Improving Performance with Metaschedulers

Brendan Doney
Virginia Tech

Abstract

Historically, Linux development has focused on the development of one general-purpose scheduler. However, programs can achieve better performance from schedulers that are tailored to their specific patterns and needs. Unfortunately, such schedulers may harm the performance of programs that do not exhibit their target behavior. As a result, it is difficult to achieve good performance by leveraging specialized schedulers on systems that co-locate several dissimilar applications or dissimilar subsystems of one application. To address this issue, we introduce the metascheduler, which manages what scheduler is running at any given time, to leverage the performance benefit from using specialized schedulers while minimizing the performance hit from poor scheduler-program fit. We design and implement four rudimentary metaschedulers and test them against an application comprised of a web sever and retrieval augmented generation service. We find that the best of our metaschedulers improves throughput by up to 3.64% and reduces 99% tail latency by up to 5.38% compared to EEVDF, also beating our results from the best single scheduler.

1 Introduction

Linux has traditionally favored the development of a single, general-purpose scheduler. From 2007 and until recently, the default general-purpose scheduler has been the well-studied Completely Fair Scheduler (CFS) [18]. Due to poor performance on latency-constrained applications, it was replaced in kernel version 6.6 onwards by the Earliest Eligible Virtual Deadline First (EEVDF) scheduler [21]. However, some researchers and companies have achieved better throughput and latency than these general-purpose schedulers with custom, specialized schedulers [9–12, 15, 17, 20].

Unfortunately, as with any form of specialization in the operating system, specialized scheduling algorithms have the potential to hurt the performance of applications that they do not target. As a result, specialized schedulers are not as fruitful on systems that co-locate many dissimilar applications or dissimilar subsystems of one application, such as systems running containerized applications or serverless functions. As a result, specialized scheduling algorithms are largely limited in their application to specific architectures that CFS and EEVDF were not designed for, such as heterogeneous high-performance computing systems [9, 10] or kernel-bypass systems [17, 20], or when a system will be used to run some narrowed set of workloads [12, 15].

While rudimentary schedulers may be implemented in kernel submodules, the vast majority of these specialized schedulers are implemented via kernel patches so that they can more tightly integrate with the kernel. Depending on the patch, a custom scheduler may or may not be compatible with other schedulers, including Linux's default scheduling policies. As a result, hosting multiple custom schedulers on the same system or making tweaks to them is a nontrivial task that will certainly require a kernel recompile and system reboot.

Recently, there has been a focus on making custom scheduler development more accessible via the `sched_ext` kernel patch [4]. This patch enables developers to write, load, and run schedulers via eBPF, the extended version of the Berkeley Packet Filter (BPF) [16] provided by the Linux kernel. While BPF itself is a safe runtime, `sched_ext` takes additional steps, like watchdog timeouts before reverting to the system's default scheduler, to prevent faulty scheduler code from breaking a system. Also, BPF schedulers can optionally integrate a user-space component by communicating via BPF maps to provide a fully user-space or hybrid scheduler.

While `sched_ext` primarily aims to improve developer experience, using BPF for schedulers also results in two other key features: the ability to dynamically load schedulers that affect the entire system and the seamless coexistence of multiple schedulers. First, because BPF programs can be loaded and unloaded dynamically and `sched_ext` places BPF schedulers at a higher priority than the system's default scheduler, schedulers can be swapped at runtime at the global scale. Second, due to the nature of BPF programs, the code for many BPF schedulers can co-exist on a system, although only one

can run at a time. Collectively, these key features enable the hot-swapping of the system's primary scheduler.

This paper aims to leverage this new hot-swapping capability to yield the performance improvement of a good scheduler-program fit, while minimizing the negative impact of poor scheduler-program fit, all on a system that is running several dissimilar programs. We do this by outlining and implementing a set of metaschedulers, which intelligently switch between schedulers, then test each metascheduler against an application comprised of a web server and retrieval augmented generation service. We argue that hot-swapping specialized schedulers can not only exceed the performance of a general-purpose scheduler, but also that of a single scheduler specialized for a full application which is comprised of many dissimilar components.

