Although load balancing eventually remedies this, it runs less frequently than scheduling does. For sharing across long-running processes, balancing occasionally works well, but for workloads with runtimes in the ms, these delays significiantly influence final processing times.

In order to globally enforce weights, each scheduling decision would require a global search for the process that should be run next. We conclude that weights are the wrong interface to use to honor reservations globally.

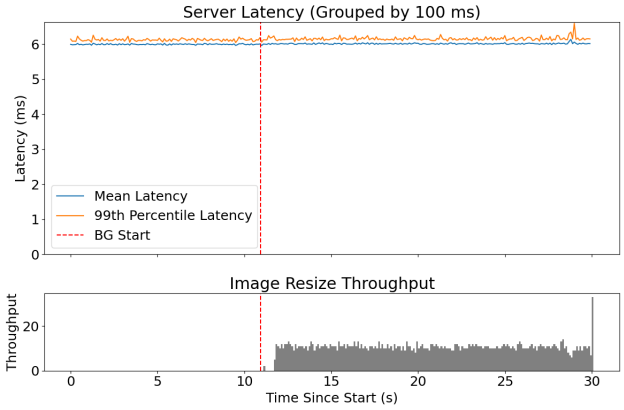
3 Approach & Background

In order to enforce reservations while still running best effort jobs opportunistically, our approach is use priority scheduling.

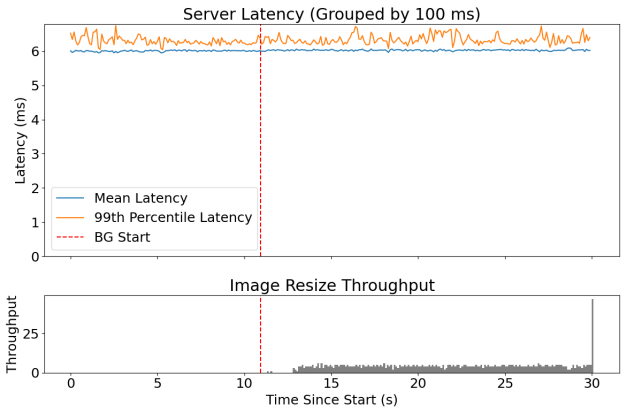
Enforcing priorities requires fewer global runqueue searches than weights do, because they only need to happen on *class boundary crossings*: on *exit*, when a core switches to running lower class processes after having previously been running high class, and on *entry*, when a core enqueues a higher class process.

Linux already provides priority scheduling across scheduling classes, of which it has three that are accessible to users (in descending order of priority): `Deadline`, `RTClass`, and `Normal`. Most load falls into the `Normal` scheduling class, it is only within the `Normal` scheduling class that the `cgroups` weight interface is relevant.

Linux prioritizes strictly between different scheduling classes, and each class has its own scheduling algorithm, keeps its own runqueues, and balances its own load across cores.



(a) Low load (85%)



(b) High load (95%)

Figure 4: when running the server as in `RTClass`, Linux enforces the server's access to its reserved cores

Figure 4 shows the result of running the microbenchmark with the LC server in the `RTClass` scheduling class. We can see that in both load settings the tail and average latency stays stable at ~6.0ms after starting the BE workload.

However, running microservices in `RTClass` is untenable because of `RTClass`'s intra-priority schedulers. Both do not support the `cgroups` weight interface, which although not good for scheduling LC and BE, is also used to allocate CPU time within LC workloads; Kubernetes, for instance, allows users to make fractional CPU requests, which are enforced using weights.

4 Design & Implementation

We design a new scheduling class `BeClass` that sits at a lower priority than `Normal`, which enforces LC applications' uncontended access to the CPUs they reserved. Doing so allows the LC microservices to stay in the same scheduling

class they were before, while getting the benefits of strict prioritization across scheduling classes.

The goal of the implementation is to enforce that no `BeClass` userspace process is ever running on cores reserved for a `Normal` workload if a `Normal` process is queued. The scheduler enforces the priorities in three different places:

(1) *local*: on each core it ensures that no `BeClass` process is chosen if there is a runnable `Normal` process,
(2) *entry*: before enqueuing a `Normal` process, a core looks for other cores running `BeClass` entities to interrupt,
(3) *exit*: when the last `Normal` process on a core's queue blocks or exits, the core tries to steal queued `Normal` processes from other cores before running a `BeClass` process.

