Efficient Microsecond-scale Blind Scheduling with Tiny Quanta

Zhihong Luo
UC Berkeley
USA

Sam Son
UC Berkeley
USA

Dev Bali
UC Berkeley
USA

Emmanuel Amaro
VMware Research
USA

Amy Ousterhout
UC San Diego
USA

Sylvia Ratnasamy
UC Berkeley
USA

Scott Shenker
UC Berkeley & ICSI
USA

Abstract

A longstanding performance challenge in datacenter-based applications is how to efficiently handle incoming client requests that spawn many very short (μs scale) jobs that must be handled with high throughput and low tail latency. When no assumptions are made about the duration of individual jobs, or even about the distribution of their durations, this requires blind scheduling with frequent and efficient preemption, which is not scalably supported for μs-level tasks.

We present Tiny Quanta (TQ), a system that enables efficient blind scheduling of μs-level workloads. TQ performs fine-grained preemptive scheduling and does so with high performance via a novel combination of two mechanisms: forced multitasking and two-level scheduling. Evaluations with a wide variety of μs-level workloads show that TQ achieves low tail latency while sustaining 1.2x to 6.8x the throughput of prior blind scheduling systems.

CCS Concepts: • **Software and its engineering → Scheduling**; *Coroutines*; *Automated static analysis.*

Keywords: Blind scheduling, microsecond scale, tail latency

ACM Reference Format:
Zhihong Luo, Sam Son, Dev Bali, Emmanuel Amaro, Amy Ousterhout, Sylvia Ratnasamy, and Scott Shenker. 2024. Efficient Microsecond-scale Blind Scheduling with Tiny Quanta. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3620665.3640381

1 Introduction

Network requests from clients to datacenter-based applications commonly produce fan outs of many requests (in some cases, thousands) to internal services. Today, these internal requests often have a service time of a few microseconds [6]; thus, efficiently scheduling μs-level requests with low tail latency and high throughput is essential to maintain end-to-end latency service level objectives (SLOs) for these applications [20, 50]. The presence of these very short jobs, while the overall service time distribution remains broad [7, 17, 62], has made efficient scheduling more difficult because it is now more likely that very short requests end up queuing behind long ones, resulting in what is called head-of-line blocking.

To minimize the occurrence of head-of-line blocking, some state-of-the-art scheduling systems have assumed knowledge of either individual requests' service times or the overall service time distribution [21]. However, such assumptions limit the system's generality and the system must be modified to adapt to new request types or evolving workloads. Therefore, we do not assume knowledge of either individual request times or their distribution in this work. This results in what is often referred to as "blind" scheduling. Our paper considers the problem of blindly scheduling μs-level workloads with broad service time distributions. (If the service time distribution were not broad, so most jobs had similar durations, then FCFS-like scheduling would suffice [52, 54, 59].)

To avoid head-of-line blocking, a blind scheduler must preempt currently running jobs in order to allow jobs queued behind them a chance to get service, without having any notion of how much more processing the current job or the queued job will require. Given this lack of knowledge, typically scheduling systems are designed around the notion of processor sharing [3, 25, 53]; that is, there is some minimal quantum of service that they allocate to jobs before they preempt them and provide service to the next job. Since there are inherent delays in switching between jobs, there is a tension between making the quantum size small so as to minimize head-of-line blocking, but large enough so that the frequency of preemption – with its resulting overhead – does not reduce the ability of the system to achieve high throughput. To calibrate the discussion below, we consider what it would take to maintain high throughput with sub-5μs quanta. This goal presents three main challenges.

First, previous approaches that use blind scheduling with small quanta relied on preemptive thread multitasking, which

ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA

ACM ISBN 979-8-4007-0385-0/24/04.
https://doi.org/10.1145/3620665.3640381

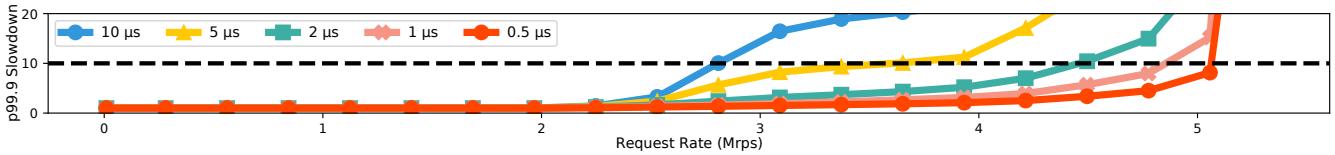

Figure 1. 99.9% slowdown with different quantum sizes, with no preemption overhead.

incurs excessive switching overhead for small quanta. In particular, even with an optimized interrupt system enabled by non-standard virtualization features, Shinjuku [34], a μs-scale preemptive scheduling system, still has a $\approx$1μs thread interrupt latency, which causes severe throughput degradations when operating at sub-5μs quanta. An alternative approach to preemption is cooperative multitasking, where jobs leverage light execution contexts and explicitly yield back to a scheduler. The main challenge of this approach is to efficiently ensure that jobs relinquish control after consuming roughly a quantum of service.

The second challenge in a system that executes sub-5μs quanta is to make its scheduler architecture scalable. Scalability is required because the total work a scheduler performs to pick the next job to run increases linearly with the number of preemptions, which roughly increases as the inverse of the quantum size. Shinjuku [34] has a CPU that runs a centralized scheduler, which becomes the throughput bottleneck as it needs to receive network requests, load-balance requests across multiple cores (*e.g.,* 16), trigger interrupts and schedule jobs at sub-5μs quanta, and send network replies.

Third, as we interleave job executions at a finer granularity, CPU caches can become polluted and result in additional overheads. Prior work on ms-level job scheduling has shown that cache misses due to frequent context switching induce overhead comparable to the switching itself [37]. However, there has been little research on cache behaviors when preemptions happen at μs scale. Thus, the third challenge is to understand the cache behaviors with sub-5μs quanta and ensure that cache pollution does not become a bottleneck.

To meet these challenges, we present a system called Tiny Quanta (TQ) that leverages sub-5μs quanta to blindly schedule μs-level jobs with wide service time distributions. In particular, TQ achieves low latency and high throughput and solves the challenges above with the following key ideas.

First, TQ uses coroutines as execution contexts and a compiler pass that inserts probe points to implement what we call *forced multitasking*. Using the compiler to generate probes frees developers from manually instrumenting yield points, while taking advantage of the low switching cost of coroutines. Moreover, TQ strategically places physical-clock based probes far apart to achieve significantly lower probing overhead, compared with prior compiler instrumentation techniques based on instruction counters [2, 8, 10].

To solve the scalability challenge, TQ uses a two-level scheduler architecture that consists of two components: a global job dispatcher that load-balances jobs across cores on admission, and a per-CPU job scheduler that interleaves jobs' quanta. The two-level scheduler design avoids the throughput bottleneck of a centralized scheduler by distributing a job's scheduling policy across two cores. For each CPU, TQ uses the blind policy of processor sharing across a CPU's admitted jobs. For the job dispatcher, TQ adopts the join-the-shortest-queue (JSQ) load balancing policy with a new tie-breaking heuristic that improves the latency of long jobs.

Lastly, compared with centralized scheduling, TQ's two-level scheduler architecture also ensures better cache locality by having each job reside in a single CPU throughout its execution. With a set of carefully designed microbenchmarks that study μs-scale data cache behaviors, we show that scheduling with small quanta in TQ is unlikely to cause notable performance degradation due to cache pollution.

We compare TQ to two state-of-the-art systems that support μs-scale blind scheduling: Shinjuku [34] and Caladan [27]. With a wide range of μs-level workloads, we show that TQ, by efficiently scheduling with tiny quanta, can sustain 1.2x to 6.8x the load of prior systems, while maintaining low tail latency. Moreover, we show that the proposed forced multitasking and two-level scheduling mechanisms play a vital role in achieving TQ's superior performance.

The contributions of this paper are: (i) a forced multitasking mechanism that enables cheap preemptions at small quanta by strategically placing physical-clock based probes; (ii) a two-level scheduling mechanism that allows fine-grained scheduling in a scalable and cache friendly manner; and (iii) a synthesis of these mechanisms into a solution that achieves low tail latency and high throughput for μs-level tasks without assuming knowledge about tasks' service times.

2 The Case for Tiny Quanta and the Need for Low Overhead

In this section, we quantify the desirability of using small, or tiny, quanta on a system that schedules jobs blindly. We consider the optimistic case where there is no overhead in preemption. Our point is to argue that if we could reduce the overhead sufficiently, then reducing the quantum size is of value. As we shall see later, TQ achieves a low enough overhead so that a quantum size of 1μs is practical (compared to the 5μs or larger quanta that are supported by Shinjuku [34]).

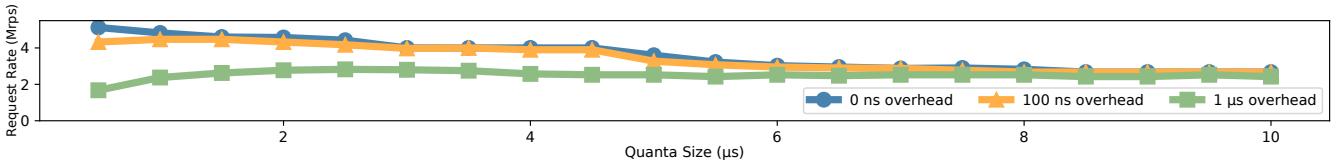

Figure 2. Maximum request rates until a slowdown of 10 is reached, for different preemption overheads and quantum sizes.

