Achieving Microsecond-Scale Tail Latency Efficiently with Approximate Optimal Scheduling

Rishabh Iyer[1], Musa Unal[1], Marios Kogias[2], George Candea[1]
[1] EPFL, Switzerland [2] Imperial College London

Abstract

Datacenter applications expect microsecond-scale service times and tightly bound tail latency, with future workloads expected to be even more demanding. To address this challenge, state-of-the-art runtimes employ theoretically optimal scheduling policies, namely a single request queue and strict preemption.

We present Concord, a runtime that demonstrates how forgoing this design—while still closely approximating it—enables a significant improvement in application throughput while maintaining tight tail-latency SLOs. We evaluate Concord on microbenchmarks and Google's LevelDB key-value store; compared to the state of the art, Concord improves application throughput by up to 52% on microbenchmarks and by up to 83% on LevelDB, while meeting the same tail-latency SLOs. Unlike the state of the art, Concord is application agnostic and does not rely on the non-standard use of hardware, which makes it immediately deployable in the public cloud. Concord is publicly available at https://dslab.epfl.ch/research/concord.

ACM Reference Format:
Rishabh Iyer, Musa Unal, Marios Kogias, George Candea. 2023. Achieving Microsecond-Scale Tail Latency Efficiently with Approximate Optimal Scheduling . In *ACM SIGOPS 29th Symposium on Operating Systems Principles (SOSP '23), October 23–26, 2023, Koblenz, Germany.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3600006.3613136

1 Introduction

Datacenter applications for web search, e-commerce, social networking, etc. have strict microsecond-scale tail latency service level objectives (SLOs) [7, 36, 43, 65]. Since every user request is fanned out across thousands of servers with the end-to-end response time being determined by the slowest response, bounding tail latency at each server is critical [18]. These strict tail-latency SLOs at individual servers are only expected to get tighter over time [30, 37] as applications are modularized into increasingly finer microservices [23, 41, 44, 56] and communication stacks are offloaded to specialized hardware [5, 34, 38, 55, 64].

Since the tail latency of a request at individual servers is dominated by its queueing delay (and not service time), state-of-the-art schedulers are optimized based on queueing theory results. Weirman and Zwart [61] show that there is no single scheduling policy that minimizes tail latency across all possible workloads; the First Come, First Served (FCFS) policy is optimal for light-tailed workloads, and Processor Sharing (PS) is optimal for heavy-tailed workloads [26]. Additionally, single-queue scheduling improves tail latency when compared to multi-queue scheduling for both FCFS and PS policies. Since both light- and heavy-tailed workloads are common in production [19, 36], state-of-the-art microsecond-scale schedulers need to support both (1) preemptive scheduling to implement PS for heavy-tailed workloads, and (2) a single queue.

However, optimizing systems for tail latency inevitably sacrifices the maximum throughput they can sustain, with the sacrificed throughput only increasing as request service times grow shorter. For example, single-queue systems such as ZygOS [50] and Shinjuku [36] achieve lower maximum throughput than IX [9], an earlier system that had no tail-latency optimizations. Similarly, preemptive scheduling in Shinjuku imposes a 20% throughput penalty at a scheduling quantum of 5μs, and a 50% throughput penalty at a quantum of 2μs. Fig. 1 conceptualizes the trade-off faced by tail-optimized microsecond-scale systems: chasing tight bounds on tail latency makes such systems move from the blue to the orange curve, which results in them saturating sooner than their non-tail-optimized counterparts.

Systems optimized for tail latency also frequently sacrifice deployability and generic support for applications. For example, Shinjuku relies on non-standard use of virtualization hardware to achieve microsecond-scale preemption, but this precludes its deployment on VMs in public clouds. Similarly, Persephone [19], another state-of-the-art system, relies on non-blind scheduling, i.e., it requires prior knowledge of the application's service time distribution and is restricted to applications with request classes that have disjoint service-time distributions known a priori. However, this makes it ill-suited for the datacenter where blind policies are required to deal with heterogeneous applications [28].

SOSP '23, October 23–26, 2023, Koblenz, Germany
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0229-7/23/10…$15.00
https://doi.org/10.1145/3600006.3613136

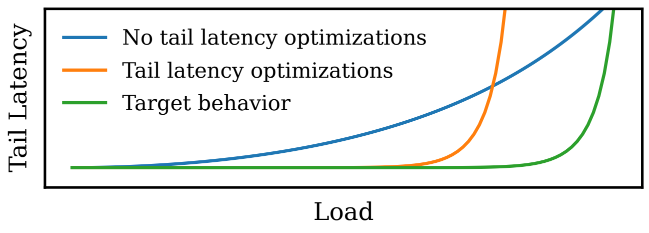


Figure 1. Abstract, comparative visualization of throughput overhead in state-of-the-art datacenter systems.

This paper describes Concord, a scheduling runtime for microsecond-scale applications that presents an improved trade-off between tail latency and throughput without sacrificing deployability or generality. Concord's key insight is that careful *approximation* (as opposed to canonical implementation) of optimal scheduling policies enables new microsecond-scale mechanisms that provide significant throughput gains at negligible tail-latency costs. Thus, Concord does not introduce new scheduling *policies*. Instead, it introduces new scheduling *mechanisms* that efficiently approximate existing policies to increase application throughput while maintaining similar tail latency. This enables existing systems optimized for tail latency (orange curve in Fig. 1) get closer to their ideal behavior (green curve).

Concord relies on three mechanisms to efficiently approximate the theoretically-optimal single queue and precise preemption. First, Concord's compiler-enforced cooperation approximates precise preemption using asynchronous communication between threads. While this asynchrony leads to slightly imprecise scheduling quanta, it enables Concord to eschew interrupts and reduce the preemption overhead by 4× without significantly impacting tail latency. Second, Concord leverages Join-Bounded Shortest Queue (JBSQ) scheduling [40] and approximates a single queue with bounded core-local queues. This design eliminates cache-coherence-induced stalls in worker threads and reduces the throughput overhead of single queue scheduling by 9−13× without significantly impacting tail latency. Finally, the Concord dispatcher —a thread normally dedicated to enqueueing requests—is work-conserving and runs application logic when all worker threads are busy. While this work conservation approximates both a single queue and precise preemption, it enables the dispatcher to contribute to application throughput, which is not the case in existing systems. All three of Concord's mechanisms eschew reliance on non-standard use of hardware or application-level assumptions, making Concord immediately deployable in the public cloud.

To evaluate Concord, we compare it to Shinjuku [36] and Persephone [19] across several service time distributions from both academic and industrial sources. We observe that Concord not only outperforms these systems at the scheduling quanta they were designed for (5 – 15µs), but provides even larger improvements at the smaller timescales which will likely be required in the future [30, 37]. For the synthetic workloads used by Shinjuku and Persephone, Concord sustains up to 52% greater application throughput for a scheduling quantum of 2µs, and up to 18% greater throughput for a quantum of 5µs, while meeting identical tail-latency SLOs. For Google's LevelDB, Concord sustains up to 83% greater throughput for a 2µs quantum and up to 52% greater throughput for a 5µs quantum, while once again meeting identical tail-latency SLOs. Finally, we demonstrate that Concord's mechanisms will remain useful even as datacenter hardware evolves to provide increased support for microsecond-scale scheduling by showing how Concord enables preemptive scheduling at 2× lower overhead than Intel's recently launched user-space interrupts [62].

In the rest of the paper, we first perform a quantitative analysis of the throughput overheads in existing systems optimized for tail latency (§2), before using the results of the analysis to design (§3) and implement (§4) Concord. We then evaluate Concord and demonstrate its throughput benefits (§5), discuss its limitations and broader applicability (§6), present related work (§7), and conclude (§8).

2 Throughput Overheads at µs Scale

In this section, we analyze the throughput overheads that arise from implementing preemptive and single queue scheduling at microsecond scale.

Schedulers that implement a single queue fall in two categories: those that maintain a *physical* and *logical* single queue, respectively. In the former [19, 36], one thread (the dispatcher) is dedicated to maintaining the queue and sending requests to the others, while in the latter [46, 50] there is no dedicated thread and idle threads *steal* requests from other threads to mitigate load imbalance. In this section, we focus on single *physical* queue systems. We do so for two reasons: (1) the only prior work [36] that implements both a single queue and preemption—and is thus the most relevant baseline—employs such a queue, and (2) having a dedicated thread with global visibility of the entire system allows provides the flexibility to implement arbitrary queueing policies. We defer a detailed discussion of systems that maintain a single *logical* queue to §6.

We use an analytical model to describe the sources of throughput overhead in single physical queue systems. Our model does not focus on particular prior implementations; instead, we reason abstractly about the trade-offs of multiple implementations. We first introduce our system model (§2.1), and then use it to show how existing systems suffer from double-digit overheads at today's 5µs timescales and triple-digit overheads at tomorrow's 1µs timescales (§2.2).

2.1 System Model

We consider a system with 1 dedicated dispatcher thread and *n* worker threads, all of which are pinned to individual CPU cores. The dedicated dispatcher maintains the single queue.

This model reflects how many state-of-the-art microsecond-scale systems are built [19, 36, 39, 43].

We define the system throughput overhead ($Overhead_{sys}$) as the fraction of CPU cycles that do not contribute towards application goodput. Eq. 1 describes the overall overhead for a system with n workers and 1 dispatcher. We separate the overhead based on the types of threads, i.e., $Overhead_w$ and $Overhead_d$ denote the per-worker and dispatcher overheads, respectively. Since the dispatcher does not run application logic, $Overhead_d = 1$.

To define $Overhead_w$, we consider the CPU cycles wasted during the lifetime of a request with a service time of S CPU cycles and summarize it in Eq. 2. Intuitively, for every request there are some cycles lost during processing (c_{proc}) beyond the application logic. These include overheads of the underlying runtime, such as for logging, and are proportional to the service time. Further, in a system that supports preemptive scheduling, there are also lost cycles associated with each preemption (c_{pre}) that include context switch and inter-thread communication costs. Finally, after the completion of a request, the worker will need to communicate with the dispatcher and wait for the next request, which will incur further wasted cycles (c_{fin}).

