Mind the Gaps: Honoring LC reservations in the presence of BEs

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 [9, 12, 13], 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.

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

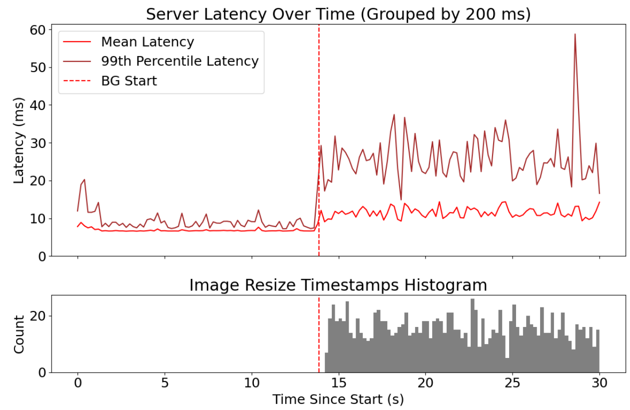

Figure 1: In Kubernetes, running a BE has latency impacts on an LC with reservations

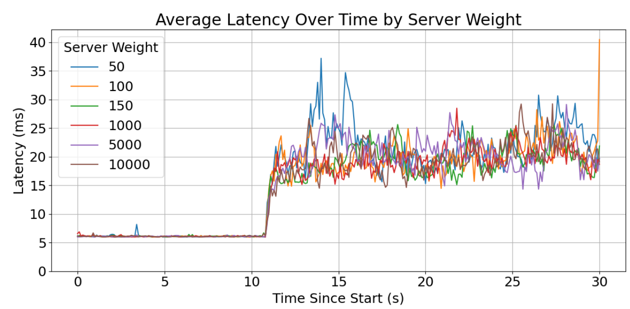

Figure 2: the weight of the server has little impact on how much the weight 1 BE task interferes

2 Background & Problem

All Open Container Initiative compliant containers, including Kubernetes, enforce CPU reservations using `cgroups`' weight interface [3, 10, 11], as do VM frameworks, including Firecracker and libvirt [2, 4, 5].

A simplified experiment shows `cgroups` weights are unable to enforce reservations. We run a simple CPU-bound server sharing 4 cores with an image resize job, each in their own group. `cgroups` supports weights in the range [1,10000]; Figure 2 shows running the server with any weight above 50 does not change the latency impact of the BE.

Visualizing the trace using schedviz [1] shows us this latency impact happens because for up to 10s of milliseconds, one core runs a BE thread (its own runqueue is otherwise empty), while server threads are queued on the other cores.

This happens because weights are only enforced within per-CPU runqueues. Load balancing eventually remedies imblances, but runs less frequently than scheduling. For workloads with request processing times in the millisecond range, waiting for the load balancer affects processing times.

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.

3 Related work

Linux's `sched_idle` policy addresses the observed weight inversion, but is inadequate: running the microbenchmark using `sched_idle` leads to a mean latency increase from 6ms to 8ms.

Other schedulers run primarily in userspace in order to work around the kernel scheduler [6–8]. This paper avoids doing so by identifying the core problem and addressing it.

4 Approach

Our approach replaces weights with priority scheduling.

Enforcing priorities requires fewer global runqueue searches than weights, because searches only need to happen on *class boundary crossings*: on *exit*, when switching to running a low-class process, and on *entry*, when enqueueing a high-class process. These checks ensure that a core running a BE thread t knows there are no queued LC threads anywhere: the *exit* check ensures there's none when starting to run t,

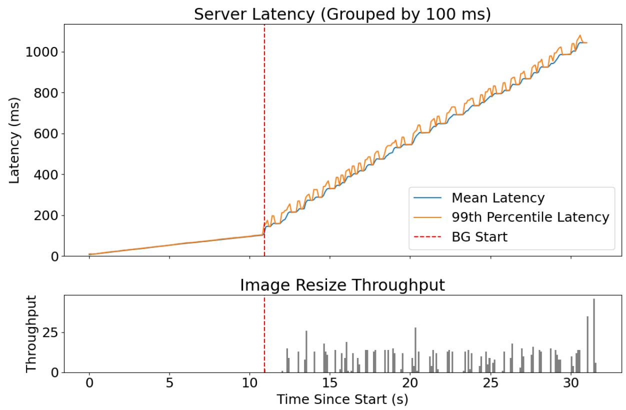

Figure 3: when running the server in real time, throttling degrades performance at high load