2 Design and Implementation

2.1 Schedulers

Because the focus of this work is how to best choose between a suite of specialized schedulers, rather than the design and implementation of performant schedulers, a set of pre-written schedulers from `sched_ext` [5] were used. Unless otherwise noted, they are each a single BPF program:

- `scx_central`: scheduling decisions are made from a single CPU.
- `scx_nest`: based on Inria-Paris' Nest scheduler [12], which uses core frequency to inform scheduling decisions.
- `scx_pair`: only co-locates tasks if they are in the same cgroup.
- `scx_qmap`: five-level FIFO queue scheduler, where tasks placed in higher-level queues get scheduled more often.
- `scx_simple`: FIFO queue scheduler that additionally weights timeslices by vtime.
- `scx_simple -f`: simple FIFO queue scheduler.
- `scx_rustland`: user-space Rust scheduler that schedules the lowest vruntime weighted by the number of voluntary context switches, to prioritize I/O intensive tasks.
- `scx_rusty` hybrid scheduler that does a round-robin of each domain in BPF while the user-space Rust component determines how to balance the domains.

Other available schedulers, namely `scx_userland` and `scx_flatcg`, were excluded because they were explicitly marked as not production ready and consistently stalled the system when run.

To inform the metaschedulers, we collect data on the performance and 99% tail latency of programs under each of these schedulers. For discussion of this data collection, see Evaluation.

2.2 Metaschedulers

For a proof of concept, four metaschedulers were implemented. As input, they are each provided with a base timeslice and the data collected on the throughput and 99% tail latency of each application under every scheduler. They then use some amount, or none, of the data to calculate a weight (as a percentage) for each application. The base timeslice is multiplied by this weight to derive the timeslice for each application's preferred scheduler, defined as the scheduler that gave the best throughput for that application. More succinctly, given a base timeslice t_b, the application i of n total applications, and the calculated weight for the application w_i, the timeslice for the application's preferred scheduler is calculated with $t_i = nt_b * w_i$

The four metaschedulers we implement are as follows:

1. **Round robin**: equally distributes fixed timeslices, using no data. This is overly simplistic and to be used as a baseline. So,
$$w_i = \frac{1}{n}$$
2. **Delta weighted**: uses latency data for each application i under EEVDF l_i to weight each timeslice by the estimated percent of time spent running its associated application. In effect, we prioritize the schedulers of applications that are throughput bottlenecks. So, for n total applications,
$$w_i = \frac{l_i}{\sum_{j=0}^{n} l_j}$$
3. **Benefit weighted**: uses the throughput r_i^{BEST} of each application i under its best scheduler to weight each timeslice by the amount of benefit its associated preferred scheduler yields compared to other schedulers. In effect, we prioritize the schedulers that yields the most performance boost to its application. So, for n total applications,
$$w_i = \frac{r_i^{BEST}}{\sum_{j=0}^{n} r_i^{BEST}}$$
4. **Delta-benefit weighted**: combines the logic of the delta and benefit weighted metaschedulers to prioritize schedulers that give the biggest performance boost for the most throughput-limiting applications. Formulas are omitted for brevity, but this strategy multiplies the delta weight and benefit weight, then normalizes it.

Schedulers are run via binaries compiled from the `sched_ext` project [5] in processes that the metascheduler

controls. To switch schedulers, the metascheduler kills the process for the old scheduler, and starts a new one for the next scheduler. While switching the BPF schedulers directly is likely more efficient, the `sched_ext` binaries are used because they load the BPF scheduler and simultaneously manage the user-space components of the hybrid schedulers `scx_rustland` and `scx_rusty`.

2.3 Base Timeslices

Because metaschedulers are a novel concept, there is no prior work that can inform our choice of base timeslice value. Furthermore, because each metascheduler calculates its weights very differently, we do not believe one universal timeslice value to be sufficient. Rather, we present the base timeslice value as a tunable hyperparameter unique to each metascheduler.

Observations about the overhead of switching between schedulers can provide a lower bound for the base timeslice tuning. In preliminary tests, we found that, on average, starting a scheduler took 102ms, stopping took 24ms, and a complete switch took 133ms. Given that Linux's default timeslice for `SCHED_RR` is 100ms and CFS/EEVDF timeslices will be much smaller, this switching overhead nontrivial. This would suggest that, for each scheduler to have a chance to provide any performance benefit, the metascheduler will have to stick with it for many of the chosen scheduler's timeslices.

3 Evaluation

In our evaluation, we focus on improving the performance of a single application decomposed into multiple, dissimilar subsystems. Note that running these subsystems is analogous to running multiple dissimilar applications. Using this scenario, we seek to answer the following questions:

1. How much of a difference is there between performance under the best scheduler for the full system and under the best schedulers for each application (§3.2)?