We now further break down these costs: c_{proc} is a fixed fraction of the service time (S) and depends on the implementation of the underlying runtime. c_{pre} is a cost paid on every preemption event, so $\lfloor S/q \rfloor$ times for every request, where q is the scheduling quantum; it includes the cost of receiving the preemption notification (c_{notif}), the context-switch cost (c_{switch}), and the cost of waiting for the next request (c_{next}), as seen in Eq. 3. Finally, c_{fin} consists of the context switch cost and the cost to fetch the next request (shown in Eq. 4).

$$Overhead_{sys} = \frac{n \times Overhead_w + Overhead_d}{n+1} \tag{1}$$

$$Overhead_w = \frac{c_{proc} + c_{pre} + c_{fin}}{S} \tag{2}$$

$$c_{pre} = \left\lfloor \frac{S}{q} \right\rfloor \times (c_{notif} + c_{switch} + c_{next}) \tag{3}$$

$$c_{fin} = c_{switch} + c_{next} \tag{4}$$

2.2 Sources of Throughput Overhead

We now use this model to analyze throughput overheads in state of-the-art microsecond-scale schedulers. Later, in §3, we introduce new mechanisms that address each of these overheads.

2.2.1 Preemptive Scheduling (c_{notif} or c_{proc})

Today, there exist two approaches for implementing preemption at the microsecond scale—interrupts and code instrumentation —that introduce significant overheads via the components c_{notif} and c_{proc}, respectively. We describe each approach using the corresponding state of the art—Shinjuku [36] and Compiler Interrupts [8], respectively—as canonical examples.

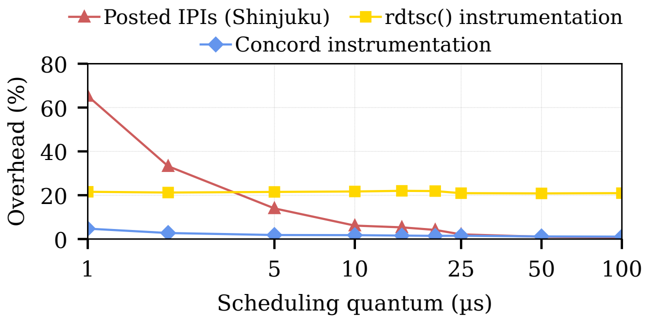


Figure 2. Overhead of preemption mechanisms as a function of the scheduling quantum. This overhead excludes the time required to context switch and receive a new request.

In interrupt-based systems, the dispatcher sends an inter-processor interrupt (IPI) to a worker whenever it has reached the desired scheduling quantum. The benefit of IPIs is that the preemption is precise; the worker promptly stops processing the current request and moves on to the next one. The drawback is the large cost of receiving IPIs (c_{notif}). Since this cost results in an overhead that is inversely proportional to the quantum size q, namely $Overhead \propto \frac{c_{notif}}{q}$ (Eq. 3), interrupt-based approaches lead to prohibitive throughput overheads at microsecond timescales. For instance, receiving an IPI in Shinjuku costs $\approx$1200 cycles which results in an $\approx$12% overhead for $q = 5\mu s$, and an $\approx$30% overhead for $q = 2\mu s$, assuming a 2GHz clock. Note, Shinjuku's IPIs rely on non-standard use of virtualization hardware and cannot be deployed in the public cloud. The corresponding overhead for Linux's easily-deployable IPIs is double [10, 36].

Instrumentation-based approaches forgo the dispatcher and rely solely on compile-time instrumentation of the code. The compiler inserts bookkeeping probes (e.g., `rdtsc()` calls) at regular intervals in the application code, enabling the worker to track how long it has been executing for and yield the CPU when the quantum has elapsed. This approach offers the benefit of avoiding IPIs, thus eliminating c_{notif}. However, bookkeeping probes are expensive (e.g., calling `rdtsc()` costs $\approx$30 cycles) and so inserting them frequently leads to prohibitively high overheads, while inserting them infrequently leads to poor preemption timeliness. In our model, the total cycles lost to bookkeeping is represented by c_{proc}. Since the bookkeeping is typically performed at significantly smaller granularities than microsecond-scale scheduling quanta [8], c_{proc} is a fixed fraction of the service time and leads to a throughput overhead independent of the scheduling quantum.

Fig. 2 provides empirical evidence for our model's analysis of the two preemption mechanisms. We measure the time it takes Shinjuku and Compiler Interrupts to service 1M requests, each running for 500µs, while handling preemption notifications with scheduler quanta from 1µs to 100µs. We compare to a baseline where each request runs to completion, without interruption. To isolate the preemption overhead, we run both systems with no-op preemption handlers.

As predicted by our model, IPIs in Shinjuku lead to an overhead that grows linearly as the scheduling quantum decreases: 33% at 2µs and 6% at 10µs. On the other hand, the `rdtsc()` probes used by compiler interrupts lead to a uniform ≈ 21% overhead across all scheduling quanta, since the probes are inserted approximately every 200 instructions, which is substantially smaller than 1µs.

2.2.2 Synchronous Inter-Thread Communication (c_{next})

Maintaining a single *physical* queue mandates synchronous communication between the dispatcher and worker threads. To ensure optimal load balancing in such systems, each worker thread must first finish processing the current request before it can pull the next request from the dispatcher. To avoid concurrency issues due to multiple workers pulling from the dispatcher, state-of-the-art systems [19, 36] implement a single queue as follows: (1) workers set a flag upon finishing a request and then poll a dedicated cache line for a new request; (2) the dispatcher continuously polls the workers' flags and sends a new request as soon as a flag is set.

This synchronous communication directly results in wasted CPU cycles (c_{next}), since workers sit idle until the dispatcher sends them a new request. In particular, c_{next} subsumes at least two cache coherence misses, which add up to ≈400 cycles in total [17]. These misses occur when (1) the dispatcher reads the flag previously written by the worker (Read after Write miss) and (2) the dispatcher writes into the worker's request queue that was last read by the worker when processing the previous request (Write after Read miss). Note, 400 cycles provides a lower bound on c_{next} since this assumes that the dispatcher sends a new request to the worker instantly. In practice, the dispatcher may be busy preempting or dispatching requests to any of the other n cores, so in the worst case, the worker thread might have to wait as long as 400×n cycles. For short requests, this idle time can lead to significant throughput overheads since the component of system overhead induced by c_{next} is inversely proportional to the service time: $Overhead \propto \frac{c_{next}}{S}$.

Fig. 3 illustrates the measured median overhead due to c_{next} for Shinjuku and Persephone when running with 8 cores. As predicted by the model, overhead is inversely proportional to service time. However, overhead increases slightly faster than $\frac{1}{S}$ because, with shorter request times, it becomes more likely that multiple workers finish while the dispatcher is busy sending a request to another worker.

2.2.3 Dedicated Dispatcher ($Overhead_d$)

Since the dedicated dispatcher does not run application logic even when idle, $Overhead_d = 1$.

While dedicating 1 core does not significantly impact throughput when running on a large server, it does have a serious impact for smaller VMs in the cloud. For example, consider a 16-core server with 1 dispatcher and 15 worker threads, where the dispatcher runs at full capacity to feed the 15 workers. When serving the same workload from a 4-vCPU VM in the

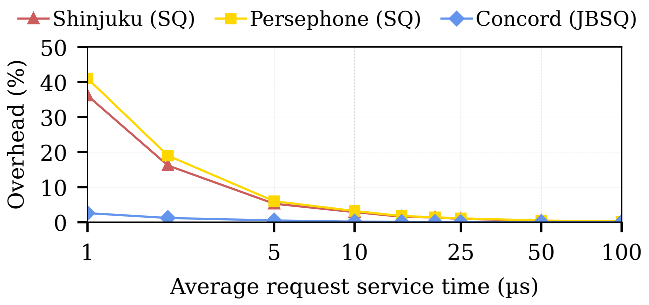


Figure 3. Time spent idle by a worker thread awaiting the next request in Single Queue (SQ) and JBSQ systems.

cloud, the dedicated dispatcher only serves 3 workers (20% of its capacity), and thus ends up being idle 80% of the time. As a result, in this particular deployment, the system as a whole sacrifices $\frac{80}{4\times100}$ = 20% of its potential maximum throughput.

To summarize, state-of-the-art microsecond-scale schedulers suffer from three main sources of throughput overhead: preemptive scheduling, synchronous inter-thread communication between the dispatcher and workers to maintain the single queue, and the dedicated dispatcher that runs no application logic. In the next section, we describe how Concord, our proposed scheduling runtime, addresses each source of overhead.

3 Design

Concord's design is driven by the insight that careful *approximation* (as opposed to canonical implementation) of optimal scheduling policies enables efficient, low-overhead *mechanisms* that lead to significant throughput benefits at negligible tail-latency costs.

Concord relies on three key mechanisms to efficiently approximate the optimal single queue and precise preemption policies and mitigate the sources of throughput overhead described in §2. First, compiler-enforced cooperation (§3.1) approximates precise preemption using asynchronous communication between the dispatcher and worker threads. While this asynchrony leads to slightly imprecise scheduling quanta, it does not significantly impact tail latency. Instead, it enables Concord to reduce the preemption overhead by 4× by minimizing c_{notif} while keeping c_{proc} low. Second, Join-Bounded Shortest Queue (JBSQ) scheduling (§3.2) adds bounded core-local queues to approximate a single queue; this enables Concord to reduce cache-coherence stalls in worker threads by 9−13× by nearly eliminating c_{next}. Finally, the Concord-dispatcher (§3.3) is work-conserving and steals work from the global single queue when all worker threads are busy. This approximates both the single queue and precise preemption—the dispatcher sends preemption notifications late when busy—but ensures $Overhead_d$ < 1 which significantly improves application throughput at low core counts. Fig. 4 provides an architectural diagram of Concord that we gradually explain throughout the section.

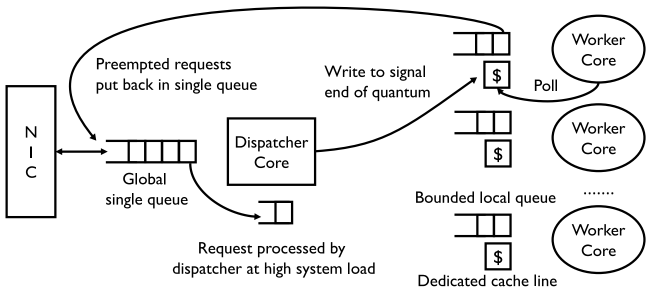

Preempted requests put back in single queue
Write to signal end of quantum
Worker Core
Poll
Worker Core
N I C
Dispatcher Core
Global single queue
Bounded local queue
Worker Core
Request processed by dispatcher at high system load
Dedicated cache line
$

Figure 4. The Concord architecture. Concord's compiler-enforced cooperation relies on communicating via a dedicated cache line, JBSQ(k) employs bounded core-local queues to eliminate coherence stalls, and the Concord dispatcher steals work at high load to contribute to application throughput.

