Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads

Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan, *MIT CSAIL*

https://www.usenix.org/conference/nsdi19/presentation/ousterhout

This paper is included in the Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19).

February 26–28, 2019 • Boston, MA, USA

ISBN 978-1-931971-49-2

Open access to the Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19) is sponsored by

NetApp®

Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads

Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, Hari Balakrishnan
MIT CSAIL

Abstract

Datacenter applications demand microsecond-scale tail latencies and high request rates from operating systems, and most applications handle loads that have high variance over multiple timescales. Achieving these goals in a CPU-efficient way is an open problem. Because of the high overheads of today's kernels, the best available solution to achieve microsecond-scale latencies is kernel-bypass networking, which dedicates CPU cores to applications for spin-polling the network card. But this approach wastes CPU: even at modest average loads, one must dedicate enough cores for the *peak* expected load.

Shenango achieves comparable latencies but at far greater CPU efficiency. It reallocates cores across applications at very fine granularity—every 5 μs—enabling cycles unused by latency-sensitive applications to be used productively by batch processing applications. It achieves such fast reallocation rates with (1) an efficient algorithm that detects when applications would benefit from more cores, and (2) a privileged component called the IOKernel that runs on a dedicated core, steering packets from the NIC and orchestrating core reallocations. When handling latency-sensitive applications, such as memcached, we found that Shenango achieves tail latency and throughput comparable to ZygOS, a state-of-the-art, kernel-bypass network stack, but can linearly trade latency-sensitive application throughput for batch processing application throughput, vastly increasing CPU efficiency.

1 Introduction

In many datacenter applications, responding to a single user request requires responses from thousands of software services. To deliver fast responses to users, it is necessary to support high request rates and microsecond-scale tail latencies (e.g., 99.9$^{\text{th}}$ percentile) [10, 24, 28, 56, 67]. This is particularly important for requests with service times of only a couple of microseconds (e.g., memcached [43] or RAMCloud [57]). Networking hardware has risen to the occasion; high-speed networks today provide round-trip times (RTTs) on the order of a few μs [54, 55]. However, when applications run atop current operating systems and network stacks, latencies are in the *milliseconds*.

At the same time, as Moore's law slows and network rates rise [26], CPU efficiency becomes paramount. In large-scale datacenters, even small improvements in CPU efficiency (the fraction of CPU cycles spent performing useful work) can save millions of dollars [72]. As a result, datacenter operators commonly fill any cores left unused by latency-sensitive tasks with batch-processing applications so they can keep CPU utilization high as load varies over time [16]. For example, Microsoft Bing colocates latency-sensitive and batch jobs on over 90,000 servers [34], and the median machine in a Google compute cluster runs eight applications [76].

Unfortunately, existing systems do a poor job of achieving high CPU efficiency when they are also required to maintain microsecond-scale tail latency. Linux can only support microsecond latency when CPU utilization is kept low, leaving enough idle cores available to quickly handle incoming requests [41, 43, 76]. Alternatively, kernel-bypass network stacks such as ZygOS are able to support microsecond latency at higher throughput by circumventing the kernel scheduler [2, 18, 50, 57, 59, 61]. However, these systems still waste significant CPU cycles; instead of interrupts, they rely on spin-polling the network interface card (NIC) to detect packet arrivals, so the CPU is always in use even when there are no packets to process. Moreover, they lack mechanisms to quickly reallocate cores across applications, so they must be provisioned with enough cores to handle peak load.

This tension between low tail latency and high CPU efficiency is exacerbated by the bursty arrival patterns of today's datacenter workloads. Offered load varies not only over long timescales of minutes to hours, but also over timescales as short as a few microseconds. For example, micro bursts in Google's Gmail servers cause sudden 50% increases in CPU usage [12], and, in Microsoft's Bing service, 15 threads can become runnable in just 5 μs [34]. This variability requires that servers leave extra cores idle at all times so that they can keep tail latency low during bursts [16, 34, 41].

Why do today's systems force us to waste cores to maintain microsecond-scale latency? A recent paper from Google argues that poor tail latency and efficiency are the result of system software that has been tuned for millisecond-scale I/O (e.g., disks) [15]. Indeed, today's schedulers only make thread balancing and core allocation decisions at coarse granularities (every four milliseconds for Linux and 50–100 milliseconds for Arachne [63] and IX [62]), preventing quick reactions to load imbalances.

This paper presents **Shenango**, a system that focuses

on achieving three goals: (1) microsecond-scale end-to-end tail latencies and high throughput for datacenter applications; (2) CPU-efficient packing of applications on multi-core machines; and (3) high application developer productivity, thanks to synchronous I/O and standard programming abstractions such as lightweight threads and blocking TCP network sockets.

To achieve its goals, Shenango solves the hard problem of reallocating cores across applications at very fine time scales; it reallocates cores every 5 microseconds, orders of magnitude faster than any system we are aware of. Shenango proposes two key ideas. First, Shenango introduces an efficient algorithm that accurately determines when applications would benefit from additional cores based on runnable threads and incoming packets. Second, Shenango dedicates a single busy-spinning core per machine to a centralized software entity called the *IOKernel*, which steers packets to applications and allocates cores across them. Applications run in user-level *runtimes*, which provide efficient, high-level programming abstractions and communicate with the IOKernel to facilitate core allocations.