2. What are the optimal base timeslice or each metascheduler type (§3.3)?

3. Can switching between schedulers yield better performance than using the one best scheduler for the full system (§3.3)?

3.1 Experimental Setup

To evaluate the performance of our metaschedulers, we test against a web server that makes request to vector database as part of the retrieval augment generation process used commonly with large language models (LLMs) [13]. This is a component used in an LLM question and answer application that serves approximately 400 students in the CS 3214 Computer Systems course at Virginia Tech. This workload was chosen because it breaks cleanly into multiple subsystems, analogous to separate, dissimilar applications:

1. Frontend: HTTP Python/Flask API running under gunicorn [2] with 1 process and 4 threads. This is a primarily network constrained subsystem.

2. Backend: Calculates an embedding for a given query using the `multi-qa-mpnet-base-cos-v1` model, then finds four similar documents using the Chroma vector database [1] (containing 54,967 chunks with a maximum length of 500 characters each) using the cosine similarity metric. The subsystem is primarily I/O constrained, but is briefly CPU-constrained while computing the embedding before querying the vector database.

In testing, the "full application" refers to the combination of these two subsystems, where the frontend receives a POST request to the `/ask` endpoint, decodes and validates the JSON payload, performs some minor data transformations, then hands the transformed payload to the backend for it to find and return similar documents (so the frontend can respond with them). When testing the frontend in isolation, we substitute the call to the backend for a dummy function that always returns the same data. When testing the backend in isolation, we give it a fixed payload.

Experiments were run on a quad-core Intel i7-1068NG7 CPU @ 2.30GHz with HyperThreading enabled and 32 GB of total memory, running the sixth version of the SCX patches on Ubuntu 24.04 nightly. Benchmarking of the full application and frontend was performed using the wrk [6] HTTP benchmarking tool with 10 threads, 20 open connections, and a 30 second duration. In the benchmark, wrk uses a custom script to send a POST request to the `/ask` endpoint on the web server running on localhost. The backend is benchmarked by directly calling the Python code that computes the result of the `/ask` endpoint and measuring the time it takes to finish.

3.2 Scheduler Results

In order to inform the metaschedulers, we collect data on the throughput and latency of the full application and each subsystem under each scheduler. In a production setting, a CI/CD pipeline could easily collect these results for use at deploy time. Our results are shown in Figure 1. We can notice a couple of things from this data.

First, it is clear that the user-space scheduling strategy used in `scx_rustland` simply isn't performant enough for use on production systems. This is no surprise, given the overhead associated with delegating scheduling decisions to user-space via eBPF maps and switching to the user-space Rust component. However, note that this can be made much more efficient as in ghost user-space schedulers [11] and as `sched_ext` develops.

Second, it is obvious that there is some performance to be gained over EEVDF by simply choosing a better scheduler for the full application's workload. In our data, several schedulers yield better throughput or latency than EEVDF even on the full application. In fact, `scx_nest` yields the best results for throughput and 99% tail latency for the full application when compared to EEVDF, with a 2.36% improvement and 4.17% reduction, respectively.

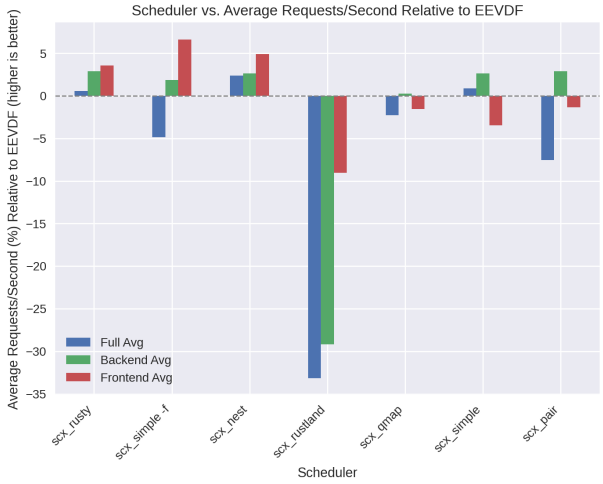
But, more importantly, we can see that several schedulers also perform much better for individual subsystems than `scx_nest`. For the frontend, `scx_simple -f` gave the best results for throughput and 99% tail latency, with a 6.62% improvement and 15.32% reduction, respectively. For the backend, `scx_rusty` and `scx_pair` both gave a 2.91% improvement to throughput and `scx_simple` gave a 20.38% reduction to 99% tail latency. Given that these subsystem-specific results are all higher than the best full application results, this furthers our hypothesis that switching between schedulers could improve performance over using one scheduler.

