# SKQ: Event Scheduling for Optimizing Tail Latency in a Traditional OS Kernel

Siyao Zhao
*RCS Lab, University of Waterloo*

Haoyu Gu
*RCS Lab, University of Waterloo*

Ali José Mashtizadeh
*RCS Lab, University of Waterloo*

## Abstract

This paper presents Schedulable Kqueue (SKQ), a new design to FreeBSD Kqueue that improves application tail latency and low-latency throughput. SKQ introduces a new scalable architecture and event scheduling. We provide multiple scheduling policies that improve cache locality and reduce workload imbalance. SKQ also enables applications to prioritize processing latency-sensitive requests over regular requests.

In the RocksDB benchmark, SKQ reduces tail latency by up to 1022× and extends the low-latency throughput by 27.4×. SKQ also closes the gap between traditional OS kernel networking and a state-of-the-art kernel-bypass networking system by 83.7% for an imbalanced workload.

## 1 Introduction

Applications and hardware have evolved tremendously. Modern server applications span hundreds to thousands of nodes across the data center to serve user requests. The depth and breadth of the service tree cause users to experience request latency dominated by the tail latency of any node in an application [11]. Advancements in storage and networking are making low latency services possible [2]. Recent research systems propose using kernel-bypass and custom dataplanes to reduce latency [3, 19, 23, 37].

However, most applications in data centers are still built directly on top of traditional OSes and employ an event-driven approach, which uses an event loop that polls the kernel's event subsystem. Even popular user-level threading systems internally depend on kernel event subsystems to dispatch events efficiently across user-level threads, e.g., Go language [16], Arachne [38] and Fred [27].

Event subsystems in traditional OSes such as *Kqueue* [28] in FreeBSD and Mac OS X, *epoll* [30] in Linux, *IO Completion Ports* (IOCP) [9] in Windows and *Event Completion Framework* [4] in Solaris were developed nearly 20 years ago, predating many innovations in modern hardware. These event subsystems undermine features such as receive side scaling (RSS) [8] that can improve scalability and latency. Userspace event libraries, e.g., libevent [33], are influenced by event subsystems and propagate these issues. Research systems such as Megapipe [17] and Affinity-Accept [35] improve event subsystems but only solve part of the problem.

The evolution of modern server applications and hardware inspired us to revisit event facilities in traditional OS kernels to address the latency problem. Improving tail latency in a traditional OS is challenging. Unlike kernel-bypass and custom dataplanes, existing kernels are multipurpose and complex. Kernel event facilities tightly integrate with many subsystems including storage, network, and process management. Furthermore, we must carefully tradeoff our system's overhead with the performance benefit to the rest of the system.

This paper presents Schedulable Kqueue (SKQ), a novel event notification facility based on FreeBSD Kqueue that provides a more flexible abstraction, and allows applications to use features of modern hardware. SKQ improves tail latency and extends low-latency throughput. SKQ introduces a new architecture with improved scalability, fine-grained event scheduling and event delivery control. Application threads share a single, scalable SKQ instance, which automatically schedules events across all threads. While improvements to kernel event notification facilities have been proposed [17, 35], SKQ differentiates itself with the following contributions:

- SKQ offers multiple application-controlled scheduling policies to improve cache locality and workload imbalance. We present guidelines for policy selection in § 4.3.
- SKQ presents explicit control over event delivery including event pinning and prioritization.
- SKQ has been extensively tested and deployed on production servers. We also developed event libraries to facilitate integration in existing applications.

We evaluate SKQ with microbenchmarks and multiple real world workloads with different characteristics. We examine workloads with uniform and non-uniform request service times, and IO-bound workloads. In microbenchmarks, we

show that SKQ reduces lock contention, improves cache locality and multicore scalability. In our RocksDB [14] benchmark running the Facebook ZippyDB workload, SKQ extends the low-latency throughput by 27.4× and reduces latency by up to 1022×. Event prioritization allows a saturated server to service low-latency requests to a high priority client with 8× lower latency while having little performance impact on other clients. Additionally, SKQ closes the gap between traditional OS kernel networking and a state-of-the-art kernel-bypass networking system by 83.7% for an imbalanced workload.

## 2 Background and Related Work

SKQ was motivated by the design and performance problems with existing event subsystems. While our observations hold on popular platforms, i.e., Linux epoll and Windows IOCP, we will explain everything in the context of FreeBSD Kqueue.

**Sources of Latency:** Li et al. [29] studied several server applications and identified multiple factors in the OS that contribute to high tail latency. The authors suggested reducing background tasks, pinning threads to cores and avoiding NUMA effects. Even after following all the recommendations, we found two additional factors that lead to high tail latency.

One factor is cache misses due to receive side scaling (RSS) [8] found in modern NICs. RSS creates a send and a receive NIC queue per core and distributes connections across them using hash functions. Recent implementations of RSS also load balance by migrating groups of connections between cores. However, applications are unable to detect this and will process the connection on the original core, losing connection affinity and causing cache misses. In our Memcached benchmark, we found that up to 77% of total L2 cache misses are due to improper connection affinity and are avoidable. SKQ maintains connection affinity and follows connection migration with the *CPU affinity* scheduling policy.

Another factor is workload imbalance that arises from differences in request service time and suboptimal connection distribution across worker threads. Workload imbalance oversaturates some worker threads while under-saturating others. To put this into perspective, we measured the difference in total processing time between the most and least busy threads in two workloads. In Memcached, a uniform workload, we measured a 2.8% difference. In a GIS application with a Zipf-like service time distribution, we measured a 46% difference. As a result, the GIS application's tail latency increases much faster than Memcached as a function of throughput. SKQ offers scheduling policies to minimize workload imbalance.

**Event-Driven Models:** Most modern event-driven applications use either the *1:1 model* or the *1:N model*.

Applications using the 1:1 model create one Kqueue per thread where connections are assigned to thread private Kqueues. This model scales well and maintains connection affinity. However, the 1:1 model suffers from two problems. First, it hinders efficient event migration. Moving events between Kqueues requires two system calls that remove events from the source Kqueue and add them to the target Kqueue. This migration process also involves multiple kernel and userspace locks, leading to poor scalability. Second, this model interacts poorly with RSS in modern NICs as applications cannot detect nor react to RSS connection migration.

The 1:N model means all threads share a single Kqueue and process connections in a round-robin fashion. This model maximizes CPU utilization and simplifies event scheduling, but suffers from two problems. First, the model leads to lock contention on multicore machines (see § 6). Second, the model does not preserve connection affinity as each connection is processed by different threads, which results in cache misses. As a result, applications and event libraries have mostly avoided the 1:N model except for a few low-throughput services [25].

SKQ uses a hybrid architecture that offers the best of both worlds through our event scheduler. SKQ exposes a 1:N model to applications for efficient event scheduling and delivery while internally using a 1:1 model for multicore scalability.

**Need for Application Control:** Applications often need to deliver events to a specific thread. For example, one thread needs to notify another thread of control events. Kqueue does not provide applications using the 1:N model with fine-grained event delivery control. SKQ allows applications to pin events to specific threads.

SKQ also introduces application controlled event prioritization, which allows applications to prioritize latency-sensitive events over batch-processing events. To our knowledge, no existing kernel event subsystem supports event prioritization. The latest generation NICs provide hardware support for event prioritization with application device queues (ADQs) [20]. SKQ can be used with ADQs to improve event prioritization throughout the networking stack.

**RFS, RPS and Intel Flow Director:** Receive Flow Steering (RFS) [12] and Receive Packet Steering (RPS) [6] are Linux networking stack features. RFS improves connection locality by enabling the kernel to process packets on the core where the corresponding application thread is running. RPS is a software implementation of hardware RSS, which provides hash-based packet distribution across cores. Intel Flow Director [18] is a hardware implementation of RFS. SKQ eliminates the need for RFS with the *CPU affinity* policy.

**io_uring:** *io_uring* [7] is a recent Linux kernel feature that enables efficient asynchronous IO and avoids system call overhead via polling. Each io_uring object uses a single submission queue and a single completion queue for kernel-user communications. However, io_uring does not provide event scheduling between threads, which is the goal of SKQ.