Concord uses an asymmetric threading model with a dispatcher thread D and worker threads $W_1, ..., W_n$; in §3.1 we describe how this design enables Concord to enables low-overhead preemption while retaining the flexibility to support arbitrary scheduling policies. Each worker thread is pinned to a CPU core, to ensure maximum locality. This architecture is consistent with §2.1 and how state-of-the-art microsecond-scale systems are built today [16, 19, 36, 39, 43, 46].

3.1 Compiler-Enforced Cooperation

We now describe how Concord's compiler-enforced cooperative scheduling provides an alternative to IPIs that: (1) enables preemption at lower overhead for microsecond-scale tasks, (2) does not require non-standard use of hardware and can be deployed in the public cloud, and (3) makes it easier to port applications and preempt safely.

In Concord, scheduling decisions are communicated between workers W_i and the dispatcher D via a per-core dedicated cache line L_i, instead of IPIs. The Concord runtime enforces this communication for arbitrary applications using automated compiler instrumentation. The dispatcher monitors how long each request has been executing and writes to L_i when the request has reached the end of its scheduling quantum. Concord's compiler instrumentation ensures application code running on W_i periodically checks cache line L_i for a preemption signal from the dispatcher. When the signal is received, the worker thread writes to L_i indicating to D that preemption has taken place, and yields. Yielding consists of saving the context corresponding to the current request, and then switching to the default worker context, which awaits the next request. The dispatcher re-places the preempted request on the main queue. Thus, Concord *automatically* converts worker threads from being "interrupt-driven CPU drivers" to "poll-mode CPU drivers". This is consistent with how the majority of low-latency systems today eschew interrupts in favor of polling due to the associated overheads [20, 32, 33, 36, 40, 47, 50, 63].

Concord deliberately separates scheduling concerns between D, in charge of signaling the end of a quantum, and W_i, in charge of yielding. D has global visibility of the system, and so it is best positioned to decide when W_i should stop processing a request and which request it should begin processing instead. On the other hand, cooperative yielding allows worker threads to switch between requests within $\approx 100ns$, and avoids expensive preemptive context switches. Delegating the preemption notifications to the dispatcher ensures that Concord can support scheduling algorithms beyond First Come, First Served (FCFS) and Processor Sharing (PS). For instance, Concord can easily be extended to support algorithms such as Shortest Remaining Processing Time [52] or ones that takes locality into account and prioritize scheduling preempted requests back on to the core they were last processed by. Implementing such algorithms in single *logical* queue systems is hard, since they do not have a dispatcher, and thus have no core that possesses visibility of all the requests in the system.

Communicating scheduling decisions via shared cache lines enables Concord to *minimize* c_{notif} (cost of preemption notification), while keeping c_{proc} (instrumentation overhead) low. c_{notif} is minimized since a shared cache line is the fastest way for two cores to communicate in commodity shared-memory processors. This minimization does not significantly increase c_{proc} because, unlike an `rdtsc` call, which always costs $\approx$30 cycles, the cache line L_i is in the L1 cache of worker W_i for all but the final check, so most checks consist of an L1 cache hit plus a compare, i.e., 2 cycles. The final check (after which the request yields), incurs a Read after Write cache-coherence miss since it is the first check after the dispatcher writes to L_i. However, this miss only costs $\approx$150 cycles leading to a c_{notif} that is $\frac{1}{8}^{\text{th}}$ the cost of a Shinjuku IPI (which costs $\approx$1200 cycles), while eschewing reliance on the non-standard use of virtualization hardware.

Fig. 2 shows this overhead, for scheduling quanta from 1-100µs. We see that Concord's overhead is near-constant at around 1-1.5%, mainly coming from the instrumentation and not the notification itself, which is consistently 16× cheaper than invoking `rdtsc()`. Concord's overhead is also 12× lower than that of Shinjuku's IPIs at a scheduling quantum of 2µs and 10× lower at a quantum of 5µs. As the quantum increases further, the percentage overhead of an IPI decreases until the two become roughly equal ($\approx$ 0.7%) at around 25µs. Note, 25µs refers to the scheduling quantum and not the service time, so even datacenter applications that have some long requests (e.g., 100µs to 10ms) will benefit from Concord, as long as there are also many short requests (1-10µs) for which we would like to preempt the long-running requests. Many real-world applications have such distributions e.g., search engines, microservices and function-as-a-service (FaaS) frameworks, and in-memory stores or databases such as RocksDB, LevelDB, and Redis that support both point and range queries.

Compiler-enforced cooperation approximates precise preemption since workers do not yield instantaneously: the application code must first reach the cache-line check to see the

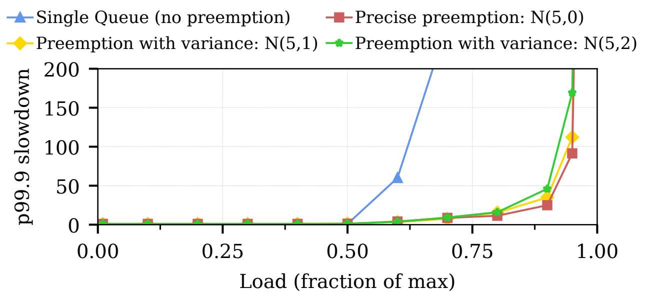


Figure 5. The impact of non-instantaneous preemption on 99.9th percentile request slowdown. $N(x, y)$ represents a normal variable with mean x and standard-deviation y

preemption notification. In practice though, we observed that as long as preemption occurs within a "small" interval around the desired quantum, tail latency is not significantly affected. This ensures that compiler-enforced cooperation outperforms IPIs by achieving greater throughput while meeting the same tail-latency SLO.

Fig. 5 demonstrates the impact of non-instantaneous preemption using a queueing simulation for a bimodal service time distribution from prior work [19, 36]. In this distribution, 99.5% of requests take 0.5µs, and 0.5% of requests take 500µs. We model Concord's preemption as a one-sided Normal random variable[1] with a mean of 5µs and different standard deviations. We also plot the simulation results for precise preemption (red line), which is the optimal behavior, and no preemption (blue line), which serves as a lower bound. We observe that for small standard deviations, the latency behavior of imprecise preemption is almost identical to that of precise preemption, indicating that approximating the preemption quantum does not significantly affect tail latency. In §5.4, we show that for a 5µs quantum, Concord's instrumentation keeps the standard deviation within 2µs across 25 benchmarks from standard benchmark suites.

Safety-first preemption: Concord takes a safety-first approach to preemption that we believe is particularly suited to microsecond-scale applications. Concord ensures safety by not preempting worker threads when they are either performing external calls that might acquire locks (e.g., system calls) or holding a lock in the application code. While such an approach can (in theory) lead to tail-latency spikes due to long-running critical sections or system calls, in our experience this is rarely the case in practice because such calls are infrequent in microsecond-scale application code.

Concord guarantees that preemption is avoided within external calls by construction, since the compiler has full control over the portions of code it instruments. This has the added benefit of ensuring that widely-used libraries (e.g., `libc`) can be used in Concord without modification. In contrast, such libraries must be modified to ensure safety in systems that rely on IPIs (e.g., Shinjuku) since the worker thread has no control over what code it will be executing when it receives an IPI from the dispatcher.

To avoid preemption while holding application locks developers must modify their code; however, in our experience, this takes negligible effort. For example, to achieve safety in LevelDB we only had to add a total of 4 lines of code that incremented/decremented a counter whenever a mutex was locked/unlocked in the application code. By only preempting if the counter was zero, Concord ensured that it would never preempt a worker thread while it held a lock. On the other hand, the Shinjuku prototype avoids this issue by disabling preemption during entire LevelDB API calls. However, this approach can lead to significant tail-latency spikes since entire LevelDB API calls can run for significantly longer than just their critical sections. It was easy for us to create a microbenchmark where the worker thread in Shinjuku was not preempted until 100µs due to a long-running LevelDB GET API call. For this microbenchmark, Concord improved throughput by 4× in comparison to Shinjuku while meeting the same tail-latency SLO.

3.2 Stall-Free Workers

To eliminate the overhead due to worker threads being idle, Concord carefully trades the optimal single queue policy in favor of the Join-Bounded-Shortest-Queue policy [40], abbreviated JBSQ(k). JBSQ(k) approximates an ideal, work-conserving single queue by combining a single, central queue with short, bounded per-worker queues, each with a maximum depth of k messages. JBSQ(1) is therefore equivalent to a single queue.

JBSQ enables Concord to forego the purely pull-based single queue and adopt a controlled *push-based* policy: Whenever there is a pending request in the main queue, and one or more per-worker queues have empty slots, the dispatcher pushes the request to the shortest per-worker queue. This ensures that, upon completing a request, worker threads can immediately begin processing a new request from their local queue, thus eliminating the idle time spent waiting for the next request.

To ensure that the per-worker queues do not significantly impair load balancing (and hence tail latency), k must be just large enough to ensure that a worker is never idle during the dispatcher-worker communication. Any larger value of k only hurts tail latency without improving throughput. While the exact communication delay is a complex function of the number of workers and the service time distribution, we found $k = 2$ to be sufficient for service times above 1µs. Approximately, a value of $k = \lceil \frac{c_{next}}{S} \rceil + 1$, where S is the service time, should ensure zero idle time. Prior work [40] has shown that $k = 2$ imposes a negligible tail-latency penalty over the optimal single queue.

[1]One-sided because Concord never preempts before the quantum

Fig. 3 compares the throughput overhead due to idling in state-of-the-art systems implementing a single queue and Concord, which uses JBSQ(2). We observe that using JBSQ(2) results in an overhead that is 9−13× lower. Of course, JBSQ(2) does not make c_{next} zero. This is because the asynchronous dispatching and processing of requests requires the worker to start a timer denoting the scheduling quantum; in a synchronous single queue, this can be done by the dispatcher.

Concord is the first to make the observation that JBSQ(k) is necessary to mask cache-coherence latencies during inter-thread communication. In fact, JBSQ(k) is the accepted policy when the communication delay between the dispatcher and workers approaches the average service time [15, 27, 30, 40, 54]. This is because it enables explicit control of the trade-off between tail latency and throughput, through the choice of k, thus making it possible to pick an optimal queue depth. Prior work has already explored the use of JBSQ(k) in scenarios where the dispatcher was located on either a programmable switch [40] or a smartNIC [27, 30].