To make our argument, we simulate an "extreme bimodal" workload from prior work [21, 54]: a mix of 99.5% small requests with a service time of 0.5μs, and 0.5% long requests with a service time of 500μs. Our simulation includes 5 seconds of Poisson requests arriving to a 17-core system, where one core serves as a centralized scheduler that schedules jobs among other 16 cores using a processor sharing policy [67]. We measure the metric of slowdown; *i.e.,* the server-side service time to execute a job relative to its service time when the job runs to completion [31]. Figure 1 shows the 99.9% tail slowdown with varying quantum sizes. As expected, smaller quanta lead to less slowdown due to reduced head-of-line blocking of small (0.5μs) jobs. In contrast, Shinjuku [34] only supports quanta of at least 5μs and, as the chart shows, that leads to larger slowdowns at higher loads.

To see the importance of reducing overheads, Figure 2 shows the maximum achievable request rate while keeping the 99.9% slowdown under 10 (denoted in Figure 1 by the horizontal line), for three different values of preemption overheads: 0ns, 100ns, and 1μs. If we focus on the 0ns curve, we see that smaller quanta enable the system to run at higher loads: about 40% higher compared with 5μs quanta. However, when the overhead is 100ns, this benefit is reduced, and in fact decreasing the quanta below 1μs reduces the system capacity. With an overhead of 1μs, any reduction below 3μs reduces the capacity. Thus, this graph shows that reducing the quanta allows a system to sustain higher loads with low tail latency, but only if the overheads are sufficiently small.

The rest of this paper is devoted to meeting the three challenges: reducing the overhead, making scheduling scalable, and ensuring that the caches do not become a bottleneck.

3 Design

As shown in Figure 3, TQ consists of two main components: a *dispatcher* distributing jobs among cores, and *workers* scheduling and executing job quanta on their dedicated cores. Requests get processed as follows. First, the dispatcher polls incoming packets from the NIC. Second, the dispatcher directly forwards each request to a worker with a load balancing policy, according to its view of each worker's load. Third, each worker schedules and executes quanta of their jobs with a blind scheduling policy. Fourth, once a worker finishes a job, it sends out the response without going through the dispatcher. Lastly, the dispatcher obtains from each worker statistics regarding its load for future load balancing.

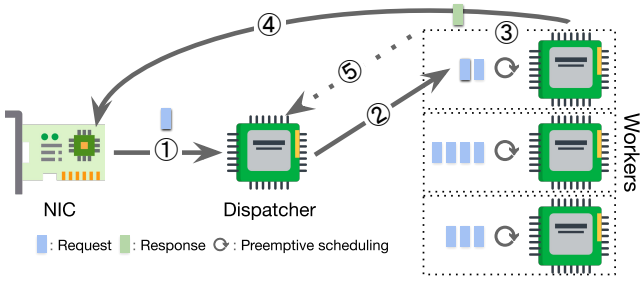

Figure 3. TQ: system diagram.

TQ differs from prior preemptive scheduling systems in two ways: (i) workers independently perform quantum scheduling without external interrupts, and (ii) the dispatcher performs only job load balancing. These differences correspond to the two mechanisms of TQ that we will now describe: forced multitasking (§3.1) and two-level scheduling (§3.2).

3.1 Forced multitasking

To switch between jobs with low overhead, TQ leverages coroutine yields. Compared with hardware interrupts, coroutine yields are orders of magnitude cheaper: as a user-space mechanism within a single process, coroutine yield requires no system calls nor changes to virtual memory mappings. Stackful coroutines yield in tens of nanoseconds [14], whereas stackless coroutines can yield within a single digit of nanoseconds [23, 49]. Besides having lower overhead, coroutine yields also differ from hardware interrupts in terms of being cooperative – the switching is initiated voluntarily within each task rather than relying on externally generated interrupts. This however introduces the question of how to ensure that job coroutines all yield in a timely fashion.

The standard practice of having developers manually insert yield points [19, 46, 68] is suboptimal. Firstly, it is cumbersome for developers to do so, especially given that μs-level jobs are designed to have few obvious yield points like blocking I/O and system calls. Secondly, it is hard and error-prone for developers to decide where to insert yield points in order to achieve an accurate interval between consecutive yields.

Compiler-instrumented yielding: To free developers from having to instrument their code, TQ automatically analyzes and instruments the provided code at compile time. The instrumented code can then be used by TQ at run time to enable

fine-grained multitasking. TQ thus takes care of ensuring the yielding of instrumented jobs, without burdening either developers or system operators. Note that TQ assumes the availability of application code for instrumentation.

The challenge here is how to place yield points so that a running job does not exceed its quanta. Even with advanced static analysis, predicting the execution time of a series of instructions is challenging, primarily because of instructions like memory loads that can take a varied duration of time [8]. To address this issue, TQ *delays* the placement of yield points until run time. This is done by making a distinction between *probe* points and yield points. Probe points are placed at locations where yields can *potentially* happen, but the decision to actually yield or not is made only at run time.

The compiler inserts probe points at a sufficient density (see below) so that the program is capable of yielding often enough to not exceed the quantum, for all quanta above a given minimum size. At run time, the target quantum is specified in units of cycles and whenever a probe point is reached the probe then checks whether enough cycles have passed since the previous yield point; if so, it calls the yield function, otherwise it resumes the operation. Such an approach also supports dynamic quantum sizes, which are needed for scheduling policies like least-attained-service (LAS) [51, 55].

The key challenge here is to introduce only minimal probing overhead, while ensuring a reasonable preemption timing accuracy. Inserting more probes leads to better accuracy, but also more wasted probing cycles. If too many probing cycles are spent on finding the appropriate coroutine yield timings, we still end up with an expensive multitasking mechanism.

Prior approach: Prior compiler-instrumentation work uses an *instruction-counter* based approach, where the compiler maintains an instruction counter with probes – each probe increments the counter by some number that is determined at the compile time. The most common method is to instrument a probe at the end of every basic block, which increments the counter by the number of instructions of that basic block [10]. At run time, one needs to first translate the target quantum (in terms of cycles) into a target instruction count, and then the instrumented probes yield if the value of the instruction counter is larger than the target instruction count.

The instruction-counter based approach is fundamentally inaccurate because of this translation from cycles into instruction counts; using either a default or a profiled instruction-to-cycle ratio leads to poor accuracy as instructions can take varied durations [8]. More importantly, while a single instruction-counter probe is cheap, instruction-counter based approaches can introduce significant overhead. This is because one has to insert a large number of probes to maintain the *correctness* of the counter (because, for any execution path, the instruction counter needs to be incremented roughly by the actual number of instructions along that path). Since a basic block often contains only a few instructions, inserting a probe in every basic block is expensive. Recent work [8, 10] tries to reduce the number of probes by extracting (single-entry single-exit) SESE structures [33, 47] and instrument probes only at the exits of these structures. Such optimizations fail if the program has complex structures because the compiler ends up instrumenting at the granularity of single basic blocks. In fact, we measured that the state-of-the-art implementation [8] of the instruction-counter based approach introduces a 60% probing overhead to a RocksDB GET operation (*i.e.,* the instrumented GET takes 60% longer to finish), as it adds over 1000 probes for this 2μs job.

```
1  extern __thread uint64_t last_yield_ts, target_cycle;
2  extern __thread void (*call_the_yield)();
3  inline void probe() {
4    uint64_t cur_ts = rdtsc();
5    if(cur_ts - last_yield_ts >= target_cycle) {
6      call_the_yield();
7      last_yield_ts = rdtsc();
8    }
9  }
```

Listing 1. Pseudocode illustrating how TQ's instrumented probes decide whether to yield based on physical clocks.

Our approach: Instead, TQ proposes a different approach based on using physical clocks: each probe instead reads the hardware cycle counter (*e.g.,* RDTSC in x86)[1] and yields if enough cycles have passed since the previous yielding point. At first glance, using physical clocks only worsens the issue: a single RDTSC instruction can take 20 to 40 cycles [26], much more expensive than an instruction counter, which uses only ADD instructions. However, it turns out that by placing these physical-clock based probes strategically, we can achieve much lower overhead with better accuracy, due to two desirable properties of physical clock:

- **Placement flexibility:** Unlike the instruction-counter based method, where probes have to be inserted frequently to maintain the correctness of the counter, physical-clock probes can function correctly in arbitrary program locations. This allows us to place probes further apart, resulting in much fewer probes. For the same GET operation, we only instrument 40 probes, 30 times fewer than the instruction-counter based approach. Since RDTSC's latency is often partially overlapped with other instructions due to out-of-order execution, a 30x reduction in the number of probes leads to substantially lower overhead.
- **Timing accuracy:** Since a physical clock is inherently more accurate than instruction count, we can achieve accurate timings despite sparser placements.

To exploit the placement flexibility and timing accuracy of a physical clock, TQ instruments a small set of probes that bound the maximum number of instructions of any execution paths between two probes. Specifically, if the longest execution path between two probes is greater than the bound,

[1] TQ assumes the availability of hardware cycle counters that can be accessed by software. Besides x86, there are other architectures that also meet this requirement: *e.g.,* ARM with cyccnt, RISC-V with rdtime.

a probe will be inserted along that path, and this process is repeated until the longest path is shorter than the bound. Note that TQ uses instruction counts only for controlling the probe density whereas the yield timing is determined by physical clock readings, so that unlike instruction-counter based approaches, it does not suffer from inaccurate yield timings due to cycle-to-instruction translations. For loops, unless the number of iterations can be statically deduced, TQ (by default) instruments an iteration counter and only invokes the probe when the counter reaches a target number of iterations, which is calculated by dividing the target bound with the number of instructions of the longest uninstrumented path in the loop body. TQ applies various optimizations to reduce the loop instrumentation overhead when feasible. For instance, if an induction variable is found, TQ will invoke probes based on the variable, saving the cost of maintaining an iteration counter. For nested loops that consist of a single basic block, TQ will clone the loop into two versions (*i.e.,* original and instrumented), and select the one to use based on the runtime iteration count: the uninstrumented version is invoked if the iteration count is under the target, so that TQ can bypass the instrumentations in this case. For function calls, if they are to uninstrumented functions (*e.g.,* system calls or external libraries), since the compiler does not know the execution paths within that function, TQ adds an additional instruction cost for the function call.