FreeBSD offers the `lio_listio` [42] interface for asynchronous IO, which batches rather than eliminates system calls. We benchmarked the benefit of `lio_listio` combined with SKQ to improve low-latency throughput by roughly 40%,

compared to 19.8% with SKQ alone on our web server benchmark (§ 6.7). Our techniques are also applicable to io_uring.

**Windows IOCP:** Windows IOCP is functionally similar to Kqueue and provides a developer friendly API for blocking IOs when used with the 1:N model [10, 39]. Each IOCP internally uses one event queue and provides a concurrency parameter to limit the number of running threads. Threads beyond this limit block until a running thread sleeps on an unrelated IO operation (e.g. disk IO) [9]. IOCP also suffers from lock contention at high core counts because of the 1:N model. As a result, only applications with long running requests, such as Exchange and MSSQL [25], use the 1:N model.

**MegaPipe:** MegaPipe [17] permanently affinitizes accepted connections to threads to improve cache affinity. MegaPipe internally uses the 1:1 model. Prior to MegaPipe, Affinity-Accept [35] and Chronos [26] also suggest maintaining connection affinity. MegaPipe delivers socket data along with triggered connections to reduce system call overhead.

SKQ's *queue affinity* scheduling policy is similar to MegaPipe's behavior. However, queue affinity is inferior to CPU affinity that takes into account connection migration in kernel and NICs. MegaPipe was developed much earlier when connection migration was less frequent due to the maturity of RSS implementations.

**Custom Networking Stack:** Arrakis [36], Chronos [26], DPDK [19], F-Stack [41], IX [3], and mTCP [22] use kernel-bypass to reduce system calls and/or kernel networking stack processing overhead. Other systems such as Snap [31], Shinjuku [23], and ZygOS [37] apply thread scheduling on custom dataplanes. Shenango [34] implements user-level scheduling on top of DPDK and uses work stealing for load balancing.

SKQ differs from the systems above in three ways. First, SKQ schedules events between threads rather than threads between cores. Second, SKQ offers control over event delivery and scheduling policies in addition to work stealing. Last, SKQ is implemented in a traditional OS where most server applications are still being developed. SKQ enables applications to gain performance benefit with minimal changes. The systems above however generally suffer from adoption issues.

# 3 Design Overview

Modern applications require increasingly better multicore scalability, cache locality and scheduling support. FreeBSD Kqueue was designed to solve the C10K [24] problem. Server applications are now running with many cores and millions, rather than thousands, of clients. The changes in applications' scale and requirements require us to revisit Kqueue's design.

The current design presents three major issues. First, using Kqueue in the 1:N model results in scalability bottlenecks due to lock contention in some multithreaded applications with as few as 4 threads. Second, we observed cache misses and load imbalances that could not be solved. Finally, Kqueue processes events in FIFO order and lacks event prioritization.

SKQ is designed with three primary goals: scalability, event scheduling, and event prioritization. SKQ is also designed to be compatible with the Kqueue API.

- **Scalability**: SKQ scales to multicore machines and a large number of events. Our design allows a single SKQ to efficiently schedule events to multiple threads with little overhead and lock contention.
- **Event Scheduling**: SKQ implements scheduling policies that improve cache locality and minimize workload imbalance. Applications can select policies based on workload characteristics to minimize tail latency.
- **Event Prioritization**: SKQ enables applications to prioritize processing high priority events versus regular events, with minimal performance impact.

One major challenge is providing low overhead scheduling as any overhead can impact application performance. To put this in perspective, for Memcached, an increase in processing times of 150 cycles would result in a 1% drop in peak throughput, while a single L3 cache miss reduces throughput by 2.7%. Scheduling introduces unavoidable overhead due to statistics collection and making scheduling decisions.

## 3.1 Kqueue

The Kqueue API consists of two system calls. The `kqueue()` system call creates a Kqueue kernel object and returns a file descriptor. The `kevent()` system call handles event registration, update and delivery.

Applications manipulate events by calling `kevent()` on a Kqueue object with the file descriptor of a kernel object and the type of event to monitor (e.g. read availability on a socket). For event registration, the target Kqueue object allocates a *knote* to track the information and the state of the registered event. For fast lookups, all knotes on a Kqueue are kept in a hash table in the Kqueue object.

Upon creation, the knote is also attached to the kernel object of interest. In FreeBSD, all kernel objects supported by Kqueue maintain a list of attached knotes. When the kernel object triggers an event, it activates all attached knotes by notifying their corresponding Kqueue objects.

Kqueue enqueues activated knotes into a single event queue protected by a giant lock. Application threads retrieve events with the `kevent()` system call and a userspace buffer used to hold the returned events. Internally, all threads acquire the giant lock and dequeue knotes from the event queue.

One correctness guarantee that a shared Kqueue provides is to avoid returning an event that is being processed by a thread to another thread. For example, a shared Kqueue must not notify more than one thread of read availability on the same socket, otherwise a race may occur. Kqueue achieves this guarantee by relying on application threads marking shared events as *dispatch events*, which instruct Kqueue to disable the event after returning it to the application. Once an application

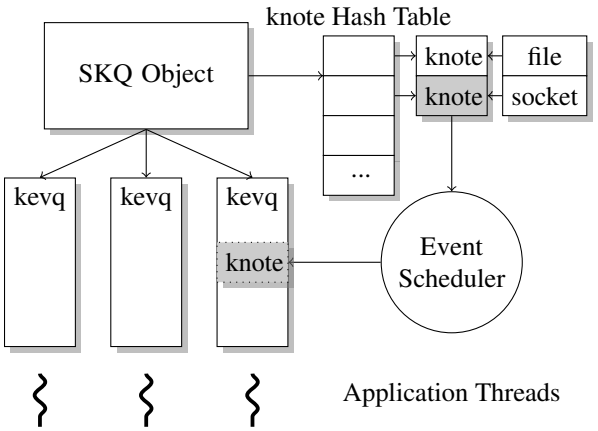


Figure 1: System Overview. SKQ creates thread private kevqs to reduce lock contention and introduces an event scheduler. In this diagram a socket activated a knote and the event scheduler enqueued it into a specific kevq based on an application defined scheduling policy.

thread finishes processing a shared event, the thread must call `kevent()` to re-enable the event so it can trigger again.

Both the single event queue design and frequent event enabling/disabling cause lock contention. Therefore, a Kqueue is rarely shared between threads in an application. Additionally, using a single event queue hinders optimizing for cache locality [17,35]. User-level scheduling is difficult as migrating knotes between Kqueues is cumbersome and inefficient.

## 3.2 SKQ

SKQ provides applications with event scheduling and delivery controls by sharing a single scalable instance between all application threads. Figure 1 shows the overall architecture. SKQ extends Kqueue in three main ways: employing a new scalable design, offering event scheduling and delivery control, and optimizing the event lifecycle. Additionally, SKQ maintains a compatibility mode that exhibits the same behavior as Kqueue.

**Scalability:** SKQ improves multicore scalability and enables efficient event scheduling by introducing a new internal, lightweight, per-thread *kevq* structure. Each SKQ creates one kevq for each application thread, which is an event queue that holds knotes assigned to the corresponding thread.

When applications query events, each worker thread locks and retrieves knotes from its private kevq, eliminating a major source of contention on the giant Kqueue lock. Furthermore, kevqs allow SKQ to quickly schedule events between lightweight internal structures instead of kernel objects. Implementing event scheduling between kernel objects requires extra locking (e.g. file descriptor locks) and complicates resource cleanup. SKQ approximates the benefit of the 1:N model without scalability bottlenecks or poor cache affinity through scheduling on top of an internal 1:1 model.

When an SKQ is destroyed, all of its kevqs are destroyed. If a thread exits while an SKQ is still active, only the exiting thread's kevq is drained and destroyed. Knotes already queued to the kevq are rescheduled to other available threads.