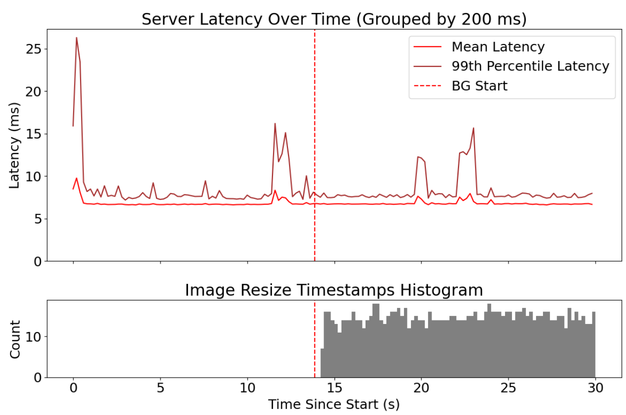

Figure 4: Kubernetes using `BeClass` honors reservations, unlike in Figure 1

and the *entry* check ensures if one wakes up while running t, it will interrupt t.

Linux enforces priority across scheduling classes, but higher class schedulers are designed for real-time applications. Additionally, Linux throttles them under high load in order to not starve the default class. We see this happening in Figure 3, where throttling leads to spikes in the BE's throughput and corresponding spikes in the server's latency.

We design a new scheduling class `BeClass` that sits at a lower priority than the default one, and enforces LCs' uncontended access to reserved CPUs. To enforce reservations under high load, without throttling LCs or killing BEs, `BeClass` uses *parking*, wherein user-space code never runs, only kernel-level services. We implement this strict priority in Linux.

5 Results

We re-run the Kubernetes application using `BeClass`. Figure 4 shows that the baseline mean latency of the LC server

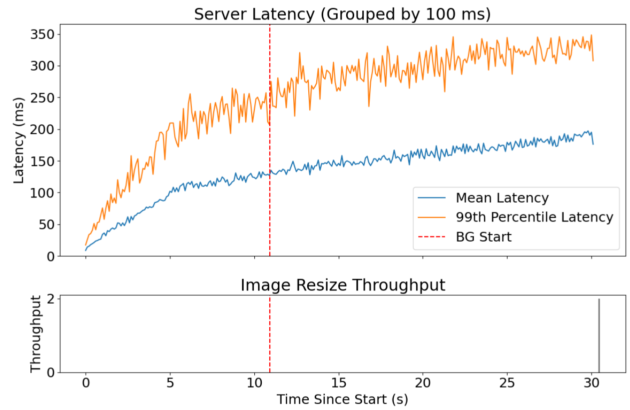

Figure 5: `BeClass` starves BE user-space threads, leaving all CPU-time to the LC

stays around 6.5ms after starting the the BE. Looking at the trace shows the BE runs only in gaps where cores would otherwise be idle.

Figure 5 shows parking enables the server to keep its reservation even under sustained 100% load. We run the same client load as Figure 3, with `BeClass` parking the BE. Notice that the BE does not make progress until the end, when the server is done processing.

6 Contributions

We identify a serious limitation of `cgroups` weights: they are unable to honor the reservations of LC applications in the presence BE workloads. We show that this is because Linux uses per-core runqueues.

We propose an API that separates BE from LC by introducing `BeClass`, and implement it in Linux. `BeClass` ensures that no BE is ever running when an LC is queued, and requires fewer cross-core interactions than a weight-based approach to do so.

References

[1] Google Open Source Blog. *Understanding Scheduling Behavior with SchedViz*. online. 2019.
[2] 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.
[3] Docker. *dockerdocs: Resource constraints*. online.
[4] libvirt Documentation. *CPU Allocation*. online.
[5] firecracker. *config-linux.md: Production Host Setup Recommendations*. github.

[6] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. "Caladan: Mitigating Interference at Microsecond Timescales". In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pages 281–297.

[7] Calin Iorgulescu, Reza Azimi, Youngjin Kwon, Sameh Elnikety, Manoj Syamala, Vivek Narasayya, Herodotos Herodotou, Paulo Tomita, Alex Chen, Jack Zhang, and Junhua Wang. "PerfIso: Performance Isolation for Commercial Latency-Sensitive Services". In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, July 2018, pages 519–532.

[8] Yuekai Jia, Kaifu Tian, Yuyang You, Yu Chen, and Kang Chen. "Skyloft: A General High-Efficient Scheduling Framework in User Space". In: *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*. SOSP '24. Austin, TX, USA: Association for Computing Machinery, 2024, pages 265–279.

[9] Will Kelly. *Overprovisioning in AWS? Cost-control tools and strategies can help*. online. 2013.

[10] Rory McClune. *How Do Containers Contain? Container Isolation Techniques*. online. 2021.

[11] opencontainers. *config-linux.md: Linux Container Configuration*. github.

[12] 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.

[13] 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.