By placing physical-clock based probes far apart while bounding the maximum distance between them as described above, TQ's compiler pass achieves much lower probing overhead and better timing accuracy than the state-of-the-art instruction-counter based approach (§5.6). This lower probing overhead leads to to better performance when scheduling μs-level workloads (§5.4). Therefore, by combining efficient compiler-instrumented probings and cheap coroutine yields, TQ's forced multitasking mechanism allows switching between tasks with extremely small overhead, which is critical for efficient scheduling of small quanta.

3.2 Two-level scheduling

TQ's scheduling framework needs to simultaneously meet two requirements. First, it should be highly *scalable*, specifically, the scheduler itself should not be the throughput bottleneck as we reduce the quantum sizes. Second, the scheduler should be capable of intelligently distributing workloads across cores to achieve low latency and high throughput.

Prior approach: Existing systems generally adopt a centralized scheduling framework. In this framework, the dispatcher processes incoming packets into pending jobs, maintains a centralized queue of pending or preempted jobs and schedules jobs' quanta among worker cores following some scheduling policy. Such a centralized scheduling approach allows the dispatcher to steer quanta to different cores with a global view, hence resulting in good scheduling performance. However, such an approach has limited scalability. Since

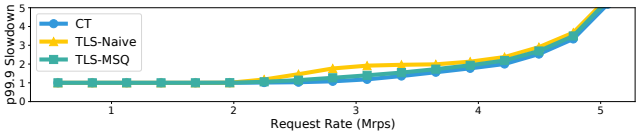

Figure 4. Centralized (CT) vs. two-level scheduling (TLS) with different policies: 99.9% slowdown for the long job of the Extreme Bimodal workload, with no preemption overhead.

the dispatcher is responsible for preempting and scheduling quanta for all the cores, its load increases reversely proportional to the quantum sizes. For Shinjuku, as the quantum size goes down from 5μs, its dispatcher becomes incapable of preempting and scheduling a large number of cores (§5.6). In this case, to maintain the centralized scheduling framework, one has to increase the number of dispatcher cores. However, it is hard to efficiently maintain a centralized queue among multiple dispatcher cores, leading to wasted cycles [34].

Our approach: One important reason why prior work adopts a centralized scheduling framework, despite the scheduler being a potential bottleneck, is that they rely on hardware interrupts to multiplex jobs [34, 70]. Since the interrupt has to be initiated externally, the dispatcher core is thus responsible for triggering interrupts and hence scheduling quanta. In TQ, however, job multiplexing is enabled with forced multitasking, which does not require the assistance of an external core. Moreover, since TQ focuses on blind scheduling, TQ's dispatcher does not need to parse packets for job information that is needed for quantum scheduling (*e.g.,* job type) [21, 52]. These opportunities lead us to a two-level scheduling framework, where the dispatcher only performs a minimal set of work while still achieving good scheduling performance.

Specifically, in TQ, the dispatcher only performs job load balancing, whereas quantum scheduling is offloaded to each worker core. When a request arrives, the dispatcher directly forwards it to a core according to some blind load balancing policy (see below). The job then stays in the core, where its quanta will be scheduled by a per-core scheduler coroutine. When the job finishes, the worker core directly sends a response back to the client without going through the dispatcher and updates statistics used for the dispatcher's load balancing decisions, *e.g.,* incrementing a counter to reflect the total number of finished jobs. The dispatcher checks the statistics of each core to have an accurate view of its load.

In terms of scalability, the dispatcher only performs operations for load balancing and hence can sustain a high throughput. Moreover, since these operations are performed at the job granularity, the dispatcher's load largely remains constant with smaller quanta. This prevents the dispatcher from being a bottleneck for fine-grained scheduling (§5.6).

In terms of scheduling performance, the performance of two-level scheduling depends on the combination of workers' quantum scheduling policy and dispatcher's load balancing policy. For quantum scheduling, TQ's workers adopt

the processor sharing (PS) policy, which is provably optimal in terms of tail sojourn time for broad/heavy-tailed service time distributions [16, 64]. Sojourn time refers to the duration between when a job arrives and when it finishes. For load balancing, TQ's dispatcher adopts the Join-the-Shortest-Queue (JSQ) policy, because (i) it is a blind policy that can be implemented with minimal overhead (§4), and (ii) the JSQ-PS combination, formally denoted as a M/G/K/JSQ/PS model, is provably near-optimal for the mean sojourn time [29, 58].

However, there is no theoretical work on the tail sojourn time of the JSQ-PS combination. We thus study the tail performance of JSQ-PS via both simulations and system evaluations. One interesting finding is that the *tie-breaking policy* of JSQ can substantially impact the latency of long jobs. Considering the "extreme bimodal" workloads (with 0.5μs short jobs and 500μs long jobs) used in §2, at a medium load, there is usually a long job on each core. In this case, naive (*i.e.,* random) tie-breaking suffices for short jobs, because a short job finishes in a single quantum and will thus receive similar treatment on any core that has the smallest number of jobs. However, with naive tie breaking, the dispatcher may assign long jobs to a core with a long job that has a large *remaining* processing time, which results in a larger slowdown. To address this issue, TQ leverages a tie-breaking policy that we call Maximum-Serviced-Quanta (MSQ). With MSQ, the dispatcher breaks a tie by picking the core that has serviced the largest number of quanta for its current jobs, with the expectation that this core has small remaining processing time. MSQ can be efficiently implemented (§4) and improves the tail latency of long jobs: in Figure 4, we simulate the 99.9% slowdown of the long job for centralized (CT) and two-level scheduling (TLS), and show that compared with centralized PS scheduling, JSQ-PS with MSQ tie breaking achieves competitive performance, much better than random tie breaking. While CT performs better in simulation *without* factoring in the preemption overhead, TLS can be efficiently supported, allowing TQ to significantly outperform Shinjuku [34], a centralized preemptive scheduling system (§5.3).

Besides being highly scalable and delivering good scheduling performance, another benefit of two-level scheduling is better *cache locality*. Specifically, centralized scheduling distributes quanta among cores and causes cache misses due to the requested cache lines being in a different core. In contrast, with two-level scheduling, each job naturally resides in the same core throughout its execution. This leads to an efficient use of the private caches (*i.e.,* L1, L2 cache) and thus a better cache performance for μs-level workloads (§5.5).

4 Implementation

We implemented TQ's compiler pass and scheduling runtime with 2200 and 1400 lines of C code respectively.[2]

Compiler-instrumented yielding: We implemented TQ's forced multitasking with Boost coroutine [57], which has a 20 to 40 ns yield time [15]. We implemented TQ's probe instrumentation as a LLVM compiler pass [35]. The instrumentation process starts by compiling the provided code into LLVM IR. We then analyze and transform this initial LLVM IR with standard passes like LoopSimplify and ScalarEvolution [41]. TQ's compiler pass then takes in the simplified IR, inserts physical-clock based probes and outputs an instrumented IR, which finally gets compiled into object files. As illustrated in Listing 1, TQ's instrumented probes call a thread-local function pointer (*i.e., call_the_yield*) when they decide to yield. However, each Boost coroutine has a specific yield function that is generated at the coroutine construction. Therefore, before resuming a task coroutine, the scheduler coroutine binds the *call_the_yield* function to the yield function of that task coroutine, so that the task coroutine can correctly yield back to the scheduler coroutine.

Networking: TQ implements a user-space network stack with DPDK [24], which has one RX queue for the dispatcher to poll for requests and one TX queue for each worker core to push out responses. A multi-producer, single-consumer memory pool is used for RX buffers so that worker cores can independently release parsed buffers back to the pool.

Dispatcher: To support JSQ load balancing, TQ's dispatcher keeps track of the number of unfinished jobs of each worker. It does so by computing the difference between the number of jobs it has assigned to a worker and the number of jobs the worker has finished, which it knows by reading the counter maintained by each core. The dispatcher also reads the number of serviced quanta of each core to implement MSQ tie breaking. Note that the sizes of these worker-side counters do not impose upper limits on the number of finished jobs or serviced quanta for each worker: the worker increments its counters regardless of overflows, and the dispatcher keeps track of the total number by computing the delta between its reads. When a request arrives, the dispatcher forwards it to the least loaded worker via a lockless ring buffer.

Workers: Each worker thread starts with a scheduler coroutine, which initializes a set of task coroutines. The scheduler coroutine keeps track of idle and busy coroutines. When there are idle coroutines, the scheduler coroutine polls the dispatch queue to see whether there are pending requests from the dispatcher. If so, the scheduler coroutine parses the request, marks an idle coroutine as busy and resumes it to execute this request. After the task coroutine has run for a quantum specified by the scheduler coroutine, it yields to the scheduler, who decides the next coroutine to resume based on its scheduling policy. To emulate PS, the scheduler maintains a queue for busy coroutines, enqueues yielded coroutines at the tail and resumes the coroutine at the head. If a task coroutine finishes a request, the scheduler pushes the response to the per-core TX queue, increments the counter of finished

[2]TQ is available at https://github.com/zhluo94/TinyQuanta.