3.3 Work-Conserving Dispatcher

To reduce $Overhead_d$ (=1 for a dedicated dispatcher) the Concord dispatcher contributes to application goodput by processing user requests for one quantum, whenever it notices that all per-worker queues are full.

Concord leverages `rdtsc()`-based code instrumentation for the dispatcher to ensure that it can process user requests while continuing to respond to network and worker events in a timely manner. `rdtsc()`-based instrumentation is necessary since there is no external agent to send preemption signals to the dispatcher and so it must be able to self-preempt. The automatically inserted `rdtsc()` probes periodically check whether it is time for the dispatcher to switch from application requests to dispatching. As a result, Concord has two differently instrumented versions of the application code. The expensive, `rdtsc()`-based instrumentation is only used for the dispatcher thread, while the cache-line polling is used on the worker threads. Since all threads are pinned to CPU cores, the second version does not cause I-cache pressure at the workers; these instructions are limited to the dispatcher's private I-cache.

Having two versions of the code requires a slight modification to the single queue. This is because requests that have been processed by a worker cannot be processed by the dispatcher and vice versa, since the instruction pointers are different due to the different instrumentation. Hence, the dispatcher can only pick up non-started requests from the central queue, and, once it starts processing a request, it is solely responsible for completing that request. So, whenever D is idle, it picks the first non-started request from the central queue. If it needs to preempt itself before completing the request, it saves the context to a dedicated buffer. The next time D is idle, it picks up the request from this buffer and continues processing it.

This approximation does not significantly impact tail latency. We present an intuitive argument now, and demonstrate it empirically in §5. First, at low loads it is unlikely that all per-worker queues will be full, hence the dispatcher is unlikely to have a request to pick up from the queue. To understand the impact at high loads, assume that the dispatcher is idle for 50% of each time quantum, and the `rdtsc` instrumentation induces 20% overhead. This makes the dispatcher only $50\% - 50 \times 20\% = 40\%$ as effective as a typical worker, causing the request to take 2.5× the usual service time. In practice, this overhead turns out to be far less than the time the request would spend queueing or bouncing around different workers at high load if the dispatcher hadn't taken it over for processing. Typical tail slowdown targets at high load are $20 - 50\times$ the service time (according to [19, 30]), and so we believe that the 2.5× is acceptable.

To summarize, Concord efficiently approximates the optimal single queue and precise preemption to mitigate the throughput overheads that plague state-of-the-art microsecond-scale schedulers. To do so, Concord leverages three mechanisms—compiler-enforced cooperation, JBSQ(k) scheduling and a work-conserving dispatcher—all of which eschew the non-standard use of hardware and application-level assumptions.

4 Concord Prototype

In this section, we describe the key implementation details of our Concord prototype.

4.1 API

Concord's API comprises three callbacks:

- `setup()` initializes global application state
- `setup_worker(int core_num)` initializes application state for each worker thread, such as local variables or configuration options
- `response_t* handle_request(request_t*)` processes a single application request and returns a pointer to the response. At any particular point in time, a request is only processed by a single thread, although preemption might cause it to be served by multiple threads over its entire service time.

This simple event-driven API hides all of Concord's underlying complexity from application developers and enables Concord to be easily integrated into existing dataplane OSes; we now describe two integrations.

4.2 Concord Runtime

We integrate the Concord runtime into two state-of-the-art microsecond-scale OSes, Shinjuku and Persephone.

The Concord-shinjuku implementation was straightforward, since Shinjuku's dispatcher already implements a preemptive scheduling policy and there exists a userlevel threading mechanism. We only had to change the preemption signal, add per-core queues and add support for dispatcher work-stealing. Concord-shinjuku only required adding 847 LOC to Shinjuku's initial codebase.

The Concord-persephone implementation required more effort since Persephone operates in a run-to-completion manner. Thus, we had to implement userlevel threading and ported Shinjuku's implementation to Persephone for the same. JBSQ was easier to implement here since Persephone already supports multi-request queues. In total, Concord-persphone adds 2358 LOC to Persephone.

4.3 Concord Compiler

We implemented Concord's code instrumentation as two LLVM passes; one each for polling a shared cache line and checking `rdtsc()`, respectively. Both passes use LLVM version 9 and comprise $\approx$ 350 LOC each.

The Concord compiler places probes at the beginning of each function call, before and after any call to un-instrumented code (e.g., syscalls) and at every loop back-edge. Placing probes as such has been shown empirically to be sufficient to yield on all long paths through code [8, 35]. For non-loop code, this translates into a probe being placed approximately once every 200 LLVM IR instructions [8], and so, to avoid prohibitive overheads arising from tight program loops, we unroll each loop body until it has at least 200 LLVM IR instructions. With additional engineering effort, it should be feasible to place probes more infrequently for both loops and non-loop code. We did not to pursue this goal in our work since Concord's instrumentation overhead is already low ($\approx$1% on average).

5 Evaluation

We evaluate Concord to answer the following questions:

- How does Concord perform across different service time distributions for which different scheduling policies are optimal? (§5.2)
- How does Concord perform for a real, latency-sensitive application? (§5.3)
- What is the contribution of each individual mechanism to Concord's performance benefits? (§5.4)
- What are the drawbacks of Concord's design? (§5.5)
- Do Concord's mechanisms remain useful as datacenter server hardware evolves to provide increased support for microsecond-scale scheduling? (§5.6)

5.1 Methodology

Baselines: We focus on blind policies, i.e. policies that do not rely on application-level information, and pick two baselines that represent the state of the art for workloads with high and low service time dispersion, respectively. Shinjuku represents the state of the art for workloads with high service time dispersion since it implements both a single queue and preemptive scheduling. To compare against recent systems [19, 46] that implement only an FCFS single queue for workloads with low dispersion, we configure Persephone to use the C-FCFS policy. We refer to this baseline as "Persephone-FCFS".

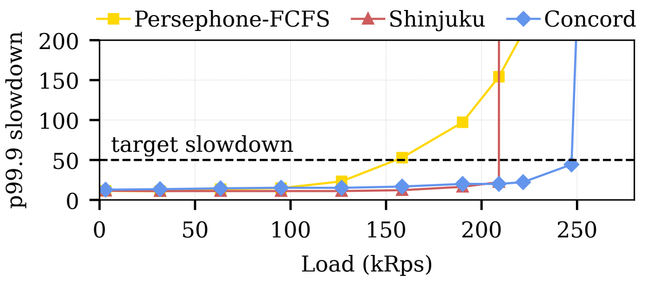
For all experiments on our own cluster, we use the Concord implementation that builds on top of Shinjuku. Since Shinjuku is the best-performing baseline in this context, using this implementation enables an apples-to-apples comparison between Concord and Shinjuku. As detailed in §4, the performance differences between the two Concord implementations are minuscule.

Testbed: We use a testbed set up as per RFC 2544 [59] with two directly connected machines—a server that runs Concord or the baselines and a client that runs a load generator. Both machines are identical Cloudlab [13] c6420 nodes with a 32-core (64-thread) Intel Xeon Gold 6142 CPU running at 2.60GHz, with 376 GB of RAM, and an Intel X710 10 Gbps NIC. The average network round trip time between the client and server is 10μs. The server machine runs Ubuntu 18.04 with the 4.4.185 Linux Kernel since this is the version that Shinjuku's kernel module requires. We set up each system as in prior work [19]: Shinjuku uses one hyperthread for the networker and another for the dispatcher, co-located on the same physical core. Persephone runs both its net worker and dispatcher on the same hardware thread. Unless otherwise specified, all systems use 14 worker threads running on dedicated physical cores.

The client's load generator sends requests according to a Poisson process centered at the workloads' mean service time to mimic the bursty behavior of production traffic [6]. Unless specified, all measurements are performed at the client, ensuring end-to-end evaluation of Concord. Each experiment runs for 60 seconds and we discard the first 10% of samples to remove warmup effects.

Workloads: We used one synthetic and one real application to evaluate Concord across several service distributions from both academic and industrial references. The synthetic workload is a server application that spins for the amount of time specified by each request; this application allows us to evaluate Concord across a variety of service time distributions. In §5.2 we describe four such distributions, three of which are based on workload A from the YCSB benchmark [14], Meta's USR workload [6] and TPCC running on an in-memory database [19], respectively.

The real application is a server running LevelDB [42], a popular and widely deployed key-value store developed by Google that supports both point queries (put/get requests) and range queries (scans). We evaluated LevelDB on two service time distributions, one from Meta's ZippyDB traces [11] and the other from prior work [19, 36]. Unless otherwise specified, we use two scheduling quanta—5μs and 2μs respectively—for all workloads. Many of our workloads were also used by



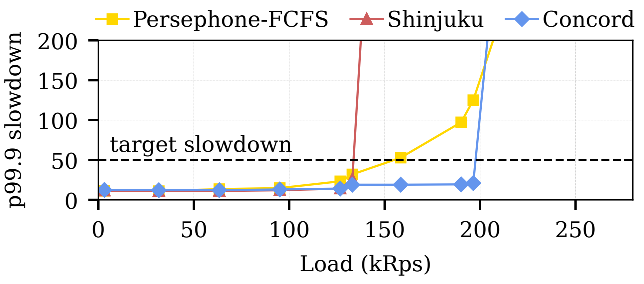
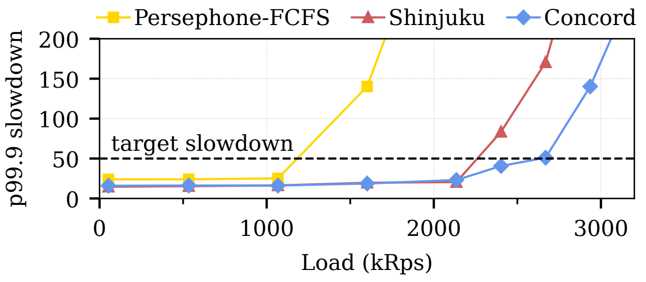


Figure 6. 99.9th percentile slowdown vs load for Bimodal$(50:1, 50:100)$. Scheduling quantum is 5µs (left) and 2µs (right).



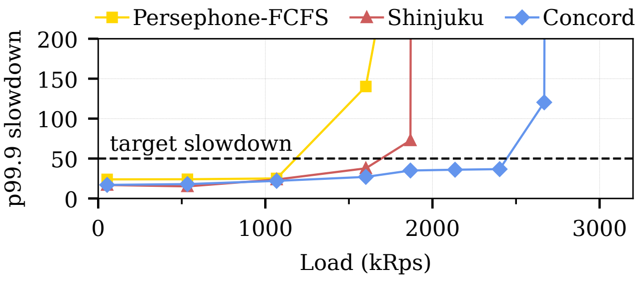


Figure 7. 99.9th percentile slowdown vs load for Bimodal$(99.5:0.5, 0.5:500)$. Scheduling quantum is 5µs (left) and 2µs (right).