Applications are allowed to create multiple SKQs at a time, meaning each application thread can correspond to multiple kevqs belonging to different SKQs. We use a hash table that maintains the mapping from kevqs to their respective SKQs in the kernel thread object.

The per-thread design allows SKQ to more easily handle applications that spawn greater or fewer threads than available cores, and thread migration. While it might seem natural to use a per-core design, some applications process blocking IO calls and have more threads than cores. Applications with fewer threads than cores would require extra handling to prevent events from being left on cores with no threads.

**Event Scheduling and Delivery Control:** SKQ introduces an event scheduler to schedule knotes both passively and actively. Upon event activation, the event scheduler determines the target kevq of the activated knote based on the selected scheduling policy. When a kevq has no active knotes, the kevq also actively tries to steal knotes from other kevqs. Applications can control the scheduling policy via `ioctl()` and new per-event flags for pinning and priorities.

SKQ also allows applications to mark individual events as high priority. SKQ favors returning high priority events first. Two tunables are provided to control the exact behavior. Applications can adjust the tunables based on their requirements. We elaborate on both features in § 4.

**Optimized Event Lifecycle:** We optimize SKQ's event handling to eliminate the need of re-enabling events after processing and to improve scheduling fairness.

SKQ adds a processing flag to knotes and a processing queue to kevq. Each processing queue holds knotes returned to the application from the last `kevent()` call. SKQ marks these knotes as processing, which can be scheduled to a different kevq although they are temporarily ignored by event queries. When the original thread finishes processing events, it calls `kevent()` again, which releases all knotes on its processing queue so they can be returned by future event queries.

Applications do not need to re-enable events because SKQ marks them as processing in the kernel and will not deliver them to other threads until the original thread releases them. This optimization improves scheduling fairness as processing events are always scheduled on activation rather than after the original thread returns, preserving the relative order of arrival.

# 4 Event Scheduling Policies

The event scheduler selects the kevq for an activated knote based on the scheduling policy. SKQ offers two categories of scheduling policies that improve cache locality and reduce

workload imbalance, which can be combined. Additionally, applications can pin events to threads and prioritize events.

We originally planned to provide a single policy that would perform well for all workloads. However, we realized from our experiments that the best scheduling policy is application and workload dependent (§ 4.3 describes policy selection). To this end, SKQ provides applications with control over scheduling policy to better meet the developer and user needs.

The biggest challenge is balancing the overhead and the optimality of making scheduling decisions. Since the event scheduler runs on every event activation, we must minimize the overhead so that the cost of scheduling does not overshadow its benefit. Towards this goal, we carefully designed SKQ while considering the impact of lock contention, CPU overhead and cache footprint.

## 4.1 Cache Locality Policies

Cache misses are detrimental to applications. For example, a read request in Memcached takes about 15k cycles from the NIC receiving the request to sending the response on our machine, while a L3 cache miss takes 400 cycles. This means that each cache miss increases processing time by 2.67% and correspondingly reduces throughput. SKQ provides two policies to help applications improve cache locality.

Both policies use our *kqdom* structure, which is a multi-level N-ary tree that mirrors the system's cache and memory topology, and keeps track of the core affinity of all kevqs. Each kqdom leaf node corresponds to a core and contains a list of local kevqs. Each kqdom level represents a shared cache level. During initialization, each SKQ creates a private kqdom. When an SKQ object registers a thread, the kevq is inserted into the corresponding kqdom leaf node. When the CPU scheduler reschedules a thread to a different core, the kevq is also moved to the new kqdom leaf node.

On a multi-socket machine, a kqdom typically contains 3 levels. The top level nodes represent different NUMA domains. The second level contains all cores that share the last-level cache within a NUMA domain. The third level consists of leaf nodes corresponding to a core containing hyperthreads.

**Queue Affinity:** The *queue affinity* policy always delivers events to the core where the event first triggered, which reduces userspace cache misses by maintaining affinity.

When a knote is activated for the first time, SKQ stores the corresponding kqdom leaf node in the knote. On subsequent activations, the event scheduler queues the knote to a kevq in the same kqdom leaf node. When multiple kevqs are present in the same kqdom leaf node, a random kevq is selected. However, when connections migrate, this policy results in more cache misses in the kernel.

**CPU Affinity:** The *CPU affinity* policy delivers events to the application threads local to the triggering core. For network sockets, the policy always queues knotes to the core that received the NIC interrupt and processed the packet.

This policy improves cache locality within the kernel. Before an event activation, the networking stack has already processed the incoming packet, which pulls socket buffers and metadata into the local core's cache. CPU affinity policy keeps these cache lines local so subsequent reads and writes in userspace will more likely result in cache hits.

Furthermore, CPU affinity cooperates well with all sources of connection migration including RSS. After a connection migrates to a different core, the policy follows the migration and queues the event to a kevq on the new core. This reduces the cache misses caused by the application and kernel processing the event on different cores. This also reduces kevq lock contention during event activation as cores will deliver events to their local kevq from the packet processing, rather than competing to deliver events to the same kevq.

## 4.2 Workload Balancing Policies

Workload imbalance is another major cause of suboptimal performance. An imbalanced load distribution leads to resource under-utilization where some worker threads are idling whereas others are overloaded. We implement two policies that mitigate imbalances.

**Best of Two:** The *best of two* policy load balances events between threads by randomly selecting two kevqs and choosing the kevq with a lower expected wait time. Mitzenmacher [32] shows that this policy offers good optimality. Expected wait time $E[t]$ refers to the wait time before the activated knote is serviced by an application thread.

For each kevq, we maintain the number of knotes currently queued $n_{cur}$, the number of knotes returned to userspace of the last `kevent()` call $n_{ret}$, the timestamp of the last `kevent()` call $t_{ret}$, and a moving average of the processing time in cycles per knote $t_{avg}$. We use an exponential moving average with $\alpha = 0.05$, hence $t'_{avg} = 0.05 * t_{new} + 0.95 * t_{avg}$, which we found to react fast enough while smoothing out workload behavior and noise from interrupts and scheduling. When applications finish processing a knote and return it to SKQ via `kevent()`, the moving average is updated.

On knote activation, the event scheduler calculates the expected wait time with $E[t] = t_{ret} + t_{avg} * (n_{ret} + n_{cur})$ for both selected kevqs. Finally, the event scheduler enqueues the knote to the kevq with earlier $E[t]$.

A potential issue is that a thread may unexpectedly stall in userspace. In this case, $E[t]$ can lag far behind and be in the past. To prevent the event scheduler from assigning too many knotes to the thread, we set $t_{ret}$ to $MAX(t_{ret}, t_{cur} - t_{avg} * n_{ret})$, where $t_{cur}$ is the current timestamp.

**Work Stealing:** Work stealing allows idle threads to steal knotes from another thread's kevq. Unlike other scheduling policies, work stealing operates during the dequeuing of knotes rather than knote activation. When a thread has no knotes to process, it normally blocks until some knotes arrive. With work stealing enabled, instead of blocking, the idle

thread picks a random victim kevq and tries to steal knotes. The system will periodically retry until either new knotes arrive or at least one knote is stolen. Applications can control the maximum stolen knotes per attempt (defaults to 1).

In order to reduce the overhead and lock contention of work stealing, we use trylocks to check the availability of the victim kevq and knotes. The idle thread first trylocks the victim kevq. If the victim kevq is busy, the idle thread sleeps for a fixed amount of time then retries. Otherwise, the idle thread scans the victim kevq for knotes. We use the same trylock technique to probe each knote, skipping knotes that are undergoing changes (e.g., being scheduled). To avoid locking the victim kevq for too long, we bound the maximum number of scanned knotes to twice the maximum stolen knotes.

Another issue is thrashing that can occur because of lock ordering issues. When a knote is stolen by an idle thread it is moved to the idle thread's kevq. For a short window of time we must drop all kevq locks to not violate lock ordering. This allows another thread to steal events from the idle thread before it can process any stolen knotes. This causes thrashing of events as they can bounce around due to this race. We added a flag to denote whether it is stolen. Knotes with the flag set are skipped during work stealing. The flag is cleared after the stolen knote is processed by the idle thread.

