Dynamic Dispatcher Assignment With Flat-Combining

Gangmin Lee
KAIST
Republic of Korea
lgm9@kaist.ac.kr

Wonsup Yoon
KAIST
Republic of Korea
wsyoon@kaist.ac.kr

Sue Moon
KAIST
Republic of Korea
sbmoon@kaist.ac.kr

1 Introduction

Modern network servers must deliver low tail latency to meet service level objectives (SLOs). In datacenters, these SLOs often fall within microseconds, posing challenges for existing operating systems. Consequently, redesigning key components of the operating system becomes essential.

To meet SLOs, various efficient scheduling designs have focused on reducing tail latency. ZygOS highlights the importance of microsecond-scale scheduler design in networked systems, modeling tail latency trends across different queuing models and introducing a work-stealing-based scheduler to improve microsecond-level tail latency [8]. Shinjuku demonstrates the benefits of preemptive scheduling in microsecond-scale scheduling and introduces a low-overhead thread preemption technique using hardware instructions [6]. Concord further enhances preemptive scheduling by leveraging compiler techniques to reduce preemption overhead [5]. Additionally, Persephone incorporates application-level information into scheduling algorithms [1], Shenango improves CPU efficiency through fast core reallocation [7], and Caladan introduces scheduling mechanisms that mitigate resource contention in CPU cache and memory bandwidth [2].

Centralized queueing is one of the most widely used designs for microsecond-scale scheduling, adopted by systems like Shinjuku, Concord, and Persephone. It employs a pinned dispatcher thread that receives network requests and distributes them to worker threads, and the worker threads process the requests. By utilizing a single queue, this design eliminates load imbalance across workers and lock contention, ensuring efficient request handling through lock-free message-passing.

Centralized queueing achieves low tail latency but requires a dedicated dispatcher thread that continuously polls the network queue and monitors worker threads. This dispatcher occupies an entire CPU core, which leads to inefficient CPU utilization. To address this problem, Concord introduces a work-conserving dispatcher that preempts running applications to handle dispatching when necessary.

To address CPU inefficiency, we adopt flat-combining [3], a synchronization mechanism in which a lock holder (combiner) processes requests from other threads, reducing lock contention and improving concurrency. Applying this to centralized queueing, we replace the dedicated dispatcher with a dynamically assigned worker thread that assumes the role when needed.

To demonstrate the potential advantages of our design, we conduct an experiment with our preliminary implementation. In the experiment, our design achieves 84% lower tail latency and 14% higher throughput than the baselines in a RocksDB key-value store workload.

2 Preliminary Design and Implementation

The core of our design is dynamically assigning a dispatcher among worker threads. Our preliminary assignment process follows these steps:

1. When a worker thread is idle, it attempts to acquire a global lock for a centralized queue. At startup, all threads are idle.
2. Among idle threads, only one thread holds the global lock and becomes a dispatcher, while the others remain worker threads.
3. The dispatcher thread distributes requests to all workers (including itself).
4. The dispatcher thread releases the global lock and resumes execution as a worker thread.
5. The worker threads process dispatched requests and repeat from the first step on completion.

We implemented this design in a networked request-handling system. The system is based on the Linux UDP networking stack and written in C++.

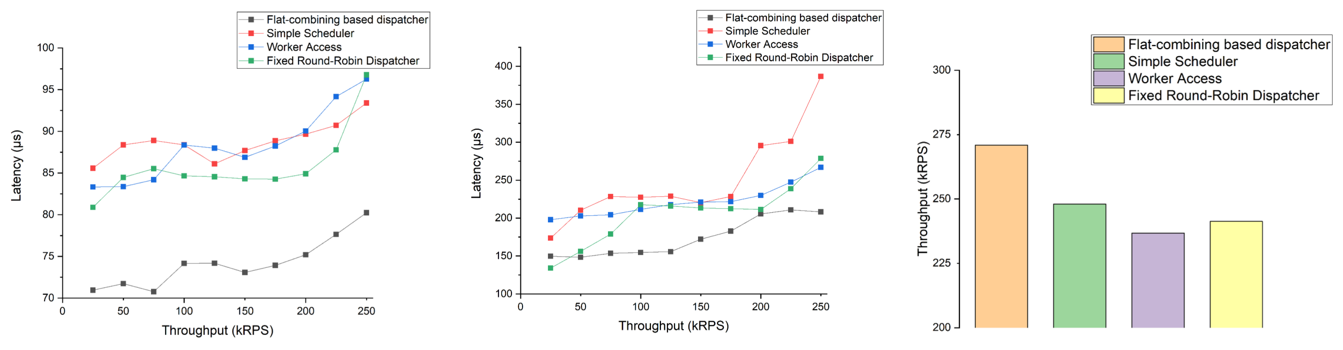
3 Evaluation

To evaluate the potential performance gains of our design, we compare it against three baselines: Simple Scheduler, Worker Access, and Fixed Round-Robin Dispatcher.