Shinjuku and Persephone; in all such cases, we were able to replicate their published results. When running on Shinjuku and Persephone, the application code (both synthetic and LevelDB) was not instrumented by the Concord compiler.

Metrics: For each workload, we primarily compare the throughput that the two systems can sustain given a target 99.9th percentile *Slowdown* — which is the ratio of the total time the request spends at the server to its un-instrumented service time. Using tail *Slowdown* (instead of latency) allows us to evaluate all workloads at a common Service Level Objective (SLO), despite their absolute latencies varying significantly. For all experiments, we set the slowdown SLO at 50× the service time which is consistent with prior work [19, 36].

5.2 Synthetic Workload Comparison

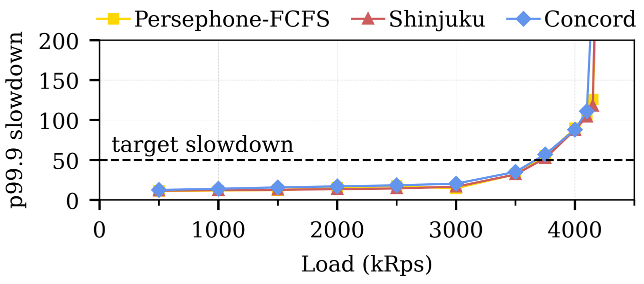
Here we mimic four service time distributions, two each with high and low dispersion respectively. The first two distributions stress Concord's approximate preemption and its approximate single queue, while the last two only stress its single queue.

Workloads with high dispersion that benefit from preemption: Both high-dispersion workloads follow a bimodal service-time distribution. In the first (Bimodal(50:1, 50:100)), 50% of the requests have a 1µs service time and the other 50%, 100µs. Such a distribution with an equal amount of short and long requests is based on workload A from the YCSB benchmark [14]. In the second (Bimodal(99.5:0.5, 0.5:500)), 99.5% of the requests have a 0.5µs service time and 0.5%, 500µs. This distribution, with a majority of short requests and a few very long requests, is based on Meta's USR workload [6].

Fig. 6 and Fig. 7 illustrate the results for the two high-dispersion distributions for scheduling quanta of 5µs and 2µs, respectively. For a scheduling quantum of 5µs, Concord can support 18% and 20% greater throughput than Shinjuku for our 99.9th percentile slowdown SLO of 50×. Similarly, for a scheduling quantum of 2µs, Concord supports 45% and 52% greater throughput than Shinjuku. Due to its lack of preemptive scheduling Persephone-FCFS crosses the slowdown SLO much earlier than the other two systems for workloads with high dispersion.

Workloads with low dispersion that do not benefit from preemption: The first low-dispersion workload (Fixed(1)) uses a fixed service time of 1µs for all requests. The second workload (TPCC) is based on the service time distribution of TPCC [57] running on an in-memory database [58] and is taken from prior work [19]. The distribution of request types and service times is as follows: Payment (5.7µs) - 44%, OrderStatus (6µs) - 4%, NewOrder (20µs) - 44%, Delivery (88µs) - 4%, and StockLevel (100µs) - 4%. For Fixed(1), we continue to use scheduling quanta of 5µs and 2µs. For TPCC, we set the quantum to 10µs to avoid unnecessary preemptions, since all requests run for longer than 5µs.

While such workloads do not benefit from preemption (since there are too few long requests that block the shorter requests), we observe that Concord still performs favorably w.r.t the state-of-the-art; Fig. 8 illustrate the results. We see that for the Fixed(1) workload (Fig. 8(a)), Concord achieves effectively the same (2% less) throughput than Shinjuku and Persephone. In such situations, the bottleneck is the dispatcher thread—common to all three systems—which cannot deliver



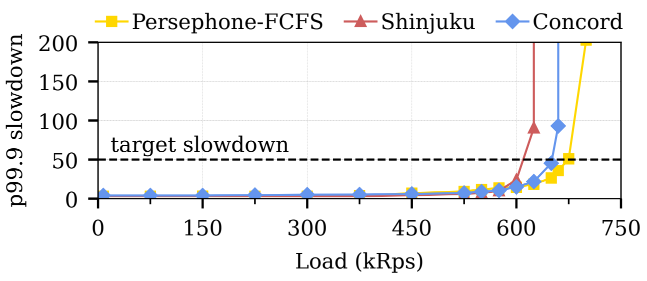
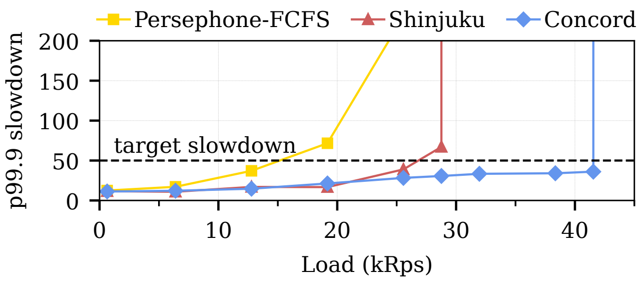


Figure 8. 99.9th percentile slowdown vs load for Fixed(1) (left) and TPCC (right) service time distributions.



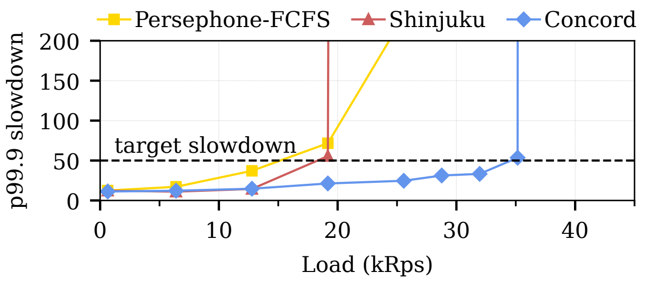


Figure 9. 99.9th percentile slowdown vs load for a LevelDB workload with 50% GETs, 50% SCANs. Quantum is 5µs (left) and 2µs (right).

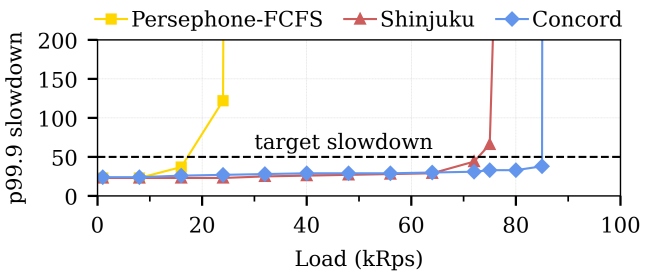


Figure 10. 99.9th percentile slowdown vs load for LevelDB workload based on ZippyDB production traces [11]. Quantum is 5µs.

requests to workers fast enough. Concord's dispatcher incurs the 2% penalty since it must calculate the "shortest queue" for each incoming request to implement JBSQ(2). For the TPCC workload (shown in Fig. 8(b)), which has low dispersion and the dispatcher is not the bottleneck, preemption overheads in Shinjuku and Concord harm throughput compared to Persephone-FCFS, yet Concord still outperforms Shinjuku given its low-overhead preemption mechanism.

5.3 LevelDB Server

We now compare Concord's performance to Shinjuku's and Persephone's for a LevelDB server that supports both point and range queries. We set up LevelDB in a manner similar to prior work [19, 36]. We populate the database with $15,000$ unique keys and use memory-mapped plain tables to keep all data in memory. In this setup, GET requests take $\approx 600ns$ each, PUT, DELETE requests take $\approx$ 2.3µs each and SCANs take approximately 500µs.

We evaluate Concord's throughput improvements for LevelDB using two request distributions. The first distribution consists of 50% GET requests for a single key and 50% SCAN requests that scan the entire database. This workload strikes a balance between the previous two Bimodal distributions and was used by both Shinjuku and Persephone. The second distribution is based on recently published Meta traces [11] from their ZippyDB service. This workload consists of 78% GETs, 13% PUTs, 6% DELETEs and 3% SCANs. We use scheduling quanta of 5µs and 2µs for the first distribution. We use only a 5µs quantum for the second distribution since all requests in the workload run for longer than 2µs and so a 2µs quantum leads to unnecessary preemptions. Both distributions also allow us to evaluate how Concord performs for real application code with locks since in LevelDB, both PUT and GET requests acquire locks.

Fig. 9 illustrates the results for the first distribution. We observe that for our 99.9th percentile slowdown target of 50×, Concord supports 52% greater throughput at a scheduling quantum of 5µs and 83% greater throughput at a scheduling quantum of 2µs. Concord's throughput improvement over prior work is larger for this workload because it has greater dispersion (1000×) than the previous microbenchmarks. At such dispersions—which are common in production workloads [3, 12, 45])—all three of Concord's mechanisms shine. JBSQ(2) ensures no worker thread is ever idle (minimizing c_{next}), compiler-enforced cooperation ensures that long requests do not suffer prohibitive overheads due to frequent preemption (eliminating c_{notif}) and the incoming requests per second is low enough for the Concord dispatcher to frequently remain idle and thus contribute to application goodput (improving $Overhead_d$). In §5.4, we provide a quantitative breakdown for this improvement.

Fig. 10 illustrates the results for the second distribution, which is based on Meta's production traces from their ZippyDB service. We see that Concord supports 19% greater throughput than Shinjuku for the target 50× slowdown. This improvement is in line with Concord's results for Bimodal($99.5 : 0.5, 0.5 : 500$), shown in Fig. 7(a). This is unsurprising, since the two service time distributions are similar.

5.4 Deep-Dive Into Concord's Mechanisms

We now evaluate each of Concord's key mechanisms—compiler-enforced cooperation, JBSQ(2) scheduling, and the work-conserving dispatcher—individually.

Overhead and timeliness of compiler-enforced cooperation across applications: Since compiler instrumentation rarely produces uniform slowdowns, we evaluated the overhead and timeliness of Concord's instrumentation across 24 benchmarks from the Phoenix [49], Parsec [48] and Splash-2 [53] benchmark suites. We compared Concord's instrumentation overheads to published numbers from prior work [8] that uses `rdtsc()`-based instrumentation. We used their published numbers because we were unable to accurately replicate their results. To obtain optimal overhead numbers, their LLVM pass must be differently configured with 8 parameters for each application and naive configurations lead to significant overheads. They do not publish numbers for preemption timeliness.

