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, however, SLOs are in a few or tens of microseconds, which are often challenging for existing operating systems. Consequently, datacenters require the redesign of many parts of operating systems.

To meet the SLOs, many efficient scheduling designs have addressed lower tail latency. ZygOS emphasizes the importance of microsecond-scale scheduler design in networked systems [8]. It theoretically models the trends in tail latency under different queuing models and designs a work-stealing-based microsecond-scale scheduler to improve microsecond-level tail latency. Shinjuku demonstrates the effectiveness of preemptive scheduling in microsecond-scale scheduling and introduces a low-overhead thread preemption technique utilizing hardware instructions [6]. To enhance the preemptive scheduling further, Concord leveraged compiler techniques to significantly reduce the overhead associated with preemption [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].

One of the most widely used designs for microsecond-scale scheduling is centralized queueing. This design is employed in many systems, such as Shinjuku, Concord, and Persephone. It uses a pinned dispatcher thread that receives requests from the network and distributes them to worker threads, and the worker threads process the requests. In this design, there is no load imbalance across workers and no lock contention because there is only one queue in a dispatcher, and it delivers requests to worker threads through lock-free message-passing.

Centralized queueing offers low tail latency; however, it requires a dispatcher thread to occupy an entire CPU core. The pinned thread keeps running to poll a network queue and worker threads' states continuously, wasting CPU cycles overall. To address this problem, Concord introduces a work-conserving dispatcher, which also runs applications and preempts them to run dispatching code.

In this work, to address the CPU wasting problem, we explore a different approach inspired by flat-combining [3]. Flat-combining is a synchronization mechanism where a lock holder (combiner) receives and processes requests from the other threads. Compared with a mechanism where multiple threads compete for a lock, this mechanism reduces lock contentions, improving the performance of concurrent data structures. We design a similar mechanism for centralized queueing. Rather than assigning a dedicated thread for dispatching, one of the worker threads becomes a dispatcher (or combiner) thread if necessary.

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

2 Preliminary Design and Implementation

The core of our design is assigning a dispatcher among worker threads. Our preliminary design for the assignment process is as follows:

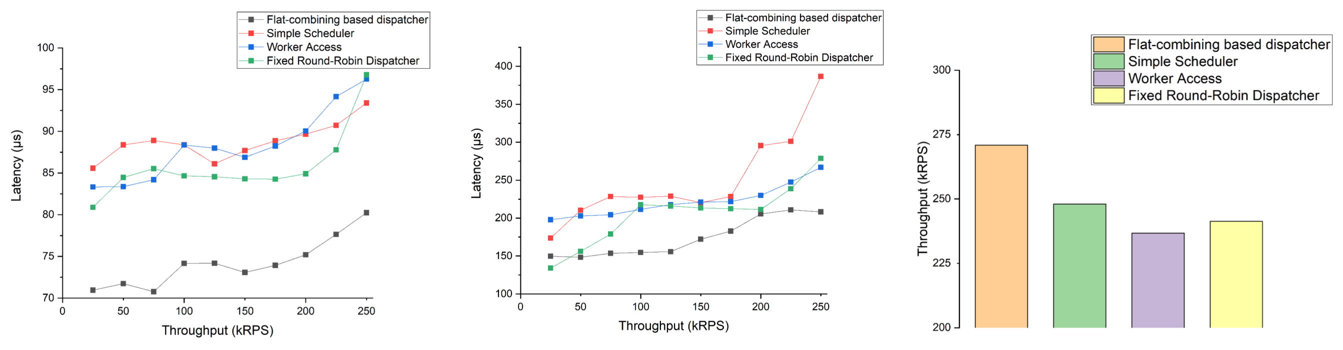
1. When a worker thread is idle, it tries to acquire a global lock for a centralized queue. At startup, all threads are idle.
2. Among idle threads, only one thread holds the global lock and becomes a dispatcher, while the other threads remain workers.
3. The dispatcher thread distributes requests to 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 have implemented a networked request-handling system with our design. The system is based on the Linux UDP networking stack and written in C++.

3 Evaluation

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

- **Simple Scheduler:** Received 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 mediates concurrent access to the global queue.



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

Figure 1. Performance comparison between our design and baselines.

- **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 a number of clients to load our implementation and the baseline. This load generator and the baselines are implemented using the C++ and Linux UDP networking stack.

For the workload, we use RocksDB [4], a popular key-value store in datacenters. At experiment startup, we populate key-value pairs (randomly generated 20-character strings), and we measure the performance of GET queries using the load generator. Key performance metrics for the evaluation are median latency, P99 latency, and achievable throughput.

Figure 1 shows the results. Ours achieves lower median and tail latency compared to baselines overall. This result is mainly due to a lower load imbalance across workers than 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. Overall, ours delivers up to 84% better tail latency and 14% higher throughput than baselines.

4 Conclusion and Future Work

In this paper, we explore a new scheduler design inspired by flat-combining and its potential performance gain. 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 on the assignment process. In the evaluation of our preliminary design and implementation, ours shows up to 84% better P99 latency and 14% higher throughput than baselines in a RocksDB workload.

Future work includes optimizing dispatcher assignment strategies for NUMA architectures, designing a complete system with a user-space networking stack, and extensively evaluating the impact of different network protocols and data structures.

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 PerséPhone. 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