XX

Graduate Category

1 Motivation

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 [5, 8, 9]. *Best effort* (BE) workloads do not have reservations and their completion time is not critical.

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 & Background

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, 6, 7], as do VM frameworks, including Firecracker, AFaas and libvirt [1, 3, 4]. 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.

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 process that is running for the whole 10ms is a thread of

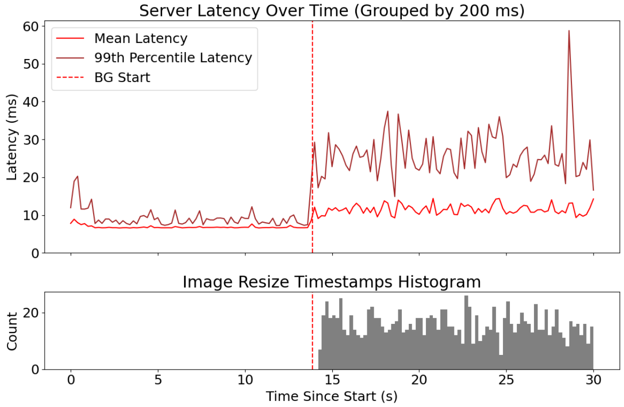

Figure 1: In Kubernetes, running a best effort application has a latency impact on server that has reserved resources

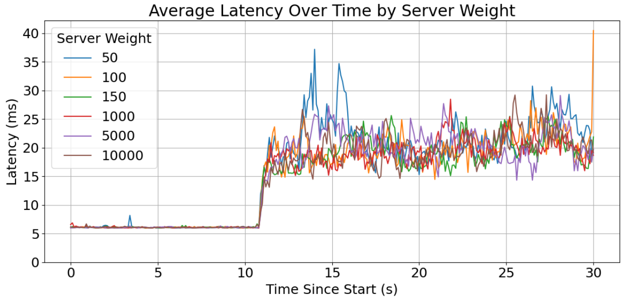

Figure 2: Changing the weight of the server has little impact on how much the weight 1 BE task interferes with it

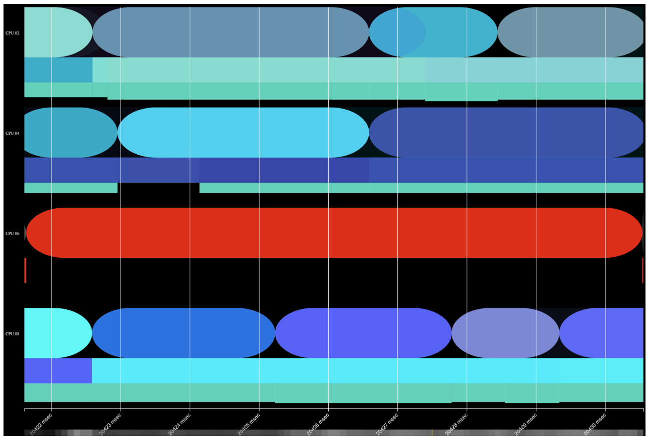

Figure 3: Core 6 runs an image resize process, unaware that cores 2, 4, and 8 all have runnable and queued server threads

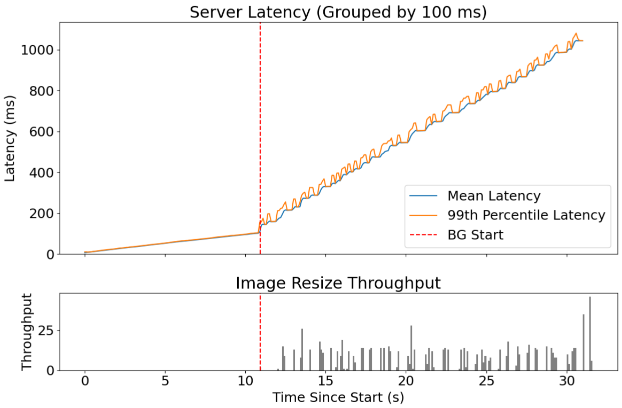

Figure 4: LC in real time, throttling

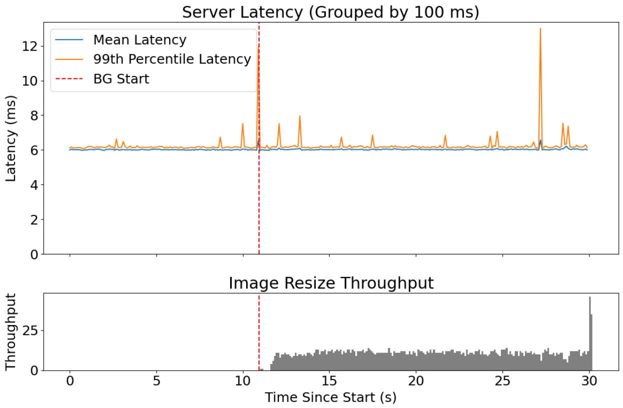

Figure 5: BeClass does a good job of isolating the server's latencies from the load from best effort jobs