Workloads	Request	Runtime (μs)	Ratio
Extreme Bimodal	Short	0.5	99.5%
	Long	500	0.5%
High Bimodal	Short	1	50%
	Long	100	50%
TPC-C	Payment	5.7	44%
	OrderStatus	6	4%
	NewOrder	20	44%
	Delivery	88	4%
	StockLevel	100	4%
EXP	N/A	1 (mean)	N/A
RocksDB (0.5% SCAN)	GET	1.2	99.5%
	SCAN	675	0.5%
RocksDB (50% SCAN)	GET	1.2	50%
	SCAN	675	50%

Table 1. The list of evaluated workloads.

jobs, and marks the coroutine as idle. The scheduler also updates the number of serviced quanta of its current jobs. Both counters (*i.e.,* for finished jobs and serviced quanta) reside in a cache line that is periodically read by the dispatcher. One could extend the implementation and have the dispatcher and each worker communicate in multiple cache lines, if they wanted to add features that require more information exchanges between the dispatcher and workers.

Critical section: We currently disable preemptions during critical sections by providing jobs with specific functions to call at the entrances and exits of these sections like Shinjuku [34]. Under the hood, these functions set/unset a flag that makes the *call_the_yield* function bypass its yielding.

5 Evaluation

In this section, we present our evaluation setup (§5.1) and investigate key questions regarding TQ: (i) how small of quanta can TQ support? (§5.2), (ii) how well does TQ perform compared to prior systems that support blind scheduling? (§5.3), (iii) how do different components of TQ contribute to its performance? (§5.4), (iv) how do caches perform with small quanta in TQ? (§5.5), and (v) how do forced multitasking and two-level scheduling perform? (§5.6). We answer (i), (ii) and (iii) by evaluating TQ with μs-level workloads, (iv) and (v) by testing the specific mechanism with microbenchmarks.

5.1 Evaluation setup

Workloads: We evaluate a wide range of μs-level workloads from prior work [21, 34, 54], which consist of both synthetic and real workloads. For synthetic workloads, High Bimodal and Extreme Bimodal workloads exhibit broad service time distributions and test how well TQ prevents head-of-line blockings. The TPC-C workload represents a multi-modal service time distribution in standardized OLTP models [61]. The Exp(1) workload has an exponential service time distribution with a mean of 1μs. For real workloads, we evaluate an in-memory key-value store built over RocksDB [44], with different ratios of SCAN operations (0.5% and 50%) and test if TQ can ensure low latency of GETs at high throughput.

Systems: We evaluate TQ and two state-of-the-art systems that support blind scheduling: Caladan [27] and Shinjuku [34]. For TQ, unless stated otherwise, we use 2μs quanta, each scheduler coroutine schedules its task coroutines in a PS fashion and the dispatcher performs JSQ load balancing with MSQ tie breaking. We observe similar performance with more than four task coroutines per worker core and we use eight for the evaluation. Caladan implements a FCFS scheduling policy by steering packets to worker cores using RSS hashes and having worker cores run jobs to completion. Worker cores perform work stealing to minimize load imbalance. In the default mode, Caladan has an IOKernel core interacting with the NIC. The alternative is the directpath mode, where worker cores directly send/receive their packets to/from the NIC. This eliminates the throughput bottleneck of the IOKernel but incurs packet processing overhead to the worker cores. We thus evaluate Caladan under both modes and report the better one for each workload. Shinjuku uses the virtualization features of Dune [9] to preempt worker cores with interrupts. It supports scheduling with quanta as small as 5μs. We evaluate Shinjuku's single-queue scheduling policy, which effectively emulates PS scheduling. Since Shinjuku becomes unstable when having to preempt jobs very frequently, we follow the practice of prior work [21, 34] and use different quantum sizes that lead to the optimal performance of Shinjuku for different workloads: 5μs for Extreme Bimodal and High Bimodal, 10μs for TPC-C and Exp(1) and 15μs for RocksDB, and report Shinjuku's performance under these quantum sizes.

Client: We adapted the open-loop load generator from [27] that transmits requests under a Poisson process centered at the workload's average service time over UDP. For each request rate, the experiment runs for 10 seconds and the first 10% samples are discarded to remove warm-up effects. Our client setup is identical for all systems.

Latency metrics: We measured two forms of latency metrics: (i) end-to-end latency, which is recorded by the client and includes the network round trip time, and (ii) sojourn time, which is calculated by the server as the time elapsed from the dispatcher receiving an incoming request to when the server finishes executing the job. We use end-to-end latency for all the comparisons between systems and sojourn time only for highlighting the effects of different configurations within TQ. Focusing on the tail latency, we report the 99.9 percentile of the measured latency. For the TPC-C workload, we also report the overall slowdown, as it helps calibrate the differences in durations of multiple job types.

Testbed: We conduct experiments using two dual-socket servers with 28-core Intel Xeon Platinum 8176 CPUs operating at 2.1 GHz. We use one of the servers for TQ and Caladan, equipped with a 40 Gbits/s Mellanox Connect X-5 Bluefield NIC, and the other for Shinjuku, equipped with a 10 Gbits/s Intel 82599ES NIC that is compatible with Shinjuku's network stack. Note that network bandwidth was not

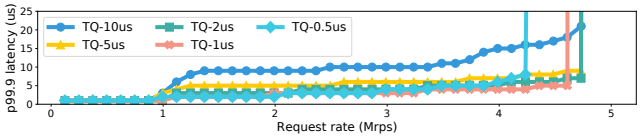

Figure 5. Extreme bimodal with different Qs – SHORT.

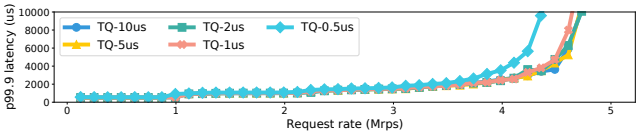

Figure 6. Extreme bimodal with different Qs – LONG.

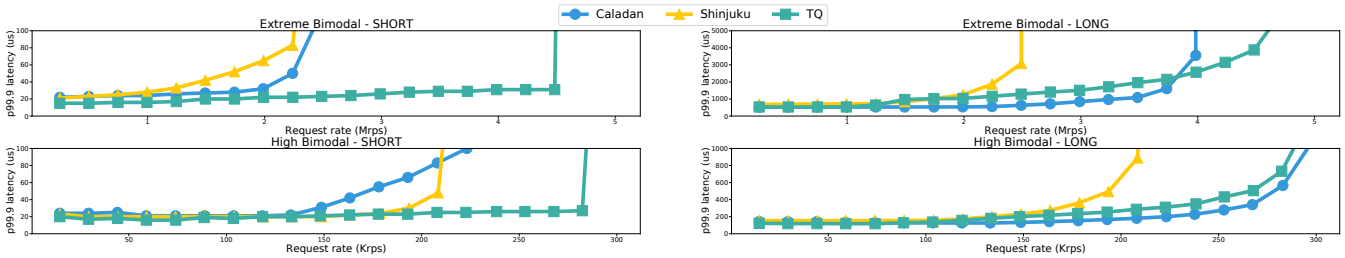

Figure 7. Extreme Bimodal and High Bimodal workloads.

a bottleneck in our evaluation, with less than 10% utilization. Caladan and TQ run on Ubuntu 22.04 with Linux kernel version 5.15.0. Shinjuku runs on Ubuntu 18.04 with Linux kernel version 4.4.0. Shinjuku uses one hyperthread for the net worker and another for the dispatcher, collocated on the same physical core. Caladan runs its IOKernel on a single core, and TQ runs its dispatcher on a single core. All systems use 16 worker threads on dedicated physical cores.

5.2 Benefits of small quanta

As shown in §2, scheduling with smaller quanta leads to lower latency, and if a system does so with sufficiently low overhead, it can simultaneously maintain a high throughput. We thus would like to know whether TQ is efficient enough to support quantum sizes much smaller than prior systems?

To answer this question, we evaluate TQ with the Extreme Bimodal workload and vary the quantum sizes from 10μs down to 0.5μs and measure the 99.9 percentile latency under different request rates. The results are shown in Figures 5 and 6. For the short jobs, TQ not only achieves lower latency with smaller quanta, but is also able to maintain the same maximum throughput as 10μs quanta with a quantum size as small as 2μs. In fact, TQ is able to maintain significant throughput until a quantum size as small as 0.5μs. Similar trends can be seen on the long jobs as well, where the throughput remains almost identical for quantum sizes larger than 0.5μs. Such results indicate that (i) TQ's forced multitasking mechanism is sufficiently cheap even for quantum sizes as small as 1μs, and (ii) TQ's two-level scheduling framework can schedule 0.5μs quanta without being the throughput bottleneck.

5.3 Comparison with Caladan and Shinjuku

Next, we evaluate TQ's performance and show that it achieves **1.2x** to **6.8x** of the throughput of Shinjuku and Caladan, while maintaining low latency for all the workloads.

Extreme Bimodal: The Extreme Bimodal workload has a dispersion ratio (*i.e.,* the ratio between the runtimes of long and short jobs) of 1000, the largest among all the evaluated workloads. Figure 7 shows the tail latencies of the short and long jobs for all three systems. For Caladan, while the system can sustain high throughput for the long jobs, because of using a FCFS policy, it achieves poor latency for short jobs due to severe head-of-line blocking. For Shinjuku, it was not able to sustain high throughput due to the large preemption overhead, resulting in poor performance for both short and long jobs, despite having low latency at small load. In contrast, TQ maintains a smaller than 50μs latency of the short jobs until a request rate of 4.5Mrps, which is **2.6x** of Shinjuku's and **2.1x** of Caladan's. For long jobs, TQ sustains **1.8x** and **1.2x** the throughput of Shinjuku and Caladan respectively due to having lower overhead. Caladan achieves lower latency than TQ for the long jobs at medium load, which is expected as FCFS prioritizes the scheduling of long jobs.