Our measurements show that for applicable workloads, work stealing improves the latency response and reduces tail latency (see § 6.5). We also observe negligible lock contention because of the trylock optimization using FreeBSD *lockstat*. Our earlier implementation did not employ the trylock optimization that caused work stealing to increase the overall latency in some workloads due to the aforementioned issues.

**Hybrid Policies:** SKQ allows applications to combine cache locality policies with workload balancing policies. In this case, the event scheduler picks the first kevq according to the cache locality policy, and then uses best of two to pick a second kevq from the rest. The expected wait time of both selected kevqs are then compared with a constant cache miss penalty applied to the kevq selected by best of two. While the cache miss penalty of migrating to a random core is both application and workload dependent, in our experiments we found a constant penalty was enough to prevent unnecessary migration when the imbalance was not significant.

Work stealing can be used with other policy combinations as it operates during knote dequeuing. In our experiments, hybrid policies that combine all three scheduling options comprise the best-performing polices for imbalanced workloads.

## 4.3 Policy Selection Criteria

Policy selection is based on the application's workload characteristics. Applications with uniform and low response times should use the CPU affinity policy to maximize cache affinity. Applications with imbalanced or IO-heavy workloads should use the hybrid policy consisting of CPU affinity and best of two with work stealing to balance the threads and extend low-latency throughput. The rationale is discussed in § 6.

## 4.4 Fine-grained Event Delivery Controls

Besides scheduling, SKQ allows applications to control event delivery on a per-event level. We currently offer two controls to handle event pinning and event prioritization.

**Event Pinning:** SKQ allows applications to pin individual events to specific threads. Application threads use the affinity flag for each event during event registration. The flag ensures that the event is always delivered to the registering thread.

This is useful to applications where threads communicate based on pipes or user events. For example, in Memcached, the main thread uses a pipe to communicate control messages to each worker thread. Scheduling these events between threads would break the notification system.

**Event Prioritization:** Event prioritization enables applications to prioritize latency-sensitive traffic such as end user requests over batch processing. SKQ defines two event priority levels: regular and high priority. By default, all registered events are of regular priority. Applications can promote an event to high priority by setting the high priority flag.

In each kevq, we maintain a separate event queue for high priority knotes. During event activation, the event scheduler queues activated knotes to their corresponding queue. During event queries, `kevent()` prioritizes dequeuing knotes from the high priority queue. The remaining space left in the userspace buffer is filled with regular priority events.

Our design also addresses two potential problems of event prioritization. First, too many high priority events may cause starvation. To prevent starvation, we introduced the *rtshare* parameter that controls the maximum percentage of returned high priority events. For example, an *rtshare* of 80 means at most 8 high priority events out of 10 total events will be returned to the application per `kevent()` call.

Second, if `kevent()` returns too many events per call, which is common at high throughput, the application thread might spend a long time processing all events, postponing the delivery of newly arrived high priority events. To address this issue, applications can set the *rtfreq* parameter to control the number of `kevent()` calls a thread should perform per second, bounding the latency of high priority events. SKQ dynamically limits the number of events returned to the application based on the average processing time statistics.

Applications gain the full benefit of event prioritization by tuning *rtshare* and *rtfreq* based on workload characteristics and user requirements. In our benchmark, we observed an $8\times$ tail latency improvement of a high priority client at peak throughput with little impact on regular clients.

## 5 Implementation

We implemented SKQ in FreeBSD 13 (commit 04816a1) with ∼3000 source lines of code (SLOC). We also developed two SKQ event libraries *libevent-skq* and *libskq* for easy adoption in ∼1700 SLOC. libevent-skq is compatible with libevent and solves its limitation with concurrent event processing on a single event base. libskq uses a custom API with reduced lock contention and system call batching via lio_listio (disabled in our evaluation). Additionally, we implemented a custom webserver in ∼1000 SLOC on top of libskq that conforms with HTTP/1.1 as defined in RFCs 7230–7237 [21].

## 6 Evaluation

We evaluate our system using microbenchmarks and four applications. We choose workloads based on their request service time characteristics, e.g. Memcached has a uniform request service time whereas RocksDB displays workload imbalance. We also measure the event prioritization benefit and compare SKQ with Shenango, a kernel-bypass system.

All server applications run on a server with dual 2.1 GHz Intel Skylake-SP Silver 4116 with an Intel X722 10 GbE NIC. The workload generating clients run on 6 identical Skylake machines and 4 additional machines with dual Intel Xeon E5-2680 and a Mellanox ConnectX-3 10 GbE. Turbo boost is disabled on all machines to minimize measurement error. Hyperthreading is enabled with one application thread per core and NIC interrupts scheduled to the adjacent hyperthread as recommended by NIC vendors.

In all experiments we use 12 server threads, 12 client threads and 12 connections per client thread. Each run consists of a 5-second warmup followed by a 25-second measurement. All threads are pinned to the first socket. The measurement client establishes one connection per thread to probe the server response time. Each data point is obtained as the average of three runs and all results are statistically significant.

In the following sections, *vanilla* refers to multiple SKQs (the 1:1 model) to isolate the scheduling benefit. We do this because SKQ in compatibility mode provides a modest performance improvement over Kqueue (§ 6.2).

### 6.1 Scalability

We built a benchmark that generates events via UNIX pipes to measure the scalability of Kqueue/SKQ. Each client thread establishes 12 pipe pairs and sends requests with a constant processing time of 5 µs. We run four configurations in total while varying the number of threads.

Shared SKQ scales linearly and on par with both multiple Kqueues and multiple SKQs. We see a constant throughput increase with each additional core, since all three setups have little lock contention and SKQ schedules events with low overhead. Shared Kqueue starts to contend after 4 cores. At

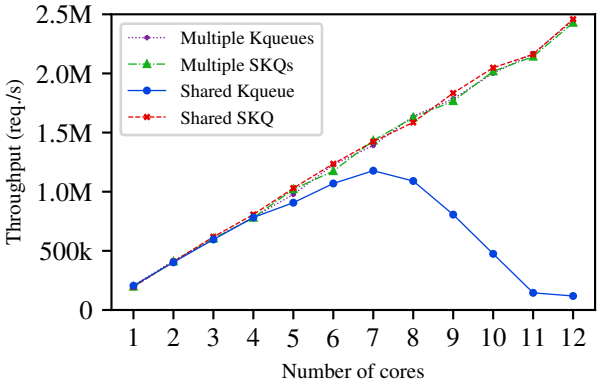


Figure 2: The impact of adding cores on various configurations' peak throughput. Shared refers to the 1:N model while Multiple refers to the 1:1 model. We use the CPU affinity policy for shared SKQ.

12 cores, shared Kqueue only achieves 4.8% of the throughput of other configurations. This reduction in throughput comes from lock contention in Kqueue. Additional experiments show that the result of shared Kqueue is workload dependent. For requests with very short processing time, the giant Kqueue lock is the main source of contention. The lock contention decreases as the processing time increases.

We use FreeBSD lockstat [15] to compare the lock contention of shared Kqueue with SKQ at maximum throughput with 12 cores. The result shows that shared Kqueue has significantly higher lock contention due to threads competing for the single event queue and the giant lock.

For shared Kqueue, the top three contending system locks belong to Kqueue, which comprise 69% of total lock contention. Shared SKQ displays much lower contention with the worst contending SKQ lock ranked the 4$^{th}$ in the system. The top three contending SKQ locks only comprise 12% of total lock contention.

By breaking up Kqueue's giant lock and introducing thread private kevqs, SKQ significantly lowers the lock contention at high core counts, achieving much better scalability.