Lastly, our results confirm that some schedulers can benefit some workloads, while harming others. For example, while the backend sees improved throughput under `scx_pair`, the frontend actually sees a 1.33% reduction. Similarly, while `scx_simple` yielded the best 99% tail latency for the backend, the frontend sees a 110.35% increase when using it. Therefore, how to switch schedulers while minimizing poor scheduler-program fit is an open problem that must be solved to achieve the best performance.

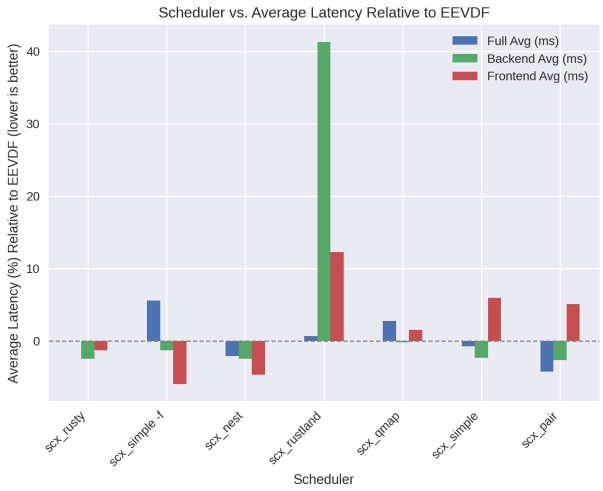
3.3 Metascheduler Results

Using the data collected on the performance of each subsystem, we tune the base timeslice value according to throughput and present the results. Note that, based on the throughput data, we fix the preferred scheduler for the frontend as `scx_simple -f` and for the backend as `scx_rusty`. These are the schedulers that each metascheduler will switch between in tests. Our results are shown in Figure 2. The results of choosing schedulers based on the best 99% tail latency values or tuning the timeslice values according to 99% tail latency is left as an open question.

In both throughput and 99% tail latency, there is an obvious extrema that marks the best combination of metascheduler and base timeslice value. For both metrics, this is the delta-weighted metascheduler with base timeslice value of 1800ms, yielding a throughput improvement of 3.94% and a 99% tail latency reduction of 5.38% compared to EEVDF. Notably, these results are also better than those of the best scheduler for the full application, `scx_nest`, which only yielded a throughput improvement of 2.36% and a 99% tail latency reduction of 4.17%. These results show that switching between specialized schedulers can yield better performance than using one general-purpose scheduler or even one scheduler specialized



(a) Throughput



(b) 99% Tail Latency

Figure 1: Throughput and latency of each subsystem measured under each scheduler as a percentage relative to EEVDF.

for the full system. Additionally, the reduction in 99% tail latency under the delta-weighted metascheduler is notable, given that we did not explicitly optimize for tail latency in our choice of preferred schedulers or in our tuning process. This outcome shows the potential for further reducing application latency through metaschedulers.

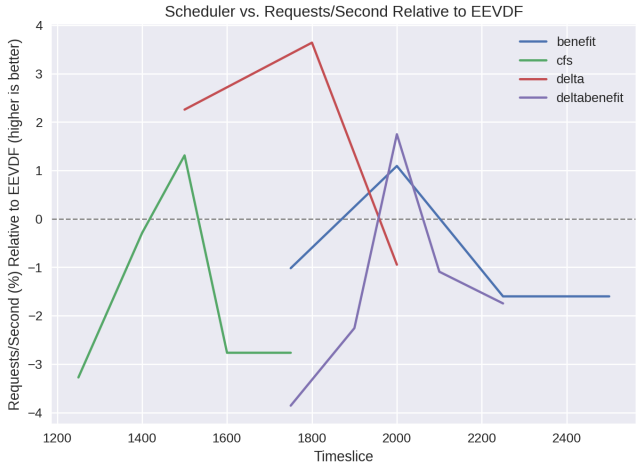
There are two other things to notice from the graphs. First, the optimal values for throughput all align with the optimal values for 99% tail latency. Further study is needed to know if this is universal or specific to the combination of workload, preferred schedulers, and metaschedulers that we tested. Second, the optimal base timeslice values vary by metascheduler, but are all well above the 133ms switching overhead observed earlier. This agrees with our earlier suggestion that we must stick with one scheduler for many timeslices, before switching, if we are to see any benefit to the performance of the full application.