Scheduling *classes* in Linux can have multiple *policies*. In order to implement `BeClass` in Linux, we build on `sched_idle`, an existing scheduling *policy* in the `Normal`. We do so because `sched_idle` has some of the features we want for `BeClass`.

One feature is that `sched_idle` was extended to be accessible via the `cgroups` API recently [4]: a whole groups' policy can be set to `sched_idle` via the `cgroups` interface. The other is that Linux developers also added what is in effect the *entry* check from the `BeClass` design [7].

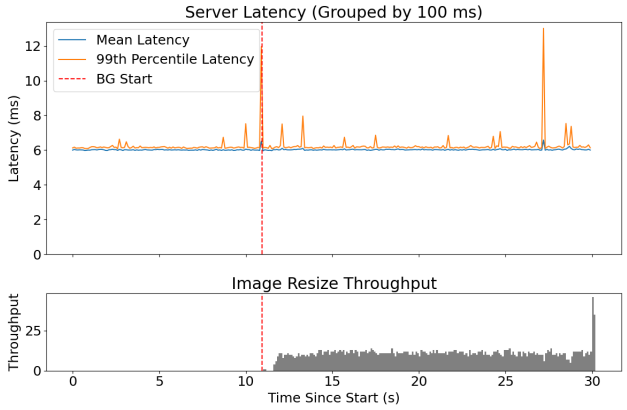
Our implementation adds the *local* and *exit* parts of the `BeClass` design.

5 Evaluation

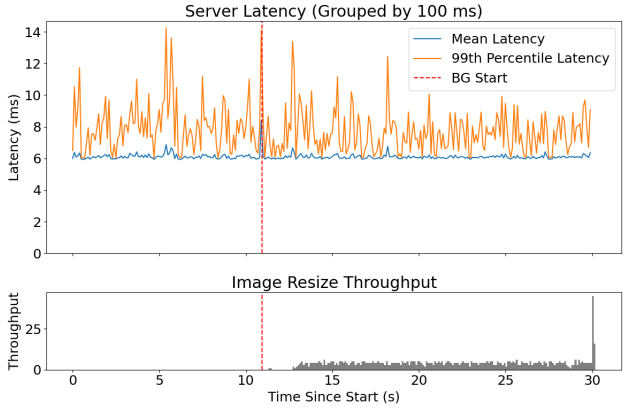
We run the microbenchmark experiment from Figure 2 using `BeClass`. We can see the resulting performance in Figure 5. As desired, the latency of the server remains stable after the background tasks start, and the background task still runs.

Importantly, the background tasks will reliably get interrupted when the LC server has a request to process. Figure 6 shows the interruption happening in an outtake of a trace. The green BE processes run only in the gaps where there is no queued LC process, and are immediately preempted when one wakes up, on whatever core that may be. The vertical red lines show when the *exit* path `BeClass` introduced runs. As we can see, this line is often followed by the core running an LC process, indicating it succesfully found and stole a queued LC thread.

We also run the Kubernetes application from Figure 1 using `BeClass`. The results are in Figure 7. We can see that the baseline mean latency of the LC server stays around 7.4ms after starting the the BE image resizing.



(a) Low load (85%)



(b) High load (95%)

Figure 5: `BeClass` does a good job of isolating the server's latencies from the load from best effort jobs

References

[1] Xiaohu Chai, Tianyu Zhou, Keyang Hu, Jianfeng Tan, Tiwei Bie, Anqi Shen, Dawei Shen, Qi Xing, Shun Song, Tongkai Yang, Le Gao, Feng Yu, Zhengyu He, Dong Du, Yubin Xia, Kang Chen, and Yu Chen. "Fork in the Road: Reflections and Optimizations for Cold Start Latency in Production Serverless Systems". In: *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. Boston, MA: USENIX Association, 2025. URL: https://www.usenix.org/system/files/osdi25-chai-xiaohu.pdf.