**CPU Affinity vs. Queue Affinity:** In our benchmarks, CPU affinity performs on par or better than queue affinity. As we mentioned before, this is due to connection migration, which breaks connection affinity with queue affinity. *Thus, we consider CPU affinity superior to queue affinity in general and will omit queue affinity from further discussions.*

### 6.2 Multiple SKQs vs. Multiple Kqueues

Figure 3 shows the latency response of an unmodified Memcached using multiple Kqueues vs. multiple SKQs (the 1:1 model). We improve tail latency at throughputs between 750k–1.1M req./s. At maximum throughput, we lower tail latency by 33%. The improvement comes from the new SKQ archi-

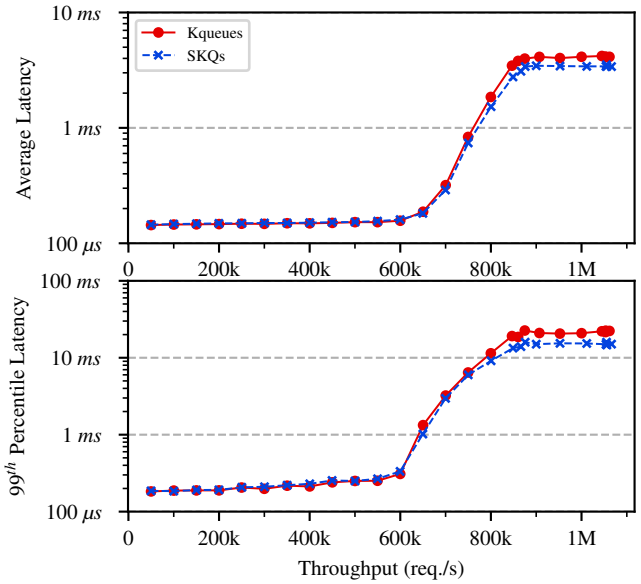


Figure 3: The latency response of Memcached using multiple SKQs vs. multiple Kqueues. SKQ lowers the tail latency by 33% at high throughput due to fine-grained locking.

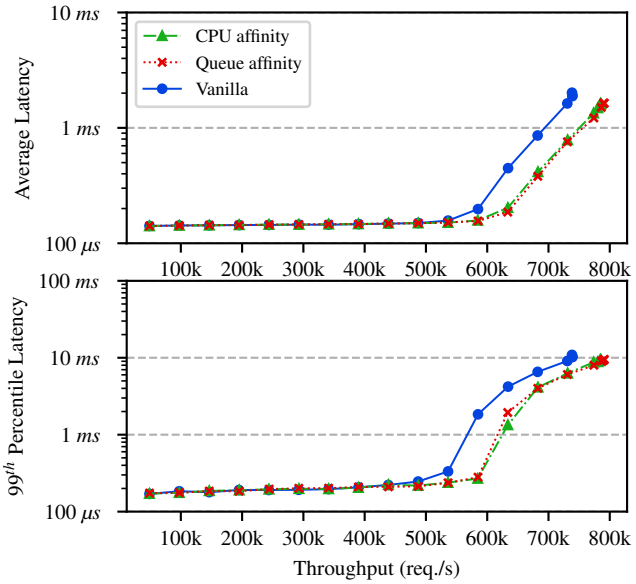


Figure 4: Cache locality policies' latency improvements of a lightweight RPC echo endpoint.

tecture and fine-grained locking.

*Throughout the evaluation, "vanilla" refers to multiple SKQs (the 1:1 model) to accurately measure the benefit of our scheduling policies.* This isolates the improvement due to architectural changes.

## 6.3 Cache Miss Analysis

We use an RPC echo service to understand how cache locality policies affect cache miss rates. Figure 4 shows the latency response for an RPC echo client while varying throughput against three different policies. Both CPU and queue affinity outperform vanilla above 500k req./s. At 580k req./s, both policies show the maximum benefit over the vanilla with a 6.8× (269 µs vs. 1839 µs) lower tail latency and 20% more low-latency throughput (585k req./s vs. 487k req./s).

Using CPU performance counters, we analyze the cache behavior to understand where the benefit comes from. Table 1 shows the L2 cache misses at 580k req./s broken down by various kernel code paths.

Both scheduling policies significantly reduce L2 cache misses in the TCP input/output path and event query path. The TCP input path includes receive-side packet processing, while the TCP output path includes processing the socket buffer into packets. The event query path is all the code executed during `kevent()` calls to retrieve events. In all paths, SKQ outperforms as both cache locality policies preserve cache affinity better, thus reducing cache misses.

The vanilla configuration of Memcached (see § 6.4) is typical of other applications that ignore cache affinity information

| Policy | TCP input | TCP output | Event activation | Event query | Total |
|---|---|---|---|---|---|
| CPU | 252k | 15k | 63k | 166k | 496k |
| Queue | 343k | 33k | 95k | 250k | 721k |
| Vanilla | 828k | 76k | 45k | 1235k | 2184k |

Table 1: L2 data cache misses collected using hardware performance counters for 20 s running at 580k req./s. TCP input/output refer to the packet transmit and receive path, event activation includes event activation and scheduling, and event query is the application retrieving events with `kevent()`.

(unavailable in userspace) and use a round-robin assignment of connections to worker threads. As a result, vanilla contains many mismatched connections, i.e. connections processed by the kernel on one core but processed by an application thread on a different core, leading to more cache misses and worse tail latency.

In the event activation path, vanilla has fewer cache misses than our scheduling policies. This is expected as both policies require extra scheduling logic and accesses to new data structures like the kqdom. However, the cache miss difference is minor in comparison to the other three code paths and has little impact on the overall performance.

CPU affinity has fewer cache misses compared to queue affinity. This is due to connection migration between cores. Recall that the queue affinity policy does not follow the migration. In this workload, the difference is minor enough to not show a significant latency difference.

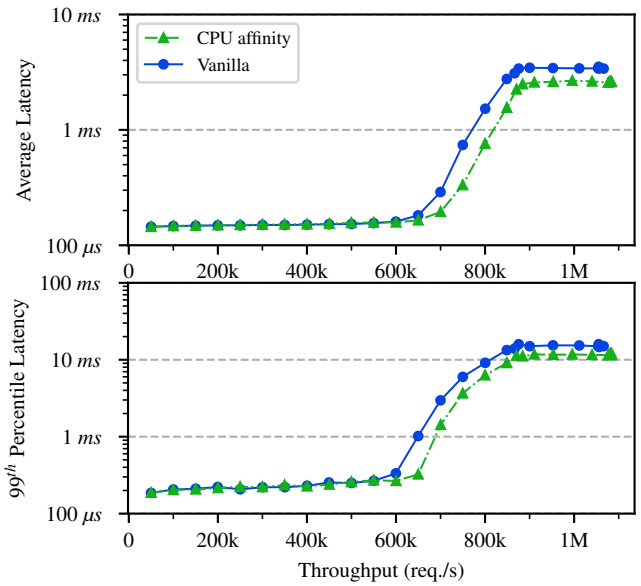


Figure 5: The latency response of Memcached using vanilla and our cache affinity policies. The CPU affinity policy increases low-latency throughput and exhibits lower tail latency in medium to high throughput range.

## 6.4 Memcached

We benchmark Memcached using Mutilate with the Facebook ETC workload [1]. Figure 5 shows the latency response of the best-performing policy. The largest latency gap occurs at 640k req./s where CPU affinity had 3× lower latency than vanilla. With a maximum tail latency of 320 µs, SKQ effectively increases 9% of low-latency throughput (650k req./s vs. 600k req./s). At high load, SKQ achieves 5% higher throughput and 26% lower tail latency.

Mutilate's request service time distribution is nearly uniform as it only issues GET and PUT requests. We measure the total time each worker thread spent processing requests in userspace at maximum throughput. The thread with the highest processing time spent 2.8% more than that of the thread with the lowest processing time, indicating that load imbalance is insignificant among worker threads.

With a uniform and balanced workload, cache locality policies offer cache affinity without the scheduling overhead of the load balancing policies. Besides preserving cache locality, there is little that can be done in the kernel event subsystem to improve the performance of Memcached without optimizing the application itself, e.g. reducing lock contention [26].

## 6.5 Application Server

Server applications often need to service client connections with different latency characteristics, e.g. connections that issue only light requests vs. heavy requests. To investigate the benefit of SKQ on such workloads, we developed a GIS application server with a Zipf-like distribution of request service times that models MyBikeRoutes-OSM traffic [40].