4 Related Work

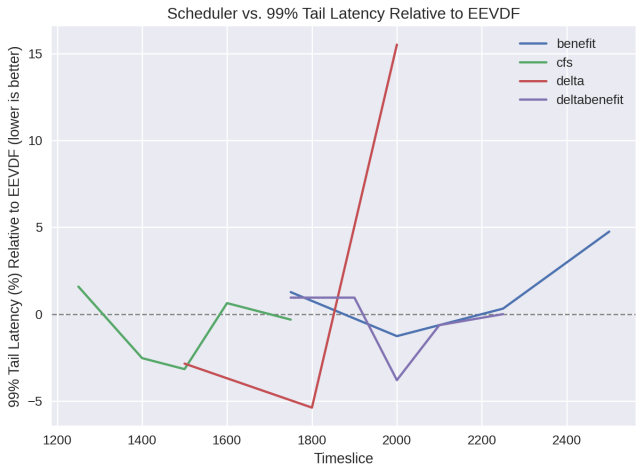
Ghost [11] is a framework for delegating scheduling decisions to user-space. It uses message-passing to facilitate communication between the kernel and the user-space threads that make scheduling decisions. Such a framework is similar to the `sched_ext` patch [4] in concept, but we use `sched_ext` because of its potential to be mainlined into the kernel and because it allows schedulers to choose whether to run entirely in kernel-space via BPF or with a combination of BPF and user-space code. As an alternative to both, applications can be represented as user-level threads running against a custom user-level scheduler [14]. However, this approach requires special integration into applications to function, complicated by a lack of kernel support and standard APIs [8].

Our work is partly inspired by schedulers that target highly customizable kernel-bypass environments. Shenango [17] and ZygOS [20] are two such schedulers, with Shenango allocating cores to applications based on estimated benefit and ZygOS implementing a work-consering scheduler to keep cores constantly busy.

There are similarities between metaschedulers and virtual machine placement (VMP). It is well-known that poor VMP can significantly degrade application performance [19], but migrations impose an overhead [7]. Similarly, with metaschedulers, it is known that poor scheduler-program fit can degrade performance, but that there is an overhead with switching schedulers. To improve VMP, Xu and Fortes [22] use a genetic algorithm, while Anand et. al [7] integrate a quantified migration overhead into the existing ILP and First Fit Decreasing algorithms. Zhao et. al [23] model VMP as a bi-objective optimization problem and use ant colony optimization to solve it. However, the difference between VMP and metaschedulers is that VMP is concerned with running potentially dissimilar applications on clusters of physical machines, rather than co-located on one machine. Similarly, for cloud workloads,



(a) Throughput



(b) 99% Tail Latency

Figure 2: Throughput and latency results from the tuning process for the implemented metaschedulers. Note that we optimize for throughput.

Kubernetes [3] allows users to tune a default scheduler or define custom schedulers for optimal container placement on a cluster, but does not focus on improving the performance of containers on nodes which host a variety of dissimilar containers as we do.

5 Conclusions and Future Work

Switching between specialized schedulers has the potential to improve application performance over using one general-purpose scheduler. Using one target application with two distinct subsystems, we have presented the design and implementation of several rudimentary metaschedulers, which already yield better throughput and 99% tail latency than EEVDF and the best scheduler for the full application. However, there is still much work to be done in this area. Most importantly, more workloads need to be tested to find how much performance benefit was specific to the application we chose. Additionally, we do not know the number of application and scheduler combinations we can run with before this approach breaks down. To improve the performance of metaschedulers at large, there is still much work to be done in reducing the overhead of switching schedulers. Running metaschedulers as BPF programs, which choose what BPF scheduler function to call while running, could provide such switching reductions and improve the overall performance of the metascheduler by running in the kernel. Finally, the design of custom BPF schedulers and metaschedulers are new and open problems. In particular, more complex metaschedulers that utilize more information (e.g. run queue information, system topology, recent application activity, and the costs of compromising between multiple applications), are likely to give results that exceed those of our metaschedulers.

