XX

Graduate Category

1 Motivation

Cloud providers like AWS, or containerization systems like Kubernetes, support reserving CPU for *latency critical* (LC) services. Developers choose reservations based on expected peak load [5, 8, 9], but load is variable, leading to a utilization problem.

Best effort (BE) workloads do not have reservations, so providers run them opportunistically on the same resources as LCs reserved; the resource that this work focuses on is CPU. The challenge is to honor resource reservations for LCs while opportunistically running BEs.

Current popular scheduling systems fail to honor reservations. *Figure* 1 shows the latency of an endpoint in a web application on Kubernetes that gets a users feed, and the throughput of a BE image resize job on the same cores. The mean response latency of the server jumps from ~7ms to ~15ms after starting the BE.

2 Problem & Background

All Open Container Initiative (OCI) compliant containers, including Kubernetes, Docker, CRI-O, and containerd, enforce CPU reservations using `cgroups`' weight interface [2, 6, 7], as do VM frameworks, including Firecracker, AFaas and libvirt [1, 3, 4]. Kubernetes creates one group for all BE services, given the lowest weight 1, and pods with reservations are in separate groups with higher weights (in the experiment the server requested 4 CPUs and Kubernetes assigned it a pod with weight 157).

A simplified experiment shows `cgroups` weights are unable to enforce reservations. We run an open-loop client on a remote machine, with a simple LC CPU-bound server and BE image resize job sharing four cores, each in their own group. `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 latency impact on the LC happens because the BE occasionally runs uncontended on one core, while another has queued server threads. Figure 3 shows this happening in the trace: on core 6, the red process running for 10ms is a BE thread, while server threads, in 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 workloads with request processing times in the millisecond range, waiting for the load balancer to run influences final processing times.

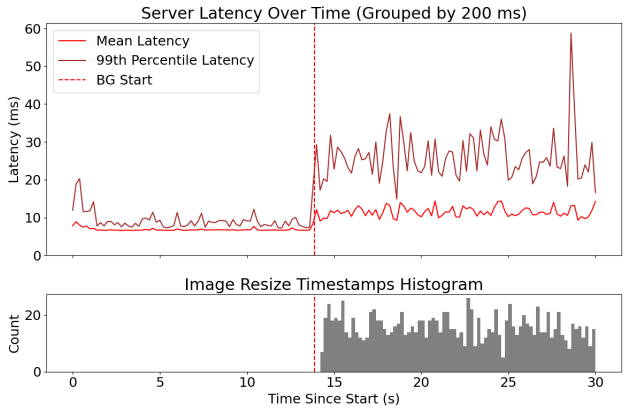

Figure 1: In Kubernetes, running a BE has latency impacts on an LC with reservations

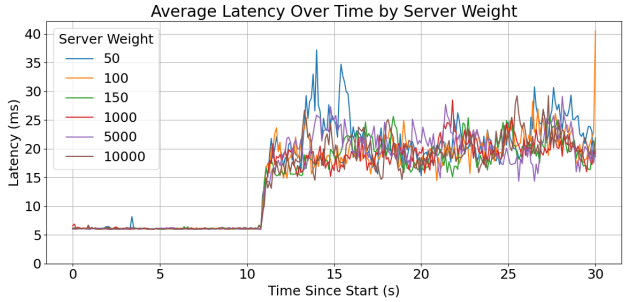

Figure 2: the weight of the server has little impact on how much the weight 1 BE task interferes

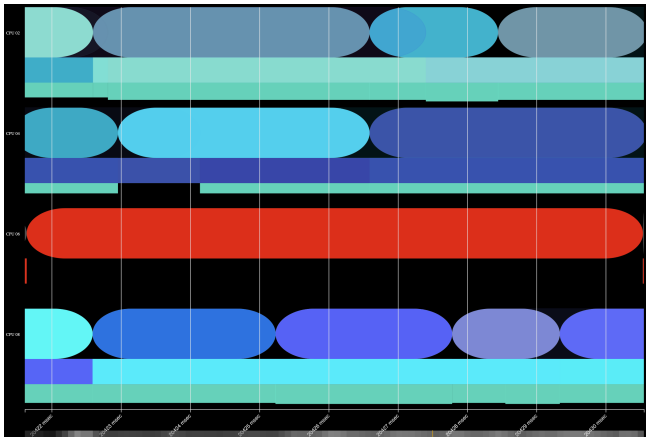

Figure 3: Core 6 runs an image resize process, unaware that the other cores have queued server threads

In order to globally enforce weights, each scheduling decision would require a search of all runqueues for potentially higher-weight processes, which is too costly.