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 & Uniqueness

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. These checks ensure that if a core c is currently running a BE thread t, the scheduler knows that there are no queued and waiting LC threads anywhere on the machine. If there is a queued LC thread on a different core c' when c starts running t, the *exit* check, that looks at every cores' runqueue, will see and steal it. If a new LC thread wakes up on a different core while t is running, the *entry* check ensures that c' will look at c's runqueue, see that it is running a BE thread, and will send the new LC thread to c via an interrupt, where the interrupt handler will trigger a scheduling that interrupts t.

Linux already provides priority scheduling across scheduling classes, which are used to separate real-time applications from all other workload. The priority scheduling between real-time and the default scheduling classes comes with an asterisk: if real-time applications experience high load, Linux throttles them in order to not starve the default class. We can see the result in Figure 4, where throttling leads to spikes in the background task as it is able to run, and corresponding spikes in the server's latency.

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. To enforce reservations even under high load, without throttling the latency critical services or killing the best effort ones, BeClass *parks* best effort processes. Parking addresses the potential issues with starvation by separating BeClass processes' logic into application logic (user-space) and critical-state management (kernel-space). When load spikes to 100%, BeClass processes' run only kernel-level services.

4 Results

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 (but only in gaps where the core would otherwise be idle)

We also run the Kubernetes application from Figure 1 using BeClass. The results are in Figure 6. We can see that the baseline mean latency of the LC server stays around 7.4ms after starting the the BE image resizing.

Figure 7 shows how parking enables the latency critical server to keep its reservation even under sustained extremely high load. We run the same client load as Figure 4, but now instead instead of throttling the server, sched_be parks the best effort processes. Notice that the sched_be image resize job does not make progress until the very end, when the server is done processing the requests.

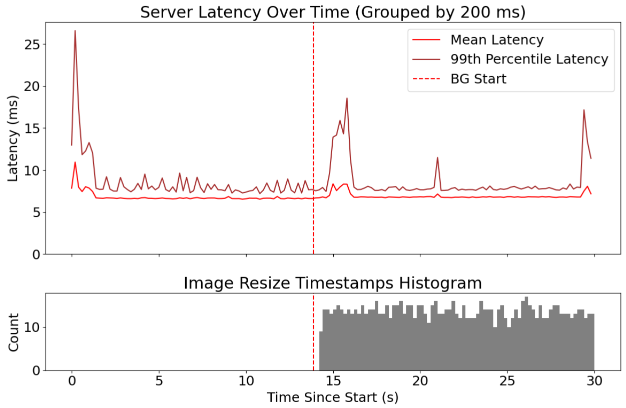

Figure 6: The same experiment as in Figure 1, but running the BE as a BeClass task

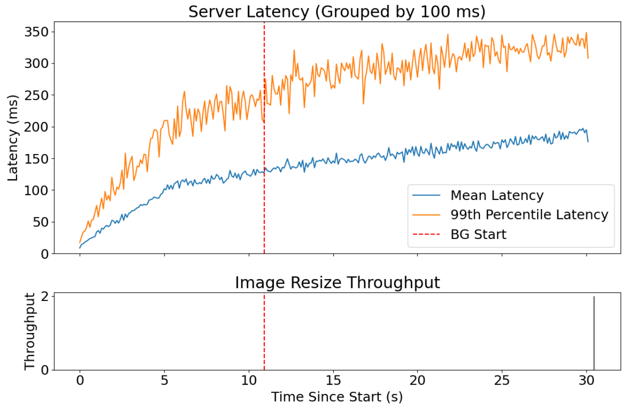

Figure 7: BE in sched_be, no throttling

5 Contributions

This work shows that cgroups weights, used by systems like Kubernetes to separate LC from BE, are unable to effectively honor the reservations of LC applications in the presence BE workloads, and that this is because Linux uses per-core runqueues.

Instead, we propose an API that separates BE from LC by introducing BeClass, which requires fewer cross-core interactions than a weight-based approach, and ensures that no BE is ever running when an LC is queued.

We implement this strict priority in Linux, and show that BeClass allows cgroups itself, as well as higher level applications like Kubernetes, to ensure LC processes' access to their reserved cores while running BE workloads opportunistically.

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

[5] Will Kelly. *Overprovisioning in AWS? Cost-control tools and strategies can help*. online. 2013. URL: https://www.techtarget.com/searchaws/news/2240209793/Overprovisioning-in-AWS-Cost-control-tools-and-strategies-can-help.

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

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

[8] 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.

[9] 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.