In this benchmark, each client issues requests that are either light, medium or heavy computation tasks (approximately 10 µs, 50 µs and 200 µs). Each client connection is assigned to perform a single task. The overall connection characteristics follow a Zipf-like distribution consisting of 95% light, 4% medium and 1% heavy connections. We collect the latency response of all the policies.

Figure 6 shows the most interesting policies. The hybrid policy (CPU affinity and best of two with work stealing) has 6% lower peak throughput compared to vanilla due to the overhead of scheduling. However, the hybrid policy provides much lower tail latency and more low-latency throughput at mid range. At 460k req./s, the hybrid policy reached the largest gap of 9.9× lower tail latency and 3.4× lower average latency. For a maximum tail latency of 280 µs, the hybrid policy extends the low-latency throughput by 3×. It is also worth noting that the hybrid policies show benefit even at low throughput. The analysis is further discussed in § 6.6.

To quantify the imbalances, we again measure the total amount of time each worker thread spent processing events in userspace at 460k req./s (the largest gap) for both vanilla and the hybrid policy. The percent difference between the busiest and the least busy thread is 46.1% for vanilla and 1.4% for the hybrid policy. This shows that the hybrid policy significantly reduced the workload imbalance between threads.

**Work Stealing:** Figure 6 shows the latency response of best of two with and without work stealing. Although best of two mitigates the imbalance to some extent, work stealing further shifts the curve outward, providing lower tail latency and more low-latency throughput. This is because while best of two balances the threads according to statistics at event activation time, work stealing enables SKQ to react faster and respond immediately to runtime changes and imbalance.

**Hybrid Policy:** Figure 6 also shows that a hybrid policy of CPU affinity and best of two with work stealing further extends the low latency throughput. This is because CPU affinity also considers cache locality. The hybrid policy always considers the thread with the better cache locality, and prefers it unless best of two shows substantial benefit.

## 6.6 RocksDB

RocksDB is a fast and persistent embeddable key-value store. Cao et al. [5] characterized and modeled several Facebook workloads that use RocksDB as the backend. We developed a frontend to RocksDB that services GET, SEEK and PUT requests. In this benchmark, we use the same client configuration as the ZippyDB (prefix_dist) model specified by Cao. We place the RocksDB directory on a memory backed file system and preload it with 50M keys.

Unlike Memcached, the ZippyDB workload presents heavy imbalances due to its 3% SEEK requests. As an experiment,

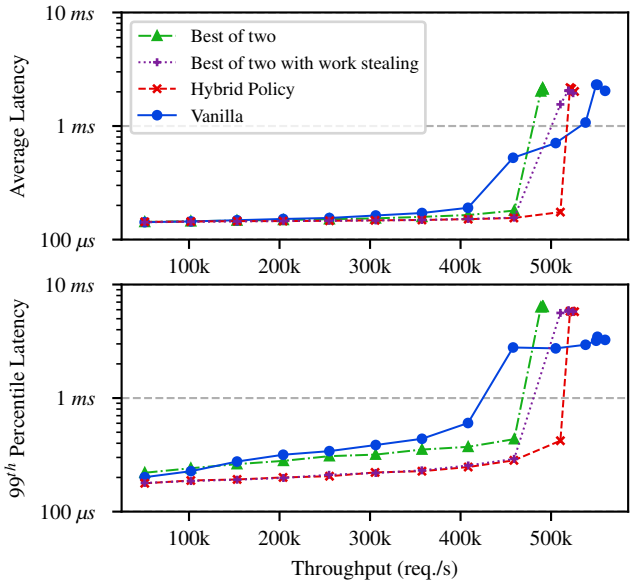


Figure 6: The latency response of the application server using various policies. The hybrid policy refers to CPU affinity and best of two with work stealing. All workload balancing policies lower the latency to different extents.

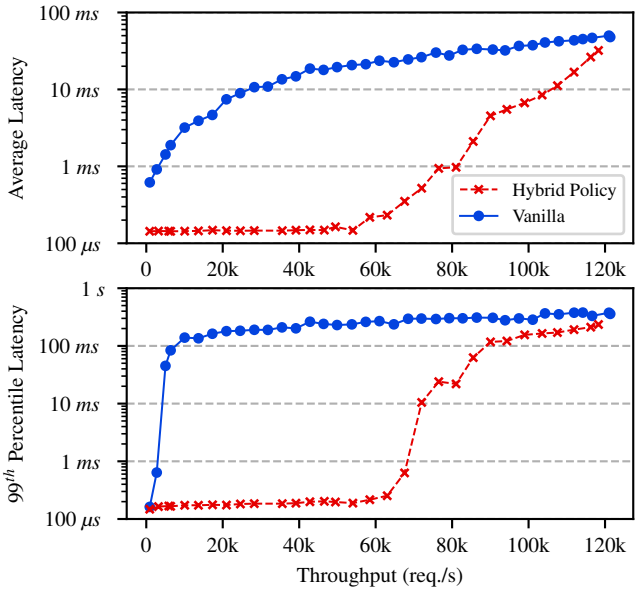


Figure 7: The latency response of RocksDB using the Facebook ZippyDB workload. The hybrid policy refers to CPU affinity and best of two with work stealing. SKQ significantly improves the low-latency throughput.

we eliminate all SEEK requests from the workload and observed 5.85× higher maximum throughput (737k req./s vs. 126k req./s). Figures 7 and 8 show the latency data over full and low throughput range. At low throughput, the tail latency of vanilla starts to increase at only 2.3k req./s whereas the hybrid policy (CPU affinity and best of two with work stealing) stays flat. This indicates extreme imbalances in the workload and is in line with what Cao reported.

Despite a lower peak throughput, the hybrid policy extends the low-latency throughput by 27.4× (63k req./s vs. 2.3k req./s). The largest latency gap occurs at 63k req./s where the hybrid policy achieves 1022× (263 µs vs. 269 µs) lower tail latency. This benchmark demonstrates SKQ's benefit to highly imbalanced workloads.

Unlike the GIS application workload where connections have a fixed request type, RocksDB connections all have the same request service time distribution. At low throughput, heavy SEEK requests are relatively rare where they can still be handled in time and do not queue up. In the GIS application, even a few misassigned heavy connections could overload a thread at low throughput, which is why the hybrid policy showed benefit earlier.

## 6.7 Web Server

So far we showed the benefit of SKQ on both balanced and imbalanced in-memory workloads. In this benchmark, we use our custom web server to demonstrate how SKQ improves IO-heavy workloads.

The web server is configured to service HTTP requests of small HTML pages (∼640 bytes) without caching residing on a HDD. Figure 9 shows the results of the best-performing policy. Similar to other imbalanced benchmarks, the hybrid policy has marginally lower peak throughput, but provides 19.8% higher low-latency throughput with 300 µs maximum tail latency. The biggest tail latency improvement occurs at 341k req./s where vanilla has 3.6× higher tail latency (1386 µs vs 384 µs). We can see that SKQ also offers benefit on heavy, uniform IO workloads.

In this benchmark, clients request web pages of similar size, which is a uniform IO workload. However, the web server benefits most from a workload balancing policy. This is because IO requests have high variance in latency. Some HTML pages might be in the file system buffer cache whereas others need to be read from the hard disk. This results in different access times and leads to an imbalanced workload. However, compared to our RocksDB and application server benchmark, IO does not induce as much imbalance. Thus, the web server benchmark shows less benefit than the other imbalanced workloads.

## 6.8 Event Prioritization

This benchmark demonstrates the effect of event prioritization on a lightweight high priority client. We use the same workload distribution as the application server.

The measurement client is assigned regular priority in the first experiment but high priority in the second experiment.