Table 1 presents the results across the 24 benchmarks. We observe that Concord's average instrumentation overhead is not only low enough to be acceptable ($\approx 1.04\%$ on average) but also 13.1× lower than the state of the art, with the maximum overhead being 5.5× lower.

To evaluate Concord's preemption timeliness, we set a quantum of 5µs and measured the standard deviation from the target quantum for the same set of applications (last column of Table 1). Across all benchmarks, we see that the standard deviation is smaller than 2µs and so well within the tolerable imprecision (§3.1). Further, the 99$^{\text{th}}$ percentile of the achieved scheduling quanta was always within 3 standard deviations ensuring that Concord's imprecise scheduling quanta do not significantly impact tail latency.

Breaking down throughput improvements: We evaluated the contribution of each of Concord's mechanisms to its throughput improvement by measuring the throughput sustained by a system that cumulatively employs Concord's key mechanisms for a LevelDB workload consisting of 50% GETs and 50% SCANs.

Fig. 11 illustrates the results. We observe that in comparison to the $\approx$ 19kRps sustained by Shinjuku at the target 50× slowdown, systems that cumulatively employ compiler-enforced cooperation, JBSQ(2) scheduling and a work-conserving dispatcher sustain a throughput of $\approx$ 22.5 kRps, $\approx$ 32 kRps, and $\approx$35 kRps, respectively.

Program name	Benchmark Suite	Concord overhead	CI overhead	Concord std.dev
`water-nsquared`	Splash-2	-0.3%	3%	0.24µs
`water-spatial`	Splash-2	-0.6%	4%	0.23µs
`ocean-cp`	Splash-2	0.1%	10%	1.8µs
`ocean-ncp`	Splash-2	1%	6%	1.1µs
`volrend`	Splash-2	0.5%	13%	0.47µs
`fmm`	Splash-2	0.4%	-2%	0.11µs
`raytrace`	Splash-2	-0.2%	4%	0.03µs
`radix`	Splash-2	0.9%	4%	0.56µs
`fft`	Splash-2	1.2%	1%	0.63µs
`lu-c`	Splash-2	4.6%	13%	0.63µs
`lu-nc`	Splash-2	-3.7%	23%	0.58µs
`cholesky`	Splash-2	-2.9%	29%	0.86µs
`histogram`	Phoenix	1.6%	20%	0.57µs
`kmeans`	Phoenix	-0.3%	3%	1µs
`pca`	Phoenix	-2.7%	25%	0.06µs
`string_match`	Phoenix	2%	18%	0.86µs
`linear_regression`	Phoenix	6.7%	37%	0.78µs
`word_count`	Phoenix	2.4%	30%	1.11µs
`blackscholes`	Parsec	4%	10%	1.14µs
`fluidanimate`	Parsec	1.3%	2%	0.04µs
`swapoptions`	Parsec	2.2%	24%	0.86µs
`canneal`	Parsec	1.5%	34%	0.02µs
`streamcluster`	Parsec	-2.1%	6%	0.08µs
`dedup`	Parsec	0.4%	4%	1.2µs
Average	-	**1.04%**	**13.7%**	**0.29µs**
Maximum	-	**6.7%**	**37%**	**1.8µs**

Table 1. Overhead and timeliness of Concord's instrumentation compared to Compiler-Interrupts (CI) [8]. The baseline (0% overhead) corresponds to un-instrumented code. Concord's overhead is often negative due to its loop unrolling.

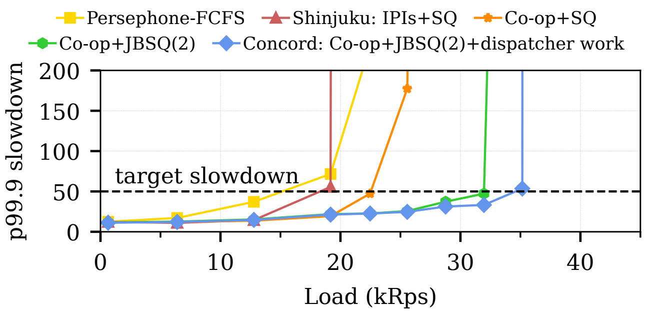


Figure 11. Contribution of each Concord mechanism towards throughput improvement for the LevelDB server in Fig. 9(b). SQ refers to the single queue scheduling policy.

We now provide an intuitive argument for these improvements. First, the 3.5kRps improvement due to cooperation can be seen as nearly eliminating the 20% cost of interrupt-based preemptions ($3.5 \approx 0.2 * 19$). Second, JBSQ(2) eliminates $\approx 400ns$ of idle time per request; this time is used to effectively double the number of GET requests processed which leads to an additional 9.5 kRps (0.5×19) in throughput. Finally, since the absolute load (in kRps) is $\approx$ 100× lower than the maximum throughput the dispatcher can sustain (Fig. 8), the dispatcher spends most of its time idle, allowing it to also contribute to application throughput.

Reduction in preemption overhead: Next, we demonstrate how Concord reduces the throughput overhead of preemptive

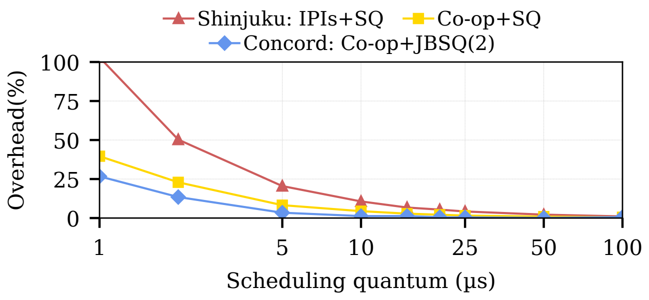


Figure 12. Contribution of each Concord mechanism towards its overall reduction in preemption overhead.

scheduling, and break down the contributing factors. To do so, we measure the time it takes to service $1M$ requests, each running for 500µs, while handling preemption notifications and yielding. Since the preemption overhead (Eq. 3) includes both the cost of the preemption notification (addressed by compiler-enforced cooperation) and the time spent waiting for the next request to arrive (addressed by JBSQ(2) scheduling), we break down the contribution of each mechanism by evaluating the overhead of systems that cumulatively use the above two mechanisms. We use the state-of-the-art interrupt-based approach (Shinjuku) as a baseline and perform this experiment for scheduling quanta ranging from 1-100µs.

Fig. 12 illustrates the results. We observe that Concord reduces the overhead of preemptive scheduling by 4× in comparison to Shinjuku. In this setting (unlike Fig. 11), we observe that compiler-enforced cooperation is the mechanism that contributes the most towards Concord's improvements. This is because unlike the workload in Fig. 11, every request must be preempted, and so reductions in the cost of the preemption notification dominate.

Does the dispatcher do useful work? Finally, we demonstrate the benefit of running application logic on the dispatcher thread in resource-constrained environments (e.g., smaller VMs in the public cloud).

Fig. 13 illustrates our results. We ran the same LevelDB workload on 4 cores—1 dispatcher, 1 networker, two workers—to simulate smaller VMs in the public cloud. In such situations, particularly with the low incoming load (in absolute kRps), the dispatcher is almost entirely idle. Hence, running application logic on the dispatcher thread improves application throughput by 33%.

5.5 The Drawback of Approximate Scheduling

While approximating optimal scheduling enables Concord to sustain higher application throughput for the same tail latency/slowdown SLO, it comes with the drawback of slightly increasing tail latency (and hence slowdown) at lower loads.

Fig. 14 illustrates this increased slowdown for the workload used in Fig. 6 (Bimodal(50 :1, 50 : 100)); we observed similar results across all workloads. We observe that Concord

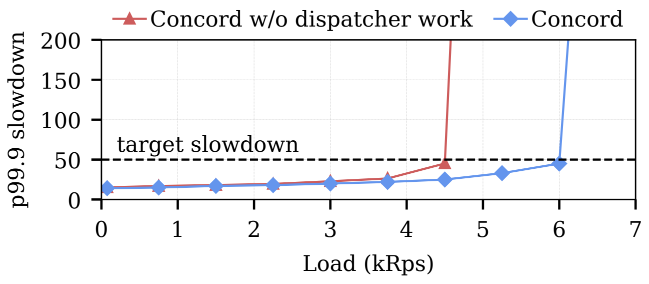


Figure 13. Application throughput for dedicated dispatcher vs. Concord dispatcher in a 4-core configuration.

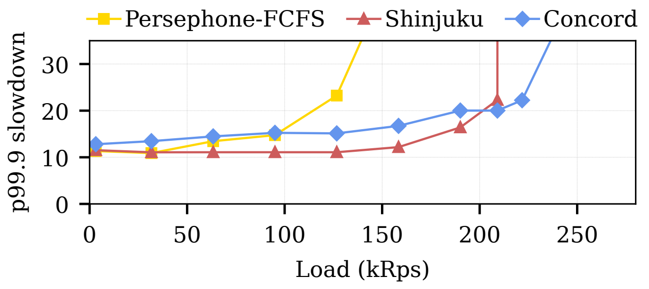


Figure 14. Zoomed-in version of Fig. 6(a) to show Concord's increased slowdown at low loads. We observed similar increases across all other workloads.

increases the 99.9th percentile slowdown by ≈ 3 in comparison to Shinjuku at lower loads. This increase in tail slowdown occurs when the Concord dispatcher steals requests during occasional bursts even at low loads. Since these requests cannot be migrated to worker cores once the dispatcher has started processing them due to the different code instrumentation (§3.3), they run slower than ones processed by worker cores, which leads to this additional slowdown. That said, we believe that this increase in slowdown is acceptable since it is much smaller than typical SLOs, which are usually $10 - 50\times$ the service time to account for queueing and network round trip times. Users unwilling to tolerate even this slight increase in slowdown at low loads can disable the Concord dispatcher's work-stealing mechanism and retain the throughput benefits of Concord's compiler-enforced cooperation and JBSQ(2) scheduling.

5.6 Is Concord Future-Proof?

Finally, we evaluate whether Concord's mechanisms will remain useful as datacenter server hardware evolves and provides increasing support for microsecond-scale scheduling.