High Bimodal: Compared with the Extreme Bimodal workload, the High Bimodal workload has a lower request rate and dispersion ratio. As shown in Figure 7, Shinjuku is able to achieve higher throughput under 50μs end-to-end latency budget for the short jobs compared with Caladan, due to performing preemptive scheduling. With a target 50μs end-to-end latency for short jobs, TQ sustains **1.65x** and **1.33x** the throughput of Caladan and Shinjuku respectively. For long jobs, TQ is able to sustain the highest throughput, while maintaining similar latency as Shinjuku. Caladan achieves a lower latency for long jobs again due to its FCFS policy.

TPC-C: TPC-C allows us to observe how each system prioritizes jobs of different sizes. Between Caladan and Shinjuku, as shown in Figure 8, Shinjuku achieves lower latency for short jobs because of using preemptive scheduling; whereas Caladan achieves higher throughput for long jobs, because Shinjuku pays a large preemption overhead. TQ gets the best

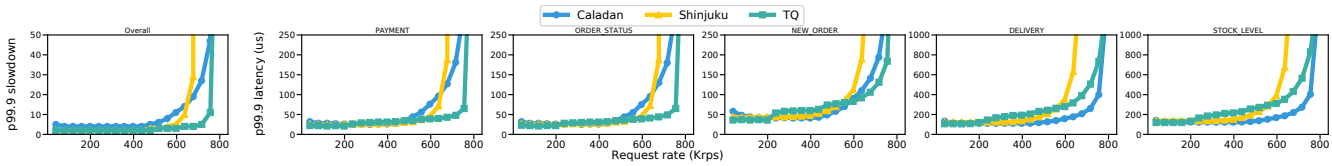

Figure 8. TPCC – overall slowdown and latency of different request types.

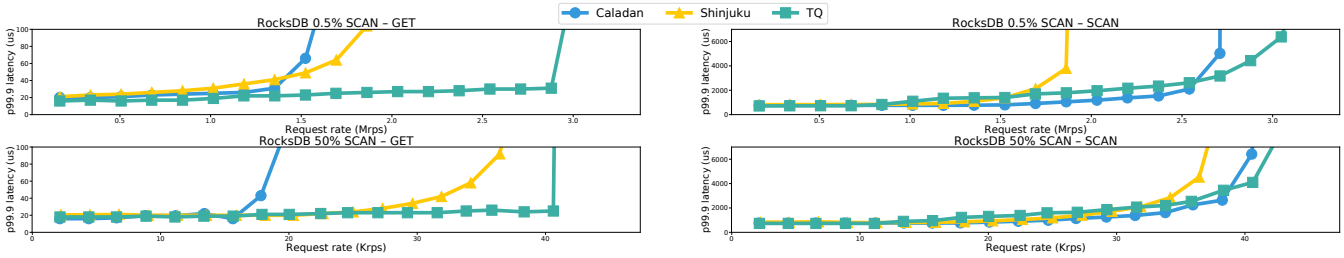

Figure 9. RocksDB workloads with different ratios of SCAN operations.

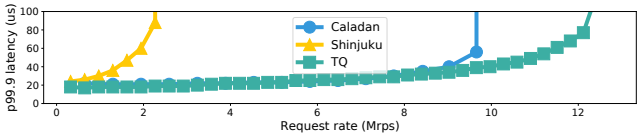

Figure 10. EXP(1) workload.

of both worlds – by supporting fine-grained scheduling with very small overhead, it achieves both low latency for short jobs and high throughput for longer ones. Overall, under a 10x overall slowdown budget, TQ achieves a throughput that is **1.29x** of Caladan's and **1.18x** of Shinjuku's.

RocksDB: Results for RocksDB workloads are shown in Figure 9. For the workload with 0.5% SCAN operations, with a 50μs latency budget for GET operations, TQ achieves **1.93x** and **2.07x** the throughput of Shinjuku and Caladan respectively. For the workload with 50% SCAN, despite the gap being smaller due to lower request rates, TQ achieves a throughput that is **1.28x** of Shinjuku's and **2.21x** of Caladan's, under a 50μs latency budget of GET operations. For the SCAN operations, despite having higher latency at medium loads, TQ manages to sustain **1.63x** and **1.13x** the throughput of Shinjuku for workloads with 0.5% and 50% SCAN respectively.

Exponential: Lastly, we evaluate the Exp(1) workload, which is a challenging workload as it has a broad continuous service time distribution as well as a high request rate. As shown in Figure 10, by efficiently scheduling small quanta, TQ is able to sustain 11Mrps with a smaller than 50μs end-to-end latency, which is **6.85x** of Shinjuku's and **1.21x** of Caladan's.

5.4 Breakdown of TQ's performance

Next, we create different variants of TQ by altering specific parts of the design and measure their performance in comparison to TQ. The differences thus directly inform us how components of TQ contribute to its performance. The results for RocksDB with 0.5% SCAN are shown in Figures 11 and 12.

Forced multitasking: As elaborated in §3.1, TQ's forced multitasking leverages the combination of (i) efficient probings and (ii) cheap coroutine yields to achieve lower overhead. Moreover, it achieves accurate preemption timings by probing with a physical clock. We thus evaluate three variants: TQ-IC, where the state-of-the-art instruction-counter based instrumentation technique [8] is used instead of TQ's compiler pass; TQ-SLOW-YIELD, where a 1μs delay is added to the coroutine yield; and TQ-TIMING, where we emulate inaccurate preemption timings with 1μs quanta for GET and 3μs quanta for SCAN. The results show that both components have significant impacts on TQ's performance. For TQ-IC, with a 50μs latency budget for the GET operation, it achieves only 62% of TQ's throughput due to the large probing overhead. This shows that an efficient instrumentation technique is crucial to TQ's high performance for μs-level workloads, and thus we later evaluate TQ's compiler pass with a large set of benchmark applications (§5.6). For TQ-SLOW-YIELD, it achieves 81% of TQ's throughput, and the gap increases for workloads that require more preemptions (*e.g.,* RocksDB with 50% SCAN). This shows that low switching cost is essential for efficient fine-grained scheduling. While inaccurate preemption timings do not hurt TQ as much as large probing or yielding overhead does, as shown with TQ-TIMING, it can still degrade performance. In particular, with a 50μs latency budget for GETs, TQ-TIMING achieves only 81% of TQ's throughput, as GET operations now receive worse head-of-line blocking due to having inaccurately small quanta.

Two-level scheduling: Two-level scheduling brings high scalability and good cache locality, which we will evaluate in §5.5 and §5.6. Here we focus on the JSQ-PS policy with three variants: TQ-RAND, TQ-POWER-TWO and TQ-FCFS. The first two adopt different load balancing policies: random and

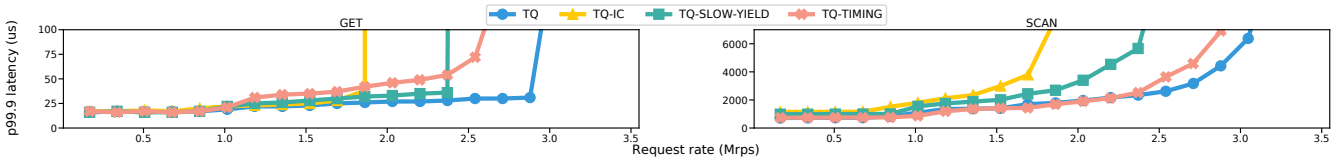

Figure 11. RocksDB 0.5% SCAN with different variants of TQ for forced multitasking.

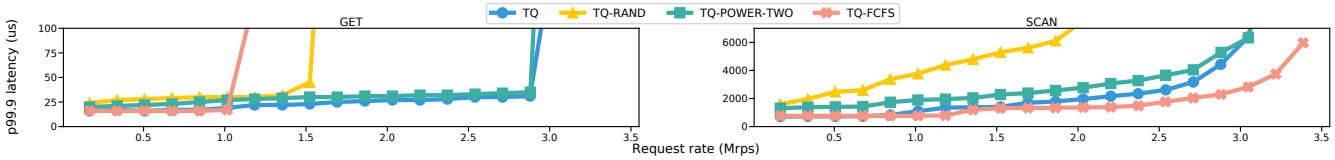

Figure 12. RocksDB 0.5% SCAN with different variants of TQ for two-level scheduling.

power-of-two choices [45], whereas the last one uses FCFS as its quantum scheduling policy. The results show that JSQ-PS is critical for TQ's performance. For the load balancing policy, TQ-RAND achieves only 53% of TQ's throughput under a 50μs latency budget for GET operations. This is due to the load imbalances caused by random dispatching and shows the value of a smart dispatcher. TQ-POWER-TWO achieves similar throughput but incurs higher latency than TQ. This matches with understandings from theoretical work [39, 45]: between JSQ and power-of-two load balancing, JSQ delivers better scheduling performance whereas it is easier to support power-of-two at a large scale. Since JSQ can be efficiently implemented in TQ (§4), it is a better choice. As for the scheduling policy, similar to Caladan, TQ-FCFS achieves only 34% of TQ's throughput for GET operations due to severe head-of-line blocking. While it achieves lower latency for SCAN due to FCFS prioritizing long jobs, PS allows TQ to better meet latency SLOs for μs-level workloads.