- **Simple Scheduler:** Requests are evenly distributed to worker queues. Since each worker has its own queue, no synchronization overhead occurs.
- **Worker Access:** All received requests are stored in a shared global queue, and workers directly access the queue. A global lock manages concurrent access to the queue.
- **Fixed Round-Robin Dispatcher:** A dedicated dispatcher thread distributes requests to workers. This design follows the centralized queueing in previous studies [1, 5, 6].

We also implement a load generator that simulates multiple clients to load our implementation and the baseline. This load generator and the baselines are also implemented using the C++ and Linux UDP networking stack.

For the workload, we use RocksDB [4], a widely used key-value store in datacenters. At experiment startup, we



(a) Median latency **(b)** P99 latency **(c)** Achievable throughput

Figure 1. Performance comparison between our design and baselines.

populate the database with key-value pairs (randomly generated 20-character strings) and measure the GET query performance using the load generator. The key performance metrics for the evaluation include median latency, P99 latency, and achievable throughput.

Figure 1 shows the results. Our design achieves lower median and tail latency compared to baselines overall. This result is mainly due to a reduced load imbalance across workers compared to Simple Scheduler, reduced lock contention than Worker Access, and a higher number of workers than Fixed Round-Robin Dispatcher. Ours also outperforms baselines for achievable throughput and delivers up to 84% lower tail latency and 14% higher throughput. However, at low throughput levels, the Fixed Round-Robin Dispatcher achieves better tail latency, as the number of worker threads becomes less significant in this scenario due to the low throughput.

4 Conclusion and Future Work

This paper explores a new scheduler design inspired by flat-combining and its potential performance benefits. It improves centralized queueing by dynamically assigning a dispatcher among worker threads if necessary. Using a flat-combining-based dispatcher assignment, the scheduler reduces synchronization overhead in the assignment process. In the evaluation of our preliminary design and implementation, our approach improves P99 latency by up to 84% and increases throughput by 14% compared to baselines in a RocksDB workload.

Future work includes optimizing dispatcher assignment strategies for NUMA architectures, developing a complete system with a user-space networking stack, extensively evaluating the impact of various network protocols and data structures, and integrating this design with state-of-the-art systems such as Shinjuku, Concord, and Persephone [1, 5, 6].

Acknowledgement

We thank anonymous reviewers for helpful comments. This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No.RS-2022-NR070293).

References

[1] Henri Maxime Demoulin, Joshua Fried, Isaac Pedisich, Marios Kogias, Boon Thau Loo, Linh Thi Xuan Phan, and Irene Zhang. 2021. When Idling is Ideal: Optimizing Tail-Latency for Heavy-Tailed Datacenter Workloads with PersēPhone. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 621–637. https://doi.org/10.1145/3477132.3483571

[2] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. 2020. Caladan: Mitigating Interference at Microsecond Timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 281–297. https://www.usenix.org/conference/osdi20/presentation/fried

[3] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. 2010. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the Twenty-Second Annual ACM Symposium on Parallelism in Algorithms and Architectures* (Thira, Santorini, Greece) *(SPAA '10)*. Association for Computing Machinery, New York, NY, USA, 355–364. https://doi.org/10.1145/1810479.1810540

[4] Meta Platforms Inc. 2022. RocksDB. https://rocksdb.org.

[5] Rishabh Iyer, Musa Unal, Marios Kogias, and George Candea. 2023. Achieving Microsecond-Scale Tail Latency Efficiently with Approximate Optimal Scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 466–481. https://doi.org/10.1145/3600006.3613136

[6] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. 2019. Shinjuku: Preemptive Scheduling for μsecond-scale Tail Latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 345–360. https://www.usenix.org/conference/nsdi19/presentation/kaffes

[7] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 361–378. https://www.usenix.org/conference/nsdi19/presentation/ousterhout

[8] George Prekas, Marios Kogias, and Edouard Bugnion. 2017. ZygOS: Achieving Low Tail Latency for Microsecond-Scale Networked Tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 325–341. https://doi.org/10.1145/3132747.3132780