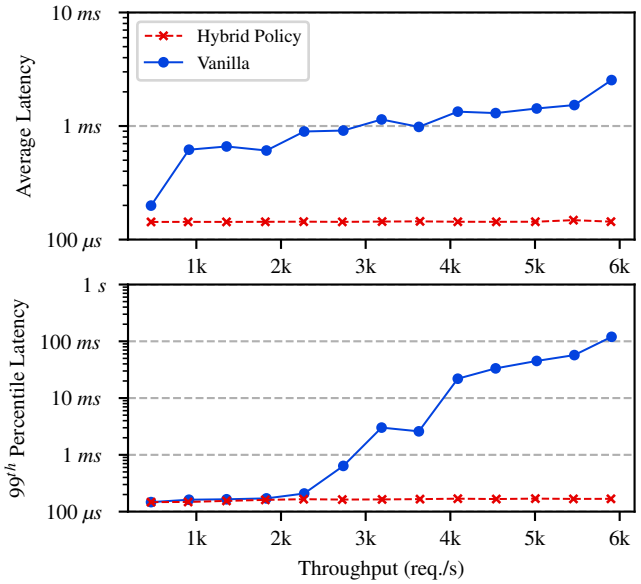


Figure 8: The low throughput latency response of Figure 7. Vanilla's latency rises at very low throughput.

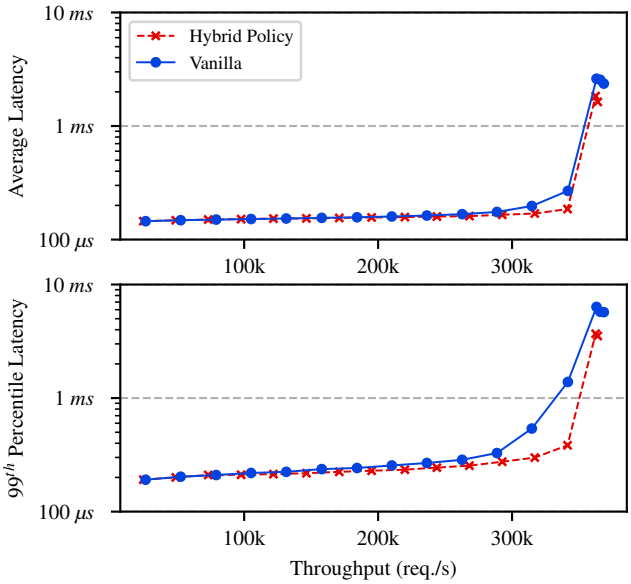


Figure 9: The latency response of our web server with file caching turned off. The hybrid policy refers to CPU affinity and best of two with work stealing. SKQ provides more low-latency throughput for IO-heavy workloads.

The measurement client also sends 30000 lightweight requests per second to probe the latency. We set rtfreq to be 10000 so that server threads call `kevent()` every 100 µs. rtshare is left at 100 to let SKQ pass back as many high priority events as possible. We measure the latency experienced by the measurement client in both settings and the regular clients' latency in the second experiment.

Figure 10 shows the latency response. At peak throughput, the high priority client's throughput comprises 5% of all traffic. The high priority measurement client's tail latency only increases by 1.2× (371 µs) than that at low throughput (263 µs) whereas the regular priority measurement client's tail latency increases by almost 10× (2656 µs). In addition, the regular clients did not experience increased latency due to the addition of the high priority client but the server did lose 0.9% maximum throughput. This benchmark shows that event prioritization enables the server to service a lightweight, high priority client with low tail latency while having minimal impact on regular clients even when the server is fully loaded.

The tunables (rtshare, rtfreq, high priority client's request rate) used are specifically tuned for this workload. Blindly changing the tunables to meet unrealistic goals only lead to decreased performance. For example, if the request rate of the high priority client is too high, it will experience increased latency as serving a large amount of high priority requests with low latency without starvation is difficult. Furthermore, using inappropriate rtshare and rtfreq may cause starvation or priority loss. Therefore, we do not offer recommended values for the tunables as they are highly situational.

## 6.9 Comparing with a Kernel-bypass System

We compare SKQ with the state-of-the-art kernel-bypass system Shenango to illustrate the benefits and tradeoffs. We also considered porting SKQ to F-Stack, a kernel-bypass library based on FreeBSD's networking stack. Unfortunately, F-Stack does not support multithreading within a single instance [13]. While outperforming kernel-bypass techniques in a traditional OS is unlikely without significant advances in hardware and software, we can still narrow the performance gap.

We built Shenango on Debian 10 running on a Skylake machine with an Intel 82599 NIC as Shenango only supports two modified NIC drivers in DPDK. We use one Skylake machine as the measuring client and six Skylake machines as workload generation clients. We benchmark three synthetic workloads with different request service time distributions: two with uniform distribution (10 µs, 20 µs) and one with imbalanced Zipf-like distribution (85% 10 µs, 12% 50 µs, 3% 200 µs). We compare how each system maximizes the low-latency throughput (<150 µs 99$^{th}$ percentile latency).

This comparison is unfair due to vastly different approaches and environments. First, Shenango implements a custom networking stack using DPDK and user-level scheduling, both of which bypass the Linux kernel. SKQ only redesigns the event subsystem in FreeBSD. Second, Shenango's TCP stack is a research prototype and lacks TCP features such as congestion control, while SKQ uses FreeBSD's networking stack.

In Figure 11, at 10 µs request service time, Shenango shows 1.67× higher low-latency throughput than SKQ. However,

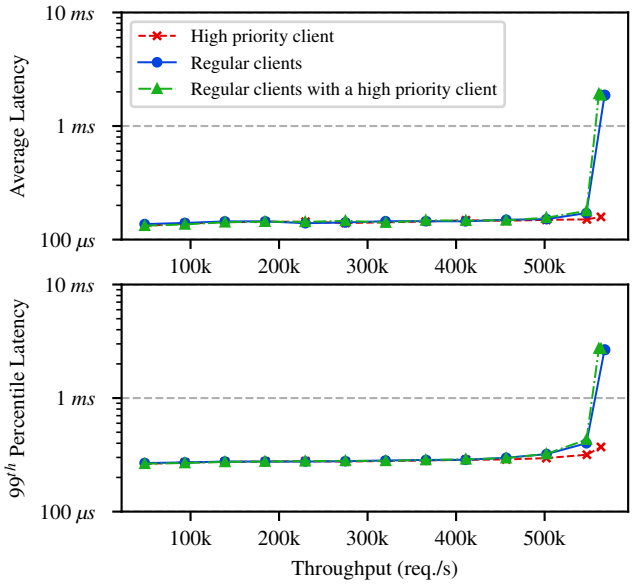


Figure 10: The latency response of two runs where the measurement client is marked as high priority vs. regular priority. The server services a high priority client with low latency while having little impact on other clients at peak throughput.

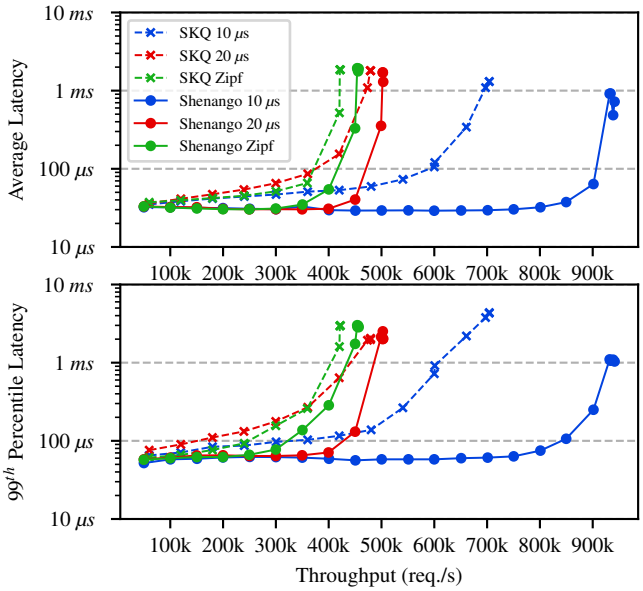


Figure 11: The latency response of three workloads running on Shenango and SKQ. SKQ uses CPU affinity scheduling policy for 10 µs, 20 µs benchmarks; CPU affinity and best of two with work stealing for the Zipf-like benchmark.

for 20 µs request service time, Shenango only achieves 1.5× higher low-latency throughput. This is because system call overhead comprises a larger fraction of time when the request service time is low. Heavier requests mask system call overhead as applications spend more time in userspace.