5.5 Cache behavior at microsecond scale

To understand μs-scale cache behavior, we investigate two questions: (i) how do smaller quanta in TQ affect cache behavior? and (ii) how does two-level scheduling (TLS) impact cache performance compared to centralized scheduling (CT)?

5.5.1 Experiment setup. To study cache behavior under μs-scale preemptions for workloads with different footprints, we run experiments on the following synthetic workload:
Workload: We create an array of configurable size, fix a random element iteration order, and iterate through the array 100K times in that order via pointer chasing. We repeat this experiment for array sizes ranging from 1KB to 1MB.

This workload is well suited for studying data cache behavior at μs-scale. First, iterating over arrays multiple times exhibits strong *intra-job* locality, which can be affected by preemptions [36, 48]. Specifically, a cache line that was accessed by a preempted job is likely to be already evicted when the job resumes and re-accesses it. Second, random pointer chasing amplifies cache misses. When a preempted job re-accesses a cache line that has been evicted due to other jobs' accesses, if a sequential access pattern is adopted, the cache line is likely prefetched by the hardware after the job resumes, which effectively *conceals* the negative effects of preemptions. In contrast, with random pointer chasing, (i) hardware prefetching is ineffective and (ii) cache miss latency is fully exposed. This thus allows us to clearly observe how preemptions can hurt cache performance. Lastly, varying array sizes emulates μs-level workloads with different intra-job locality. We focus on intra-job locality as it is directly affected by preemptions. Inter-job locality, where the previous access of a cache line happened in earlier jobs, is not investigated. As shown later, even for jobs like RocksDB SCAN, there is a substantial amount of intra-job locality.
Methodology: We emulate two-level and centralized scheduling as follows. For each of the 16 cores, we have a thread that (i) accesses X elements via random pointer chasing of an array and (ii) saves the progress for the current array, switches to the next array and repeats (i). Step (i) emulates a job iterating over its array in a quantum. X is set to match with the target quantum size. Step (ii) emulates switching to a different job (and thus a different array) for the next quantum. By emulating scheduling frameworks instead of invoking specific mechanisms, we are able to study cache pollution that stems from interleaving job executions.

Arrays iterated by each core depend on the scheduling framework. For TLS, each core switches between its own set of arrays (*e.g.*, arrays 0-3 for core 0). This matches with how each job resides in a single core in TLS. For CT, arrays are shared among all the cores and cores iterate over each array on a rotating basis. This captures how quanta of a job can be executed by different cores in CT. We set 4 arrays per core (64 in total) to emulate 4 jobs per core, a high degree of concurrency occuring when TQ operates under heavy loads.

We report the average pointer access latency to study the cache behaviors. For question (i), we show the access latency of TLS with different quantum sizes. For question (ii), we compare the access latency of CT with TLS's at 2μs quanta.

Scheduling framework	The first access of the element within the quantum?	
	Yes	No
CT	C * J * A	A
TLS	J * A	A

Table 2. Reuse distances of accesses in array iterations for centralized (CT) and two-level scheduling (TLS); C: number of worker cores, J: number of jobs per core, A: array size.

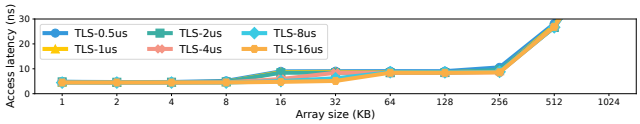

Figure 13. Access latency for TLS with different quanta.

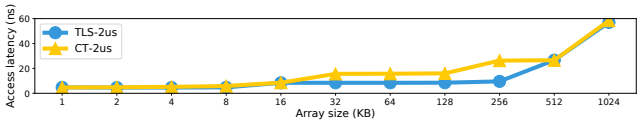

Figure 14. Access latency for TLS vs. CT.

5.5.2 Experiment results. To methodically interpret our results, we first present an analysis based on *reuse distance*, a metric widely used for studying cache behavior [11, 22, 40]:
Reuse distance analysis: Reuse distance of a memory access refers to the number of distinct accesses between the previous access to the same address and the current access. For example, when iterating over an array multiple times, all accesses starting from the second iteration have a reuse distance of the array size. For a fully associative LRU cache of capacity C, an access with a reuse distance R results in a cache hit if R is smaller than C, cache miss if otherwise. Despite modern CPU caches not being fully associative and adopting complex replacement policies, this property still largely holds for cache capacity misses.

Table 2 shows how smaller quanta and scheduling frameworks affect reuse distances of our workloads. With preemptions, the reuse distance of an access depends on whether it is the first access to this element *within the quantum*: the answer is yes if the prior access happens in a previous quantum, and the reuse distance is *amplified* beyond the array size. Therefore, if the quantum is small w.r.t. array iteration time, more accesses will fall into this category and have amplified reuse distances, likely causing cache misses. Moreover, since the amplification ratio equals to the number of jobs sharing the cache, for L1/L2 caches, TLS bounds the ratio by the number of jobs per core (4 in our case), whereas the ratio for CT is the total number of concurrent jobs (64 in our case).
Effects of smaller quanta: Figure 13 shows TLS's access latency for different quanta. There are two findings, both consistent with our analysis. First, smaller quanta cause extra cache misses *only* for 8-32KB arrays. This is because: (i) with a 32KB L1 cache and 8-32KB arrays, accesses with amplified (with a ratio of 4) reuse distances cause L1 misses, whereas

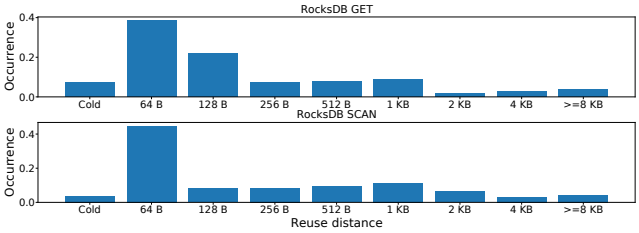

Figure 15. Reuse distances for RocksDB GET and SCAN.

other accesses are L1 hits; and (ii) the iteration time of 8-32KB arrays is on par with the evaluated quanta, and hence varying quanta makes a *notable* difference in the ratio of accesses with amplified distances. As a result, while TLS-16μs has mostly L1 hits for array sizes up to 32KB, TLS-2μs receives L1 misses once arrays are larger than 8KB. Note that TLS-2μs does not cause extra L2 misses compared with TLS-16μs for arrays larger than 256KB (with a 1MB L2 cache). This is because condition (ii) no longer holds: even 16μs is too short w.r.t. the iteration time of 256KB arrays, and thus most accesses of TLS-16μs suffer from amplified distances for these arrays, in the same way as TLS-2μs. Second, for sufficiently small quanta, further shrinking their sizes *does not* worsen the cache performance: TLS-0.5μs behaves similarly as TLS-2μs, because even for small 8KB arrays, 2μs is short enough that most accesses already have amplified distances.

These findings indicate that small quanta are unlikely to cause notable performance degradation due to cache pollution: (i) additional cache misses occur only for accesses with certain reuse distances, and (ii) even with these accesses, the latency of resulting L1 misses can mostly be hidden by out-of-order executions of processors in real workloads [1, 60].
TLS vs. CT: Figure 14 compares the access latency of TLS and CT for 2μs quanta. Consistent with our analysis, CT introduces additional cache misses due to a higher reuse distance amplification ratio, which in turn stems from CT having more concurrent jobs per core. Specifically, with a ratio of 64, CT has L2 cache misses starting from an array size of 16KB (16KB*64=1MB), whereas TLS does not incur L2 cache misses for arrays smaller than 256KB.
Implications to real workloads: While real workloads involve more complex access patterns, one can infer the impacts of smaller quanta or scheduling frameworks with our experiment results. Specifically, one can estimate the cache behaviors received by a workload based on the reuse distances of its accesses [22, 40]. For instance, if a workload has most reuse distances smaller than 8KB, it is expected to be largely insensitive to reducing quantum sizes (Figure 13). Figure 15 shows the histograms of reuse distances for RocksDB GET and SCAN, which we measure with MICA [13], an open-source Pin tool. Both jobs have very few accesses with reuse distances that are sensitive to quantum size changes: only 3.7% and 4.5% out of all accesses have reuse distances larger

	Overhead (%)			MAE (cycles)		
	CI	CICY	TQ	CI	CICY	TQ
water-nsquared	2.97	4.55	5.86	743	635	432
water-spatial	3.78	6.22	7.42	501	456	408
ocean-cp	6.96	7.43	5.05	1966	1502	617
ocean-ncp	5.44	6.91	4.98	905	606	774
barnes	15.47	17.29	13.31	351	371	304
volrend	15.99	17.41	9.06	2766	972	698
fmm	4.44	4.61	4.38	3136	3027	496
raytrace	3.17	5.27	2.77	1259	1012	661
radiosity	38.69	48.32	20.27	664	672	593
radix	0.43	0.45	0.42	1126	1212	770
fft	7.39	7.63	7.22	2255	1558	1858
lu-c	21.21	23.51	17.31	2613	2063	322
lu-nc	25.74	29.42	22.24	979	749	418
cholesky	32.53	32.24	14.07	1619	1608	845
reverse-index	25.38	26.03	16.32	8685	8479	3023
histogram	7.94	8.41	7.58	4297	3989	757
kmeans	30.42	31.04	11.72	842	858	749
pca	86.92	87.11	27.16	923	912	598
matrix-multiply	5.96	6.41	2.51	5817	5312	1875
string-match	45.81	49.92	12.86	368	376	359
linear-regression	39.17	39.82	16.61	513	522	503
word-count	8.23	9.18	6.28	1037	917	773
blackholes	1.47	2.09	4.57	1578	1562	1103
fluidanimate	2.96	2.99	2.44	6169	6061	2319
swaptions	26.64	34.22	18.80	991	621	466
canneal	6.52	6.91	3.73	4721	4712	2311
streamcluster	4.81	5.86	6.05	461	311	341
mean	**17.65**	**19.30**	**10.05**	**2122**	**1891**	**902**

Table 3. Comparison among CI, an instruction-counter based mechanism [8], CI-Cycles, a hybrid variant with CI-gated physical clock checking [8], and TQ's compiler pass; MAE: mean average error.

than 8KB. This matches with our evaluation that the cache behaviors of both jobs are similar across different quanta.