[2] Docker. *dockerdocs: Resource constraints*. online. URL: https://docs.docker.com/engine/containers/resource_constraints/.

[3] libvirt Documentation. *CPU Allocation*. online. URL: https://libvirt.org/formatdomain.html#cpu-allocation.

[4] Josh Don. *sched: Cgroup SCHED_IDLE support*. online. 2021. URL: https://lore.kernel.org/all/162971078674.25758.15464079371945307825.tip-bot2@tip-bot2/.

[5] firecracker. *config-linux.md: Production Host Setup Recommendations*. github. URL: https://github.com/firecracker-microvm/firecracker/blob/main/docs/prod-host-setup.md.

[6] Will Kelly. *Overprovisioning in AWS? Cost-control tools and strategies can help*. online. 2013. URL: https://www.techtarget.

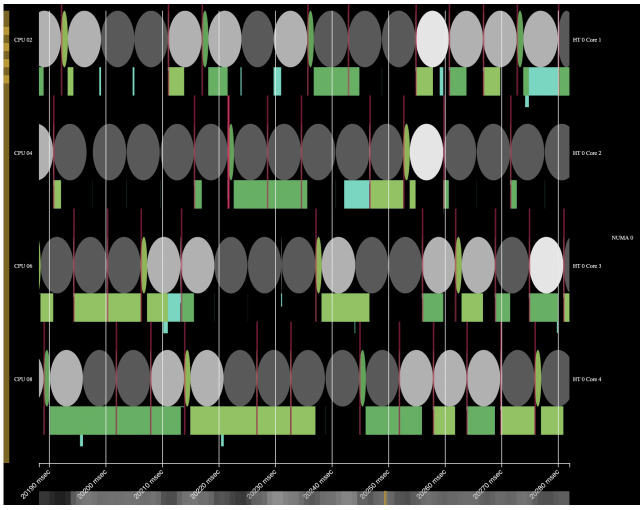


Figure 6: BE threads only run in the gaps when there are no queued LC threads on any core

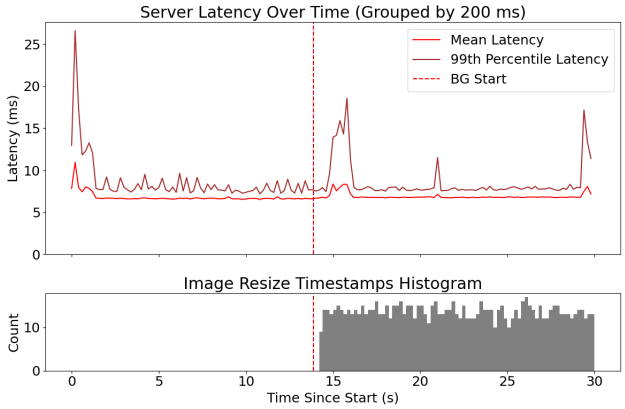


Figure 7: The same experiment as in Figure 1, but running the BE as a `BeClass` task

com/searchaws/news/2240209793/Overprovisioning-in-AWS-Cost-control-tools-and-strategies-can-help.

[7] Viresh Kumar. *Fixing SCHED_IDLE*. online. 2019. URL: https://lwn.net/Articles/805317/.

[8] Rory McClune. *How Do Containers Contain? Container Isolation Techniques*. online. 2021. URL: https://www.aquasec.com/blog/container-isolation-techniques/.

[9] opencontainers. *config-linux.md: Linux Container Configuration*. github. URL: https://github.com/opencontainers/runtime-spec/blob/main/config-linux.md.

[10] Zhenyuan Ruan, Seo Jin Park, Marcos K. Aguilera, Adam Belay, and Malte Schwarzkopf. "Nu: Achieving Microsecond-Scale Resource Fungibility with Logical Processes". In: *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. Boston, MA: USENIX Association, Apr. 2023, pages 1409–1427. URL: https://www.usenix.org/conference/nsdi23/presentation/ruan.

[11] Muhammad Tirmazi, Adam Barker, Nan Deng, Md Ehtesam Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. "Borg: the Next Generation". In: *EuroSys'20*. Heraklion, Crete, 2020.