References

[1] Chroma. https://www.trychroma.com/. Accessed: 2024-04-23.

[2] Gunicorn. https://github.com/benoitc/gunicorn. Accessed: 2024-04-25.

[3] Kubernetes scheduler. https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/. Accessed: 2024-04-23.

[4] sched_ext kernel development repository. https://github.com/sched-ext/sched_ext. Accessed: 2024-04-25.

[5] sched_ext schedulers and tools. https://github.com/sched-ext/scx. Accessed: 2024-04-25.

[6] wrk - a http benchmarking tool. https://github.com/wg/wrk. Accessed: 2024-04-25.

[7] ANAND, A., LAKSHMI, J., AND NANDY, S. Virtual machine placement optimization supporting performance slas. In *2013 IEEE 5th International Conference on Cloud Computing Technology and Science* (2013), vol. 1, IEEE, pp. 298–305.

[8] ANDERSON, T. E., BERSHAD, B. N., LAZOWSKA, E. D., AND LEVY, H. M. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems (TOCS) 10*, 1 (1992), 53–79.

[9] DAOUD, M. I., AND KHARMA, N. A high performance algorithm for static task scheduling in heterogeneous distributed computing systems. *Journal of Parallel and distributed computing 68*, 4 (2008), 399–409.

[10] HAGRAS, T., AND JANECEK, J. A high performance, low complexity algorithm for compile-time task scheduling in heterogeneous systems. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.* (2004), IEEE, p. 107.

[11] HUMPHRIES, J. T., NATU, N., CHAUGULE, A., WEISSE, O., RHODEN, B., DON, J., RIZZO, L., ROMBAKH, O., TURNER, P., AND KOZYRAKIS, C. ghost: Fast & flexible user-space delegation of linux scheduling. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (2021), pp. 588–604.

[12] LAWALL, J., CHHAYA-SHAILESH, H., LOZI, J.-P., LEPERS, B., ZWAENEPOEL, W., AND MULLER, G. Os scheduling with nest: keeping tasks close together on warm cores. In *Proceedings of the Seventeenth European Conference on Computer Systems* (2022), pp. 368–383.

[13] LEWIS, P., PEREZ, E., PIKTUS, A., PETRONI, F., KARPUKHIN, V., GOYAL, N., KÜTTLER, H., LEWIS, M., YIH, W.-T., ROCKTÄSCHEL, T., ET AL. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems 33* (2020), 9459–9474.

[14] MARSH, B. D., SCOTT, M. L., LEBLANC, T. J., AND MARKATOS, E. P. First-class user-level threads. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles* (1991), pp. 110–121.

[15] MARTY, M., DE KRUIJF, M., ADRIAENS, J., ALFELD, C., BAUER, S., CONTAVALLI, C., DALTON, M., DUKKIPATI, N., EVANS, W. C., GRIBBLE, S., ET AL. Snap: A microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (2019), pp. 399–413.

[16] MCCANNE, S., AND JACOBSON, V. The bsd packet filter: A new architecture for user-level packet capture. In *USENIX winter* (1993), vol. 46, pp. 259–270.

[17] 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.

[18] PABLA, C. S. Completely fair scheduler. *Linux Journal 2009*, 184 (2009), 4.

[19] PAUL, I., YALAMANCHILI, S., AND JOHN, L. K. Performance impact of virtual machine placement in a datacenter. In *2012 IEEE 31st International Performance Computing and Communications Conference (IPCCC)* (2012), IEEE, pp. 424–431.

[20] PREKAS, G., KOGIAS, M., AND BUGNION, E. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), pp. 325–341.

[21] STOICA, I., AND ABDEL-WAHAB, H. Earliest eligible virtual deadline first: A flexible and accurate mechanism for proportional share resource allocation. *Old Dominion Univ., Norfolk, VA, Tech. Rep. TR-95-22* (1995).

[22] XU, J., AND FORTES, J. A. Multi-objective virtual machine placement in virtualized data center environments. In *2010 IEEE/ACM int'l conference on green computing and communications & int'l conference on cyber, physical and social computing* (2010), IEEE, pp. 179–188.

[23] ZHAO, H., WANG, J., LIU, F., WANG, Q., ZHANG, W., AND ZHENG, Q. Power-aware and performance-guaranteed virtual machine placement in the cloud. *IEEE Transactions on Parallel and Distributed Systems 29*, 6 (2018), 1385–1400.