5.6 Performance of TQ's components

Next, we evaluate with a microbenchmark (i) whether TQ's physical clock based approach consistently outperforms prior instruction-counter based approaches and (ii) whether two-level scheduling allows TQ to easily scale to small quanta.

Compiler instrumentation: We measure the probing overhead and yield timing accuracy of TQ's compiler pass with a set of workloads used by prior work [8]. These workloads come from SPLASH-2 [66], Phoenix [56] and Parsec [12] benchmark applications. They encompass various program structures and hence were used to test the completeness of the instrumentation technique. Our results from running these workloads on a single core with a target quantum size of 2μs are summarized in Table 3. Compared with Compiler Interrupt (CI) [8], the state-of-the-art instruction-counter based instrumentation technique, TQ incurs lower probing overhead while achieving better timing accuracy for 22 out of the total 26 workloads. For the other 4 workloads, it increases their overhead by less than 3%. In contrast, it significantly reduces the overhead of several workloads that have large probing overhead with CI, *e.g., string-match* from 45% to 12%, *cholesky* from 33% to 14% and *kmeans* from 30% to 12%. As elaborated in §3.1, the reason for this substantial reduction of probing overhead is that unlike CI, which instruments a large number of probes to maintain the correctness of its instruction counter, TQ places much sparser physical-clock based probes. In fact, TQ introduces 25x, 59x and 47x less probes than CI, for the *string-match*, *cholesky* and *kmeans* workloads respectively. Moreover, TQ achieves a better yield timing accuracy thanks to the accuracy of physical clocks. On average, compared with CI, TQ's compiler pass reduces the probing overhead by 43% and the mean average error (MAE) of yield timings by 57%. Note that the key to the superior performance of TQ's compiler pass is not only the decision to use a physical clock, but more importantly the way TQ strategically places these probes. To see this, we evaluate CI-Cycles, a hybrid variant that checks the physical clock when the CI instruction counter is over the threshold [8]. With the *same* probe placement as CI, CI-Cycles inherits some drawbacks of instruction-counter based probing: it further increases the already high probing overhead of CI, and still delivers worse timing accuracy than TQ as the physical clock checking is only invoked when instruction counters reach thresholds. Both using physical clocks, such a dramatic difference in the performance of TQ and CI-Cycles thus shows the importance of strategic probe placements.

Two-level scheduling: Next, we evaluate whether two-level scheduling allows TQ to easily schedule in small quanta. For this we have the client generate a workload consisting of only 1-ms jobs and Shinjuku and TQ try to schedule with some target quantum sizes. Note that we use long 1-ms jobs to minimize the packet processing cost of the dispatcher and focus on the overhead of quantum scheduling. For each quantum size, we then check the maximum number of cores that Shinjuku's and TQ's dispatchers can support. Specifically, we compute the average quantum size scheduled by the dispatcher, and if it is more than 10% larger than the target, we conclude that the dispatcher is unable to keep up with the target quantum size at this number of cores, so we reduce the number of cores. We repeat this process until we find the number of cores that the dispatcher can sustain for the target quantum size. The results for target quantum sizes from 5μs to 0.5μs are shown in Figure 16. Shinjuku is unable to keep up with 16 cores as soon as the quantum size drops from 5μs to 3μs, and is only capable of maintaining 3 cores if the target quantum size is 0.5μs. In contrast, TQ's dispatcher performs operations at the job granularity, hence reducing quantum sizes does not increase its load. TQ's dispatcher is thus not a bottleneck for scheduling in small quanta.

6 Limitations and Discussion

Dispatcher throughput: Due to the simplicity of its design – not parsing incoming requests to make forwarding decisions nor maintaining a queue, TQ's dispatcher achieves

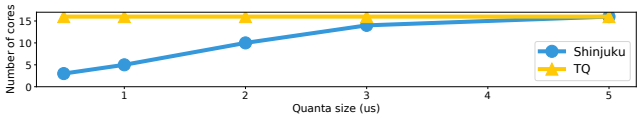

Figure 16. Cores supported with varying quantum sizes.

much higher throughput (14 Mrps) than centralized scheduling systems [32, 34, 52], where a dispatcher core can sustain only around 5 Mrps. However, such a dispatcher throughput could still be insufficient for short requests and many cores. To mitigate this issue, one could increase the number of dispatcher cores: achieving good load balancing with multiple dispatchers has been extensively studied in theoretical work [63, 69], and future systems can build on these studies.

Synthetic cache evaluation: Evaluating with synthetic workloads does not directly show the cache behavior of real applications. As the first systematic study on cache behavior under μs-scale preemptions, TQ makes this compromise to obtain interpretable results: real workloads exhibit complex memory access patterns, making it challenging to dissect the results. Prior works that studied cache behavior under kernel context switches adopt synthetic workloads for similar reasons [37, 40]. We thus call for large-scale evaluations with an extensive suite of real μs-level workloads.

MSQ: MSQ tie-breaking is motivated by empirical observations in bimodal workloads, and its performance with other service time distributions requires more theoretical studies.

Reentrancy: The current job of a worker core may call a function that a concurrent job yielded at (*i.e.,* reentrancy), which can be problematic as these jobs reside in the same process. To handle this, one needs to prevent unsafe reentrancy by not instrumenting yields in non-reentrant functions. TQ currently does not provide support for this.

7 Related work

μs-scale scheduling: To schedule workloads with broad service time distributions, prior work either assumes additional knowledge of the workload [21], or blindly preempts at a coarser time grain due to the large interrupt overhead [34]. We focus on blind scheduling and show that TQ outperforms prior systems by efficiently scheduling in tiny quanta, and TQ's forced multitasking and two-level scheduling mechanisms could provide benefits in non-blind settings as well. Concord [32] is a concurrent work that also focuses on μs-scale scheduling. While both use coroutines, TQ and Concord differ substantially in their designs. Concord adopts the centralized scheduling framework and replaces interrupts with a shared cache line that the dispatcher periodically sets and workers frequently read. As a result of centralized scheduling, Concord suffers from lower scalability: its dispatcher's load increases with the preemption frequency and core numbers and it saturates at around 4 Mrps. In contrast, TQ relies on forced multitasking so that preemptions take place without needing any external signal. This enables two-level scheduling and allows TQ's dispatcher to have a constant load with smaller quanta and sustain up to 14 Mrps.

Compiler-instrumented yields: Analyzing and instrumenting code either statically or dynamically for purposes other than inserting yield points, such as collecting program statistics and finding event frequency, has been extensively studied in prior literature [4, 5, 18, 42, 65]. For prior works that instrument code for preemptions like TQ, most of them, as discussed earlier, use an instruction-counter based approach [2, 8, 10], which suffers from large probing overhead and poor timing accuracy (§5.6). This large probing overhead results in significant performance degradation when scheduling μs-level workloads (§5.4). Ghosh et al. [28] use compiler-instrumented hooks to implement timer interrupts for Nautilus [30], a research kernel with a parallel runtime that has full access to the entire machine. It performs instrumentation based on estimated cycle counts of instructions and is conceptually most similar to TQ's approach. In comparison, TQ achieves sparser instrumentation by bounding only the maximum distances between probes and leverages optimizations like self-loop cloning and reusing the induction variable to further reduce probing overhead. None of these prior work exploits low-overhead forced multitasking to achieve efficient scheduling of μs-level workloads.

Hardware-assisted user-level interrupt: User Interrupts (UINTR) [43] is a hardware technology that enables delivering interrupts directly to user space. It achieves low interrupt overhead without requiring non-standard use of hardware virtualization features like Shinjuku. However, the overhead of UINTR is around 2000 cycles, which is still notable for small quanta. LibPreemptible [38], a preemptive user-level threading library based on UINTR, thus only supports quantum sizes larger than 3μs. In contrast, TQ achieves good performance with quantum sizes down to 1μs thanks to the ns-scale preemption overhead of forced multitasking.

Two-level scheduling: Similar concepts to two-level scheduling were proposed in other contexts such as rack-scale scheduling [70]. In TQ, we identify the opportunity enabled by forced multitasking, where there no longer needs to be an external core triggering interrupts, to efficiently schedule μs-level workloads in a single machine. Moreover, TQ leverages MSQ tie breaking to reduce the latency of long jobs.

8 Conclusion

This paper addressed a longstanding problem, that of how to blindly schedule μs-level workloads with broad service time distributions such that the tail latency is low and the system capacity is high. TQ achieves this with forced multitasking (that enables cheap preemptions at small quanta) and two-level scheduling (that handles fine-grained scheduling in a scalable and cache friendly manner). We show that this novel combination allows efficient scheduling with tiny quanta.

References

[1] Haitham Akkary and Michael A Driscoll. A dynamic multithreading processor. In *Proceedings. 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pages 226–236. IEEE, 1998.

[2] Matthew Arnold and Barbara G Ryder. A framework for reducing the cost of instrumented code. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 168–179, 2001.

[3] Remzi H Arpaci-Dusseau and Andrea C Arpaci-Dusseau. *Operating systems: Three easy pieces*. Arpaci-Dusseau Books, LLC, 2018.