To do so, we compare the throughput overhead of Concord's preemption mechanism (compiler-enforced cooperation) with that of user-space IPIs (UIPIs), a feature recently introduced by Intel on their new Sapphire Rapid servers. UIPIs reduce the overhead of traditional IPIs by allowing application threads to directly send each other interrupts while bypassing the kernel [62]. We set up our experiment just like the one in Fig. 2; we measure the time it takes to service $1M$ requests, each running for 500µs and isolate the overhead of the

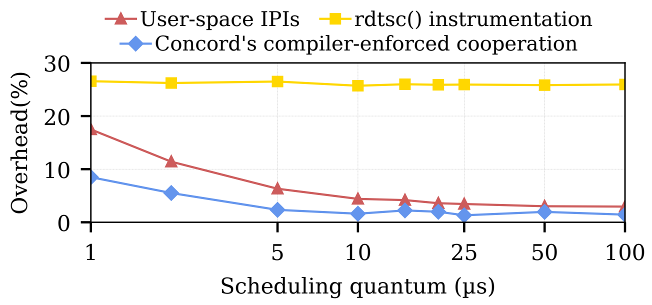


Figure 15. Comparing the overhead of Concord's compiler-enforced cooperation to Intel's new user-space IPIs.

preemption mechanism by running the application with no-op preemption handlers. We use a 192-core Intel Xeon Platinum 8648 Sapphire Rapid server to compare the mechanisms.

Fig. 15 demonstrates the results. We observe that Concord's compiler-enforced cooperation outperforms even the most recent hardware support and imposes a throughput overhead that is ≈ 2× lower. This is unsurprising, since no matter how fast interrupts get, they will always be slower than Concord's reads and writes to a shared cache line which will remain the fastest way for two cores to communicate on shared memory hardware. Note, the absolute value of Concord's overhead is slightly higher on this machine as compared to the machines we used in our evaluation and for Fig. 2. This is because of the large number of cores, which makes cache coherence misses approximately 1.5× more expensive. That said, the *relative* overhead of Concord with respect to UIPIs will remain the same even on Sapphire Rapid machines with fewer cores, since sending an interrupt also requires writing to memory-mapped registers and thus relies on the same cache coherence mechanisms as Concord's compiler-enforced cooperation.

To summarize, our evaluation demonstrates that Concord improves application throughput while preserving the tail-latency properties of the state of the art for a wide range of service time distributions. While Concord is particularly effective—18-83% higher throughput than the state of the art—for service time distributions that benefit from preemptive scheduling (Figures 6, 7, 9, 10) it continues to perform favorably even for service times that only require single queue scheduling (Fig. 8). Finally, each of Concord's mechanisms contributes to its overall throughput improvements (Figures 11, 12) and will remain useful even as datacenter servers provide increasing hardware support for microsecond-scale scheduling (Fig. 15).

6 Discussion

Limitations: Concord has two main limitations:
First, it requires the application source code to be available and written in a compiled language with an LLVM backend. We believe that access to source code should not be an issue for developers deploying Concord on bare metal, or for tenants deploying their low-latency systems on VMs in the public cloud. That said, the current Concord prototype is ill-suited to being a runtime provided by public cloud providers for their tenants since this would require that all tenants share their code with the cloud provider. We plan to explore replacing Concord's source code instrumentation with approaches that directly instrument the binary to overcome this limitation.

Second, the current Concord prototype is restricted to single-dispatcher systems. This will not be a limitation for low CPU count, e.g. small VMs, but the single dispatcher can become a bottleneck as the number of CPUs increases and the service time (as well as service time variability) decreases. In such cases, replication, i.e. creating multiple single-dispatcher instances that feed disjoint sets of cores, or trading off throughput for tail-latency, e.g. using batching, can help improve scalability [36].

How Concord extends to single-*logical*-queue systems: Concord's compiler-enforced cooperation and work conserving dispatcher can enable low-overhead preemption even in systems that implement a single logical queue such as Shenango [46] and Caladan [22]. This is because compiler-enforced cooperation only requires the dispatcher to monitor the elapsed time and does not require it to maintain the single queue. Such a system would also overcome the throughput bottleneck of a single dispatcher.

Incorporating the above mechanisms will require a dedicated hyperthread (referred to as the "scheduler" henceforth) that only monitors whether a worker thread has been processing a request for longer than the scheduling quantum. Some single-logical-queue systems (e.g., Caladan [22]) already have such a scheduler thread. When the quantum has elapsed, the scheduler writes to the cache line, and the worker—instrumented to poll for it—stops processing the current request. Since the dispatch of a request is not synchronous with when a worker begins processing it, the worker must start the timer at the beginning of the quantum, but this is already the case with Concord's asynchronous dispatch for JBSQ (§3.2). Finally, the scheduler can steal requests safely using `rdtsc()` instrumentation (§3.3) since it is likely to be idle for extended periods.

Broader use of cooperative preemption: We believe Concord's compiler-enforced cooperation mechanism can be used as a replacement for IPIs in many settings beyond scheduling. For instance, any periodic event, such as garbage collection and timer management or Unix signals and global synchronization mechanisms, implemented through `membarrier()` on Linux or `FlushProcessWriteBuffers` on Windows, can eschew IPIs in favor of compiler-enforced preemption. Compiler-enforced preemption can also be used in deployments where IPIs are not available or untrusted. This is the case for confidential VMs [2, 31] in which the hypervisor is considered potentially malicious and can inject virtual interrupts at any point in time.

7 Related Work

Having already discussed the closest related work—Shinjuku, Persephone, Compiler Interrupts—in detail, we do not do so again here.

Microsecond-scale schedulers deployed by major cloud providers (e.g., ghOSt from Google [28]) typically have lower raw performance than academic ones (e.g., ghOSt aims to be within 5% of Shinjuku's maximum sustainable throughput). However this is because they prioritize requirements that academic schedulers (including Concord) do not; for instance, the ability to simultaneously support multiple tenants with different scheduling policies and the ability to provide fault isolation. In this work, we focused on extracting maximum raw performance from a single application with Concord. We plan to pursue extending Concord to support the above constraints as immediate future work.

The key ideas underlying Concord's compiler-enforced co-operation, namely cooperative scheduling and inter-core communication using dedicated cache lines, are well known. For instance, cooperative scheduling and user-level threading go back to the seminal paper on Scheduler Activations [4] and are widely used in different contexts such as language runtimes, e.g. goroutines [25], and modern thread library implementation for the datacenter, e.g. Arachne [51]. Similarly, using dedicated cache lines instead of interrupts or barriers to enable low-overhead core-to-core communication is widely used for high-performance computing [1, 24]. That said, the key difference between such prior work and Concord is that the former relies on the programmer to correctly insert both yield points and reads/writes to cache lines, while Concord does so automatically using a compiler pass.

While programming language approaches have been used extensively in the context of memory isolation [29, 47], splitting CPU time using such approaches is more challenging. Lilt [60] introduces a new language to statically enforce timing policies and the Erlang scheduler [21] depends on the underlying language virtual machine implementation to count the number of executed instructions and preempt Erlang processes in a timely manner. Finally, Libringer [10] introduces the abstraction of a preemptible function but depends on Unix signals to implement it, thus incurring high overheads for µs-scale tasks.

8 Conclusion

We presented Concord: a runtime that demonstrates that *approximating* tail-optimal scheduling policies can lead to significant throughput benefits for µs-scale applications with negligible tail-latency penalties. Concord relies on three key mechanisms to reduce overhead—co-operative preemption instead of interrupts, JBSQ(2) scheduling to eliminate idle time and automatic code instrumentation to safely run application logic on the dispatcher thread. We evaluated Concord using microbenchmarks and Google's LevelDB key-value store on a wide variety of service time distributions from academia and industry. Compared to the state of the art, Concord improves application throughput by up to 52% on microbenchmarks and by up to 83% on LevelDB, with the same tail-latency characteristics. Unlike the state of the art, Concord does not rely on custom hardware or application-level assumptions, so it can be deployed today in the public cloud for a wide range of applications.

9 Acknowledgements

We thank our shepherd Timothy Roscoe and the anonymous reviewers for their detailed feedback that significantly improved the paper. We are also grateful to Ed Bugnion and Katerina Argyraki for their helpful feedback on drafts of the paper at various stages.

References

[1] Aldinucci, M., Danelutto, M., Kilpatrick, P., and Torquati, M. Fastflow: High-Level and Efficient Streaming on Multicore. In *Journal on Programming Multi-Core and Many-Core Computing Systems* (2017).

[2] AMD SEV-SNP. https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf. [Last accessed on 2023-08-03].

[3] Amvrosiadis, G., Park, J. W., Ganger, G. R., Gibson, G. A., Baseman, E., and DeBardeleben, N. On the Diversity of Cluster Workloads and its Impact on Research Results. In *USENIX Annual Technical Conference* (2018).

[4] 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. In *Symposium on Operating Systems Principles* (1991).

[5] Arashloo, M. T., Lavrov, A., Ghobadi, M., Rexford, J., Walker, D., and Wentzlaff, D. Enabling Programmable Transport Protocols in High-Speed NICs. In *Symposium on Networked Systems Design and Implementation* (2020).

[6] Atikoglu, B., Xu, Y., Frachtenberg, E., Jiang, S., and Paleczny, M. Workload Analysis of a Large-Scale Key-Value Store. In *ACM SIGMETRICS Conference* (2012).

[7] Barroso, L. A., Marty, M., Patterson, D. A., and Ranganathan, P. Attack of the Killer Microseconds. *Communications of the ACM* (2017).

[8] Basu, N., Montanari, C., and Eriksson, J. Frequent Background Polling on a Shared Thread, Using Light-Weight Compiler Interrupts. In *International Conference on Programming Language Design and Implementation* (2021).

[9] Belay, A., Prekas, G., Klimovic, A., Grossman, S., Kozyrakis, C., and Bugnion, E. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *Symposium on Operating Systems Design and Implementation* (2014).

[10] Boucher, S., Kalia, A., Andersen, D. G., and Kaminsky, M. Lightweight Preemptible Functions. In *USENIX Annual Technical Conference* (2020).

[11] Cao, Z., Dong, S., Vemuri, S., and Du, D. H. C. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *USENIX Conference on File and Storage Technologies* (2020).

[12] Cao, Z., Tarasov, V., Raman, H. P., Hildebrand, D., and Zadok, E. On the Performance Variation in Modern Storage Stacks. In *USENIX Conference on File and Storage Technologies* (2017).

[13] Cloudlab. https://cloudlab.us. [Last accessed on 2023-08-03].

[14] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. Benchmarking Cloud Serving Systems with YCSB. In *Symposium on Cloud Computing* (2010).

[15] Daglis, A., Sutherland, M., and Falsafi, B. RPCValet: NI-Driven Tail-Aware Balancing of μs-Scale RPCs. In *International Conference on Architectural Support for Programming Languages and Operating Systems* (2019).