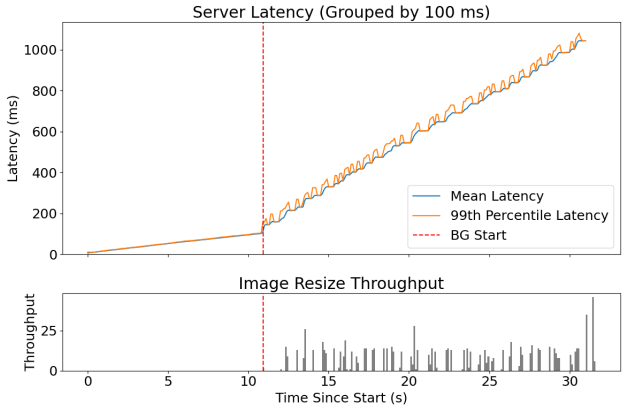

Figure 4: when running the server in real time, throttling degrades its performance at high load

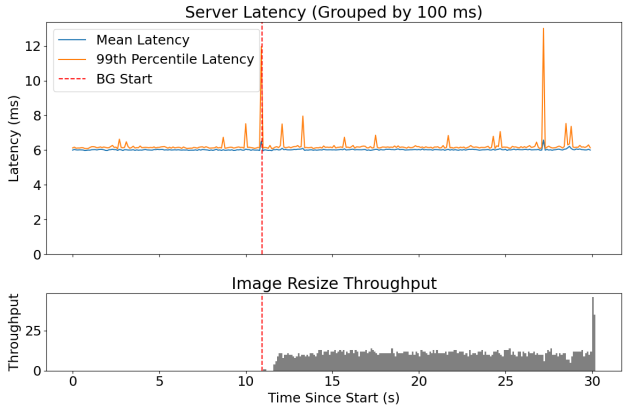

Figure 5: `BeClass` does a good job of isolating the server's latencies from the load from best effort jobs

3 Approach & Uniqueness

In order to enforce reservations while running BE jobs opportunistically, our approach uses priority scheduling instead of weights.

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 high class process. These checks ensure that a core c running a BE thread t knows that there are no queued LC threads anywhere: the *exit* check ensures theres none when starting to run t, and the *entry* check ensures if one wake up while running t it will interrupt t.

Linux already provides priority scheduling across scheduling classes, but it is designed for real time applications: if these experience high load, Linux throttles them in order to not starve the default class. We can see this happening in Figure 4, where throttling leads to spikes in the BE task's throughput, 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 CPUs they reserved. To enforce reservations under high load, without throttling the LCs or killing the BEs, `BeClass` *parks* BE processes, meaning the user-space code never runs, 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

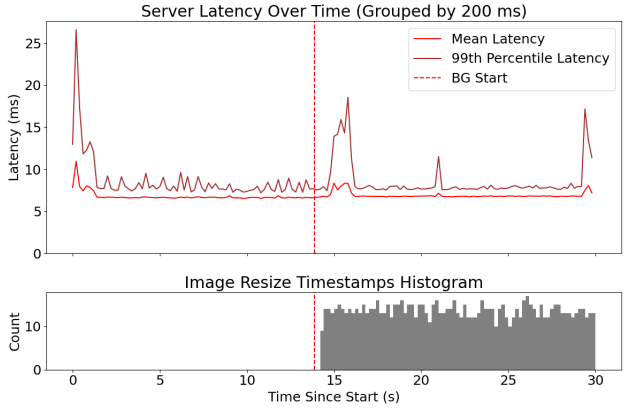

Figure 6: The same experiment as in Figure 1, but running the BE as a `BeClass` task

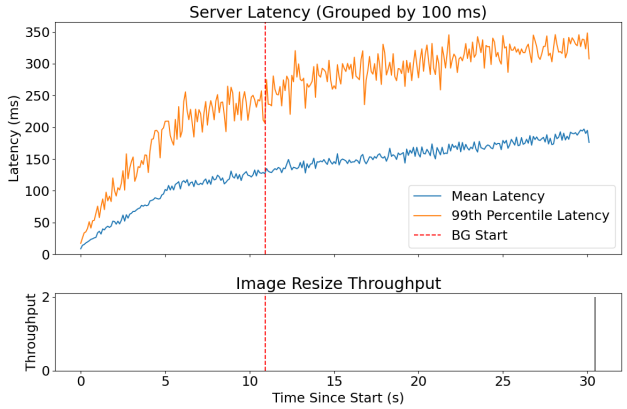

Figure 7: `BeClass` starves the BE user-space threads, leaving all CPU time to the LC

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, `BeClass` parks the best effort processes. Notice that the `BeClass` image resize job does not make progress until the very end, when the server is done processing the requests.

5 Contributions

This work identifies a serious limitation of `cgroups` weights, they are unable to effectively honor the reservations of LC applications in the presence BE workloads, and shows 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.