[4] Thomas Ball and James R Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(4):1319–1360, 1994.

[5] Thomas Ball and James R Larus. Efficient path profiling. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29*, pages 46–57. IEEE, 1996.

[6] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the killer microseconds. *Communications of the ACM*, 60(4):48–54, 2017.

[7] Luiz André Barroso, Jeffrey Dean, and Urs Holzle. Web search for a planet: The google cluster architecture. *IEEE micro*, 23(2):22–28, 2003.

[8] Nilanjana Basu, Claudio Montanari, and Jakob Eriksson. Frequent background polling on a shared thread, using light-weight compiler interrupts. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 1249–1263, 2021.

[9] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe user-level access to privileged cpu features. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 335–348, 2012.

[10] Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. Coredet: A compiler and runtime system for deterministic multithreaded execution. In *Proceedings of the fifteenth International Conference on Architectural support for programming languages and operating systems*, pages 53–64, 2010.

[11] Kristof Beyls and Erik D'Hollander. Reuse distance as a metric for cache behavior. In *Proceedings of the IASTED Conference on Parallel and Distributed Computing and systems*, volume 14, pages 350–360. Citeseer, 2001.

[12] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81, 2008.

[13] boegel. Mica: a pin tool for collecting microarchitecture-independent workload characteristics. https://github.com/boegel/MICA, 2023.

[14] Boost. Performance of boost context switch. https://www.boost.org/doc/libs/1_79_0/libs/context/doc/html/context/performance.html, 2022.

[15] Boost. Performance of boost coroutine2. https://www.boost.org/doc/libs/1_81_0/libs/coroutine2/doc/html/coroutine2/performance.html, 2022.

[16] Sem Borst, Rudesindo Núñez-Queija, and Bert Zwart. Sojourn time asymptotics in processor-sharing queues. *Queueing Systems*, 53:31–51, 2006.

[17] Sol Boucher, Anuj Kalia, David G Andersen, and Michael Kaminsky. Putting the" micro" back in microservice. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 645–650, 2018.

[18] Bryan Cantrill, Michael W Shapiro, and Adam H Leventhal. Dynamic instrumentation of production systems. In *USENIX Annual Technical Conference, General Track*, pages 15–28, 2004.

[19] Ana Lúcia De Moura, Noemi Rodriguez, and Roberto Ierusalimschy. Coroutines in lua. *Journal of Universal Computer Science*, 10(7):910–925, 2004.

[20] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 205–220, 2007.

[21] Henri Maxime Demoulin, Joshua Fried, Isaac Pedisich, Marios Kogias, Boon Thau Loo, Linh Thi Xuan Phan, and Irene Zhang. 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*, pages 621–637, 2021.

[22] Chen Ding and Yutao Zhong. Predicting whole-program locality through reuse distance analysis. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 245–257, 2003.

[23] Stephen Dolan, Servesh Muralidharan, and David Gregg. Compiler support for lightweight context switching. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):1–25, 2013.

[24] DPDK. Data plane development kit. https://www.dpdk.org/, 2022.

[25] Kenneth J Duda and David R Cheriton. Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. In *Proceedings of the seventeenth ACM symposium on Operating systems principles*, pages 261–276, 1999.

[26] Agner Fog. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs. *Technical University of Denmark*. Copyright © 1996 – 2022. Last updated 2022-11-04.

[27] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 281–297, 2020.

[28] Souradip Ghosh, Michael Cuevas, Simone Campanoni, and Peter Dinda. Compiler-based timing for extremely fine-grain preemptive parallelism. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE, 2020.

[29] Varun Gupta, Mor Harchol Balter, Karl Sigman, and Ward Whitt. Analysis of join-the-shortest-queue routing for web server farms. *Performance Evaluation*, 64(9-12):1062–1081, 2007.

[30] Kyle C. Hale and Peter A Dinda. Enabling hybrid parallel runtimes through kernel and virtualization support. In *VEE 2016 - Proceedings of the 12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 161–175, March 2016.

[31] Mor Harchol-Balter. *Performance modeling and design of computer systems: queueing theory in action*. Cambridge University Press, 2013.

[32] Rishabh Iyer, Musa Unal, Marios Kogias, and George Candea. Achieving microsecond-scale tail latency efficiently with approximate optimal scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 466–481, 2023.

[33] Richard C Johnson, David Pearson, and Keshav Pingali. Finding regions fast: Single entry single exit and control regions in linear time. Technical report, Cornell University, 1993.

[34] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for μsecond-scale tail latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 345–360, 2019.

[35] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[36] Chang-Gun Lee, Hoosun Hahn, Yang-Min Seo, Sang Lyul Min, Rhan Ha, Seongsoo Hong, Chang Yun Park, Minsuk Lee, and Chong Sang Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE transactions on computers*, 47(6):700–713, 1998.

[37] Chuanpeng Li, Chen Ding, and Kai Shen. Quantifying the cost of context switch. In *Proceedings of the 2007 workshop on Experimental computer science*, pages 2–es, 2007.

[38] Yueying Li, Nikita Lazarev, David Koufaty, Yijun Yin, Andy Anderson, Zhiru Zhang, Edward Suh, Kostis Kaffes, and Christina Delimitrou. Towards fast, adaptive, and hardware-assisted user-space scheduling. *arXiv preprint arXiv:2308.02896*, 2023.

[39] Hwa-Chun Lin and Cauligi S Raghavendra. An approximate analysis of the join the shortest queue (JSQ) policy. *IEEE Transactions on Parallel and Distributed Systems*, 7(3):301–307, 1996.

[40] Fang Liu and Yan Solihin. Understanding the behavior and implications of context switch misses. *ACM Transactions on Architecture and Code Optimization (TACO)*, 7(4):1–28, 2010.

[41] LLVM. LLVM's analysis and transform passes. https://llvm.org/docs/Passes.html, 2022.

[42] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices*, 40(6):190–200, 2005.

[43] Sohil Mehta. x86 user interrupts support. https://lwn.net/Articles/869140/, 2021.

[44] Meta. Rocksdb. https://rocksdb.org/, 2022.

[45] Michael Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1094–1104, 2001.

[46] Ana Lúcia De Moura and Roberto Ierusalimschy. Revisiting coroutines. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 31(2):1–31, 2009.

[47] Jorge Munoz-Gama, Josep Carmona, and Wil MP Van Der Aalst. Single-entry single-exit decomposed conformance checking. *Information Systems*, 46:102–122, 2014.

[48] Hemendra Singh Negi, Tulika Mitra, and Abhik Roychoudhury. Accurate estimation of cache-related preemption delay. In *Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 201–206, 2003.

[49] Gor Nishanov. C++ extensions for coroutines. 2018.

[50] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. Scaling memcache at facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, 2013.

[51] Misja Nuyens and Adam Wierman. The foreground–background queue: a survey. *Performance evaluation*, 65(3-4):286–307, 2008.

[52] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high {CPU} efficiency for latency-sensitive datacenter workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 361–378, 2019.

[53] Chandandeep Singh Pabla. Completely fair scheduler. *Linux Journal*, 2009(184):4, 2009.

[54] George Prekas, Marios Kogias, and Edouard Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 325–341, 2017.

[55] Idris A Rai, Guillaume Urvoy-Keller, and Ernst W Biersack. Analysis of las scheduling for job size distributions with high variance. In *Proceedings of the 2003 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 218–228, 2003.

[56] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 13–24. Ieee, 2007.

[57] Boris Schäling. *The boost C++ libraries*. Boris Schäling, 2011.

[58] Jori Selen, Ivo Adan, and Stella Kapodistria. Approximate performance analysis of generalized join the shortest queue routing. *EAI Endorsed Transactions on Future Internet*, 3(10), 1 2016.

[59] Hamed Seyedroudbari, Srikar Vanavasam, and Alexandros Daglis. Turbo: SmartNIC-enabled dynamic load balancing of μs-scale RPCs.

[60] John Paul Shen and Mikko H Lipasti. *Modern processor design: fundamentals of superscalar processors*. Waveland Press, 2013.

[61] TPCC. Tpc-c. https://www.tpc.org/tpcc/, 2022.

[62] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 18–32, 2013.

[63] Shay Vargaftik, Isaac Keslassy, and Ariel Orda. LSQ: Load balancing in large-scale heterogeneous systems with multiple dispatchers. *IEEE/ACM Transactions on Networking*, 28(3):1186–1198, 2020.

[64] Adam Wierman and Bert Zwart. Is tail-optimal scheduling possible? *Operations research*, 60(5):1249–1257, 2012.

[65] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenstrom. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):1–53, 2008.

[66] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs: Characterization and methodological considerations. *ACM SIGARCH computer architecture news*, 23(2):24–36, 1995.

[67] Sergey F Yashkov. Processor-sharing queues: Some progress in analysis. *Queueing systems*, 2:1–17, 1987.

[68] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, Pedro Henrique Pennar, Max Demoulin, Piali Choudhuryr, and Anirudh Badam. The demikernel datapath os architecture for microsecond-scale datacenter systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 195–211, 2021.

[69] Xingyu Zhou, Ness Shroff, and Adam Wierman. Asymptotically optimal load balancing in large-scale heterogeneous systems with multiple dispatchers. *ACM SIGMETRICS Performance Evaluation Review*, 48(3):57–58, 2021.

[70] Hang Zhu, Kostis Kaffes, Zixu Chen, Zhenming Liu, Christos Kozyrakis, Ion Stoica, and Xin Jin. Racksched: A microsecond-scale scheduler for rack-scale computers. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1225–1240, 2020.