For the imbalanced workload, Shenango shows less benefit than uniform workloads – only 16.7% higher low-latency throughput than SKQ. This gap is lower because of the longer average service times and correspondingly lower OS overhead. SKQ achieves a result very close to Shenango as we measure nearly no load imbalance (§ 6.5). We also measure the latency response of vanilla, i.e., multiple SKQs, and found that SKQ closes the gap between vanilla and Shenango by 83.7%.

Kernel-bypassing shows bigger improvements when the request service time is low and uniform. For imbalanced workloads and heavier requests, the benefit of kernel-bypassing diminishes compared to kernel event scheduling.

Additionally, Shenango suffers from difficult adoption. First, Shenango requires changes to the threading model, synchronization and networking APIs. Porting complex applications such as RocksDB proved to be challenging. SKQ requires less invasive changes and affects only one system call. As a comparison, Shenango's Memcached results in ~1200 SLOC changed whereas SKQ's Memcached modifies only ~200 SLOC, including SKQ statistics collection and exposing SKQ controls to configuration files. Second, Shenango requires DPDK and patching NIC drivers, which increases maintenance effort. SKQ is built into the kernel and supports all NICs supported by the operating system.

## 7 Conclusions

SKQ revisits existing kernel event facilities and provides a practical solution to the latency problem for applications running in traditional OSes. SKQ offers novel features and a production quality implementation that has been developed over the course of nearly two years. We show significant performance gains in applications by only revisiting kernel event subsystems. This suggests that there are still optimization opportunities in traditional OS kernels.

## Availability

All of our code and scripts to run experiments are available at https://rcs.uwaterloo.ca/skq/.

## Acknowledgments

The authors would like to thank Samer Al-Kiswany, Tim Brecht, Ryan Hancock, Martin Karsten, Amilios Tsalapatis and Bernard Wong for fruitful discussions during SKQ's development. We would also like to thank the anonymous USENIX ATC reviewers for their valuable feedback, especially our shepherd Anton Burtsev. This research is supported by NSERC Discovery grant, Waterloo-Huawei Joint Innovation Lab grant, and NSERC CRD grant.

## References


[1] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-scale Key-value Store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 53–64, New York, NY, USA, 2012. ACM.

[2] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the Killer Microseconds. *Commun. ACM*, 60(4):48–54, March 2017.

[3] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 49–65, Broomfield, CO, October 2014. USENIX Association.

[4] Robert Benson. The Event Completion Framework for the Solaris Operating System. http://developers.sun.com/solaris/articles/event_completion.html, July 2004.

[5] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 209–223, 2020.

[6] Jonathan Corbet. Receive packet steering. https://lwn.net/Articles/362339/, November 2009.

[7] Jonathan Corbet. Ringing in a new asynchronous I/O API. https://lwn.net/Articles/776703/, January 2019.

[8] Microsoft Corporation. Introduction to Receive Side Scaling. https://docs.microsoft.com/en-us/windows-hardware/drivers/network/introduction-to-receive-side-scaling, 2017.

[9] Microsoft Corporation. I/O Completion Ports. https://docs.microsoft.com/en-us/windows/win32/fileio/i-o-completion-ports, May 2018.

[10] Helen Custer. *Inside Windows NT*. Microsoft Press, Redmond, Washington, First edition, 1992.

[11] Jeffrey Dean and Luiz André Barroso. The Tail at Scale. *Commun. ACM*, 56(2):74–80, February 2013.

[12] Jake Edge. Receive flow steering. https://lwn.net/Articles/382428/, April 2010.

[13] F-Stack. Can f-stack network stack run as 1 process on multiple cores with multiple threads? https://github.com/F-Stack/f-stack/issues/27, June 2017.

[14] Facebook. RocksDB. https://rocksdb.org, 2020.

[15] FreeBSD Foundation. lockstat – report kernel lock and profiling statistics. https://www.freebsd.org/cgi/man.cgi?query=lockstat&sektion=1&manpath=freebsd-release-ports, 2015.

[16] Google, Inc. The Go Programming Language. https://golang.org/, September 2019.

[17] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. MegaPipe: A New Programming Interface for Scalable Network I/O. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 135–148, Berkeley, CA, USA, 2012. USENIX Association.

[18] Intel. Introduction to Intel Ethernet Flow Director and Memcached Performance. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/intel-ethernet-flow-director.pdf, 2017.

[19] Intel. Data Plane Development Kit (DPDK). https://www.dpdk.org, 2020.

[20] Intel Corporation. Performance Testing Application Device Queues (ADQ) with Redis. https://www.intel.com/content/www/us/en/architecture-and-technology/ethernet/application-device-queues-with-redis-brief.html, 2019.

[21] Internet Engineering Task Force. IETF HTTP Working Group. https://httpwg.org/, May 2020.

[22] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 489–502, 2014.

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

[24] Dan Kegel. The C10K Problem. http://www.kegel.com/c10k.html, May 1999.

[25] Kimberly L. Tripp Conor Cunningham Adam Machanic Ben Nevarez Kalen Delaney, Paul S. Randal. *Microsoft SQL Server 2008 Internals*. Microsoft Press, Redmond, Washington, 1 edition, 2008.

[26] Rishi Kapoor, George Porter, Malveeka Tewari, Geoffrey M Voelker, and Amin Vahdat. Chronos: Predictable Low Latency for Data Center Applications. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 9. ACM, 2012.

[27] Martin Karsten and Saman Barghi. User-Level Threading: Have Your Cake and Eat It Too. *Proc. ACM Meas. Anal. Comput. Syst.*, 4(1), May 2020.

[28] Jonathan Lemon. Kqueue: A Generic and Scalable Event Notification Facility. In *USENIX Annual Technical Conference, FREENIX Track*, ATC '01, Berkeley, CA, USA, 2001. USENIX Association.

[29] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. Tales of the Tail: Hardware, OS, and Application-Level Sources of Tail Latency. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, page 1–14, New York, NY, USA, 2014. Association for Computing Machinery.

[30] Linux Man Page Project. epoll - I/O event notification facility. http://man7.org/linux/man-pages/man7/epoll.7.html, September 2019.

[31] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C Evans, Steve Gribble, et al. Snap: a Microkernel Approach to Host Networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 399–413, 2019.

[32] Michael Mitzenmacher. The Power of Two Choices in Randomized Load Balancing. *IEEE Trans. Parallel Distrib. Syst.*, 12(10):1094–1104, October 2001.

[33] Nick Mattewson, Azat Khuzhin, and Niels Provos. libevent - an event notification library. https://libevent.org, September 2019.

[34] 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.

[35] Aleksey Pesterev, Jacob Strauss, Nickolai Zeldovich, and Robert T. Morris. Improving Network Connection Locality on Multicore Systems. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 337–350, New York, NY, USA, 2012. ACM.

[36] Simon Peter, Jialin Li, Irene Zhang, Dan RK Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The Operating System is the Control Plane. *ACM Transactions on Computer Systems (TOCS)*, 33(4):1–30, 2015.

[37] 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.

[38] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. Arachne: Core-aware Thread Management. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, pages 145–160, Berkeley, CA, USA, 2018. USENIX Association.

[39] Mark E. Russinovich, David A. Solomon, and Alex Ionescu. *Windows Internals, Part 2*. Microsoft Press, Redmond, Washington, Sixth edition, 2012.

[40] Yasir Shoaib and Olivia Das. Web Application Performance Modeling Using Layered Queueing Networks. *Electronic Notes in Theoretical Computer Science*, 275:123–142, 2011.

[41] Tencent Cloud. F-Stack | High Performance Network Framework Based On DPDK. http://www.f-stack.org/, 2020.

[42] The Open Group. lio_listio - list directed I/O. https://pubs.opengroup.org/onlinepubs/9699919799/functions/lio_listio.html, 2017.