[16] Dalton, M., Schultz, D., Arefin, A., Docauer, A., Gupta, A., Fahs, B. M., Rubinstein, D., Zermeno, E. C., Rubow, E., Adriaens, J., Alpert, J. L., Ai, J., Olson, J., DeCabooter, K. P., de Kruijf, M. A., Hua, N., Lewis, N., Kasinadhuni, N., Crepaldi, R., Krishnan, S., Venkata, S., Richter, Y., Naik, U., and Vahdat, A. Andromeda: Performance, Isolation, and Velocity at Scale in Cloud Network Virtualization. In *Symposium on Networked Systems Design and Implementation* (2018).

[17] David, T., Guerraoui, R., and Trigonakis, V. Everything You Always Wanted to Know About Synchronization but Were Afraid to Ask. In *Symposium on Operating Systems Principles* (2013).

[18] Dean, J., and Barroso, L. A. The Tail at Scale. In *Communications of the ACM* (2013).

[19] Demoulin, H. M., Fried, J., Pedisich, I., Kogias, M., Loo, B. T., Phan, L. T. X., and Zhang, I. When Idling is Ideal: Optimizing Tail-Latency for Heavy-Tailed Datacenter Workloads with Perséphone. In *Symposium on Operating Systems Principles* (2021).

[20] DPDK: Data Plane Development Kit. https://dpdk.org. [Last accessed on 2023-08-03].

[21] Erik Stenman. The Beam Book. https://blog.stenmans.org/theBeamBook. [Last accessed on 2023-08-03].

[22] Fried, J., Ruan, Z., Ousterhout, A., and Belay, A. Caladan: Mitigating Interference at Microsecond Timescales. In *Symposium on Operating Systems Design and Implementation* (2020).

[23] Gan, Y., Zhang, Y., Cheng, D., Shetty, A., Rathi, P., Katarki, N., Bruno, A., Hu, J., Ritchken, B., Jackson, B., Hu, K., Pancholi, M., He, Y., Clancy, B., Colen, C., Wen, F., Leung, C., Wang, S., Zaruvinsky, L., Espinosa, M., Lin, R., Liu, Z., Padilla, J., and Delimitrou, C. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems* (2019).

[24] Giacomoni, J., Moseley, T., and Vachharajani, M. FastForward for Efficient Pipeline Parallelism: A Cache-Optimized Concurrent Lock-Free Queue. In *Symposium on Principles and Practice of Parallel Programming* (2008).

[25] Goroutines. https://go.dev/tour/concurrency. [Last accessed on 2023-08-03].

[26] Heavy-Tailed Distributions. https://en.wikipedia.org/wiki/Heavy-tailed_distribution. [Last accessed on 2023-08-03].

[27] Humphries, J. T., Kaffes, K., Mazières, D., and Kozyrakis, C. Mind the Gap: A Case for Informed Request Scheduling at the NIC. In *ACM Workshop on Hot Topics in Networks* (2019).

[28] 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 *Symposium on Operating Systems Principles* (2021).

[29] Hunt, G., and Larus, J. Singularity: Rethinking the Software Stack. *Operating Systems Review* (2007).

[30] Ibanez, S., Mallery, A., Arslan, S., Jepsen, T., Shahbaz, M., Kim, C., and McKeown, N. The nanoPU: A Nanosecond Network Stack for Datacenters. In *Symposium on Operating Systems Design and Implementation* (2021).

[31] Intel TDX. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html. [Last accessed on 2023-08-03].

[32] Iyer, R., Argyraki, K., and Candea, G. Performance Interfaces for Network Functions. In *Symposium on Networked Systems Design and Implementation* (2022).

[33] Iyer, R., Pedrosa, L., Zaostrovnykh, A., Pirelli, S., Argyraki, K., and Candea, G. Performance Contracts for Software Network Functions. In *Symposium on Networked Systems Design and Implementation* (2019).

[34] Iyer, R. R., Ma, J., Argyraki, K. J., Candea, G., and Ratnasamy, S. The case for performance interfaces for hardware accelerators. In *Workshop on Hot Topics in Operating Systems* (2023).

[35] Jin, G., Song, L., Shi, X., Scherpelz, J., and Lu, S. Understanding and detecting real-world performance bugs. In *International Conference on Programming Language Design and Implementation* (2012).

[36] Kaffes, K., Chong, T., Humphries, J. T., Belay, A., Mazières, D., and Kozyrakis, C. Shinjuku: Preemptive Scheduling for μsecond-scale Tail Latency. In *Symposium on Networked Systems Design and Implementation* (2019).

[37] Kalia, A., Kaminsky, M., and Andersen, D. G. Datacenter RPCs can be General and Fast. In *Symposium on Networked Systems Design and Implementation* (2019).

[38] Karandikar, S., Leary, C., Kennelly, C., Zhao, J., Parimi, D., Nikolic, B., Asanovic, K., and Ranganathan, P. A Hardware Accelerator for Protocol Buffers. In *International Symposium on Microarchitecture* (2021).

[39] Kaufmann, A., Stamler, T., Peter, S., Sharma, N. K., Krishnamurthy, A., and Anderson, T. E. TAS: TCP acceleration as an OS service. In *ACM European Conference on Computer Systems* (2019).

[40] Kogias, M., Prekas, G., Ghosn, A., Fietz, J., and Bugnion, E. R2P2: Making RPCs First-Class Datacenter Citizens. In *USENIX Annual Technical Conference* (2019).

[41] Lessons Learned From Scaling Uber To 2000 Engineers, 1000 Services, And 8000 Git Repositories. http://highscalability.com/blog/2016/10/12/lessons-learned-from-scaling-uber-to-2000-engineers-1000-ser.html. [Last accessed on 2023-08-03].

[42] LevelDB. https://github.com/google/leveldb. [Last accessed on 2023-08-03].

[43] Marty, M., de Kruijf, M., Adriaens, J., Alfeld, C., Bauer, S., Contavalli, C., Dalton, M., Dukkipati, N., Evans, W. C., Gribble, S. D., Kidd, N., Kononov, R., Kumar, G., Mauer, C., Musick, E., Olson, L. E., Rubow, E., Ryan, M., Springborn, K., Turner, P., Valancius, V., Wang, X., and Vahdat, A. Snap: A Microkernel Approach to Host Networking. In *Symposium on Operating Systems Principles* (2019).

[44] Mauro, T. Adopting Microservices at Netflix: Lessons for Architectural Design. https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices, 2015. [Last accessed on 2023-08-03].

[45] Misra, P. A., Borge, M. F., Goiri, I., Lebeck, A. R., Zwaenepoel, W., and Bianchini, R. Managing Tail Latency in Datacenter-Scale File Systems Under Production Constraints. In *ACM European Conference on Computer Systems* (2019).

[46] Ousterhout, A., Fried, J., Behrens, J., Belay, A., and Balakrishnan, H. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *Symposium on Networked Systems Design and Implementation* (2019).

[47] Panda, A., Han, S., Jang, K., Walls, M., Ratnasamy, S., and Shenker, S. NetBricks: Taking the V out of NFV. In *Symposium on Operating Systems Design and Implementation* (2016).

[48] The PARSEC benchmark suite. https://parsec.cs.princeton.edu. [Last accessed on 2023-08-03].

[49] PHOENIX Benchmark Suite. https://github.com/kozyraki/phoenix. [Last accessed on 2023-08-03].

[50] Prekas, G., Kogias, M., and Bugnion, E. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In *Symposium on Operating Systems Principles* (2017).

[51] Qin, H., Li, Q., Speiser, J., Kraft, P., and Ousterhout, J. K. Arachne: Core-Aware Thread Management. In *Symposium on Operating Systems Design and Implementation* (2018).

[52] Shortest Remaining Processing Time (SRPT) Scheduling. https://en.wikipedia.org/wiki/Shortest_remaining_time. [Last accessed on

2023-08-03].

[53] SPLASH-2 Benchmark Suite. https://github.com/staceyson/splash2. [Last accessed on 2023-08-03].

[54] Sutherland, M., Gupta, S., Falsafi, B., Marathe, V. J., Pnevmatikatos, D. N., and Daglis, A. The NEBULA RPC-Optimized Architecture. In *International Symposium on Computer Architecture* (2020).

[55] TCP Offload Engine (TOE). https://www.chelsio.com/nic/tcp-offload-engine. [Last accessed on 2023-08-03].

[56] The Biggest Thing Amazon Got Right: The Platform. https://old.gigaom.com/2011/10/12/419-the-biggest-thing-amazon-got-right-the-platform/. [Last accessed on 2023-08-03].

[57] The TPC-C OLTP benchmark. http://www.tpc.org/tpcc. [Last accessed on 2023-08-03].

[58] Tu, S., Zheng, W., Kohler, E., Liskov, B., and Madden, S. Speedy Transactions in Multicore In-Memory Databases. In *Symposium on Operating Systems Principles* (2013).

[59] Benchmarking Methodology for Networking Interconnect Devices. http://www.rfc-editor.org/rfc/rfc2647.txt. [Last accessed on 2023-08-03].

[60] Vanderwaart, C. J. Static Enforcement of Timing Policies Using Code Certification. http://reports-archive.adm.cs.cmu.edu/anon/2006/abstracts/06-143.html, 2006. [Last accessed on 2023-08-03].

[61] Wierman, A., and Zwart, B. Is Tail-Optimal Scheduling Possible? In *Journal of Operating Research* (2012).

[62] x86 Support for User Interrupts. https://lwn.net/Articles/869140/. [Last accessed on 2023-08-03].

[63] Zaostrovnykh, A., Pirelli, S., Iyer, R. R., Rizzo, M., Pedrosa, L., Argyraki, K. J., and Candea, G. Verifying Software Network Functions with No Verification Expertise. In *Symposium on Operating Systems Principles* (2019).

[64] Zarandi, A. P., Sutherland, M., Daglis, A., and Falsafi, B. Cerebros: Evading the RPC Tax in Datacenters. In *International Symposium on Microarchitecture* (2021).

[65] Zhang, I., Raybuck, A., Patel, P., Olynyk, K., Nelson, J., Leija, O. S. N., Martinez, A., Liu, J., Simpson, A. K., Jayakar, S., Penna, P. H., Demoulin, M., Choudhury, P., and Badam, A. The Demikernel Datapath OS Architecture for Microsecond-Scale Datacenter Systems. In *Symposium on Operating Systems Principles* (2021).