Our implementation of Shenango uses existing Linux facilities, and we have made it available at https://github.com/shenango. We found that Shenango achieves similar throughput and latency to ZygOS [61], a state-of-the-art kernel-bypass network stack, but with much higher CPU efficiency. For example, Shenango can achieve over five million requests per second of memcached throughput while maintaining 99.9th percentile latency below 100 μs (one million more than ZygOS). However, unlike ZygOS, Shenango can linearly trade memcached throughput for batch application throughput when request rates are lower than peak load. To our knowledge, Shenango is the first system that can both multiplex cores and maintain low tail latency during microsecond-scale bursts in load. For example, Shenango's core allocator reacts quickly enough to keep 99.9th percentile latency below 125 μs even during an extreme shift in load from one hundred thousand to five million requests per second.

2 The Case Against Slow Core Allocators

In this section, we explain why millisecond-scale core allocators are unable to maintain high CPU efficiency when handling microsecond-scale requests. We define *CPU efficiency* as the fraction of cycles spent doing application-level work, as opposed to busy-spinning, context switching, packet processing, or other systems software overhead.

Modern datacenter applications experience request rate and service time variability over multiple timescales [16]. To provide low latency in the face of these fluctuations, most kernel bypass network stacks, including ZygOS [61], statically provision cores for peak load, wasting significant cycles on busy polling. Recently, efforts such as IX [62] and Arachne [63] introduced user-level core allocators that adjust core allocations at 50–100 millisecond intervals. Similarly, Linux rebalances tasks across cores primarily in response to millisecond-scale timer ticks. Unfortunately, all of these systems adjust cores too slowly to handle microsecond-scale requests efficiently.

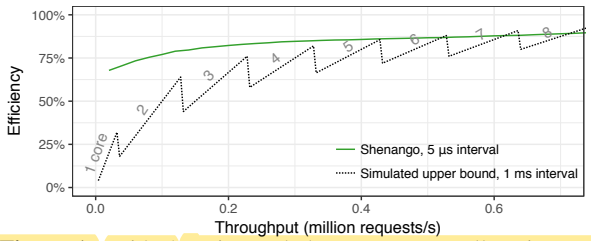

Figure 1: With 5 μs intervals between core reallocations, a Shenango runtime achieves higher CPU efficiency than an optimal simulation of a 1 ms core allocator.

To show why, we built a simulator that determines a conservative upper-bound on the CPU efficiency of a core allocator that adjusts cores at one millisecond intervals. The simulator models an M/M/n/FCFS queuing system and determines through trial and error the minimum number of cores needed to maintain a tail latency limit for a given level of offered load. We assume a Poisson arrival process (empirically shown to be representative of Google's datacenters [53]), exponentially distributed service times with a mean of 10 μs, and a latency limit of 100 μs at the 99.9th percentile. To eliminate any time dependence on past load, we also assume that the arrival queue starts out empty at the beginning of each one millisecond interval and that all pending requests can be processed immediately at the end of each millisecond interval. Together, these assumptions allow us to calculate the best case CPU efficiency regardless of the core allocation algorithm used.

Figure 1 shows the relationship between offered load and CPU efficiency (cycles used divided by cycles allocated) for our simulation. It also shows the efficiency of a Shenango runtime running the same workload locally by spawning a thread to perform synthetic work for the duration of each request. For the simulated results, we label each line segment with the number of cores assigned by the simulator; the sawtooth pattern occurs because it is only possible to assign an integer number of cores. Even with zero network or systems software overhead, mostly idle cores must be reserved to absorb bursts in load, resulting in a loss in CPU efficiency. This loss is especially

severe between one and four cores, and as load varies over time, applications are likely to spend a significant amount of time in this low-efficiency region. The ideal system would spin up a core for exactly the duration of each request and achieve perfect efficiency, as application-level work would correspond one-to-one with CPU cycles. Shenango comes close to this ideal, yielding significant efficiency improvements over the theoretical upper bound for a slow allocator, despite incurring real-world overheads for context switching, synchronization, etc.

On the other hand, a slow core allocator is likely to perform worse than its theoretical upper bound in practice. First, CPU efficiency would be even lower if there were more service time variability or tighter tail-latency requirements. Second, if the average request rate were to change during the adjustment interval, latency would spike until more cores could be added; in Arachne, load changes result in latency spikes lasting a few hundred milliseconds (§7.2) and in IX they last 1-2 seconds [62]. Finally, accurately predicting the relationship between number of cores and performance over millisecond intervals is extremely difficult; both IX and Arachne rely on load estimation parameters that may need to be hand tuned for different applications [62, 63]. If the estimate is too conservative, latency will suffer, and, if it is too liberal, unnecessary cores will be wasted. We now discuss how Shenango's fast core allocation rate allows it to overcome these problems.

3 Challenges and Approach

Shenango's goal is to optimize CPU efficiency by granting each application as few cores as possible while avoiding a condition we call *compute congestion*, in which failing to grant an additional core to an application would cause work to be delayed by more than a few microseconds. This objective frees up underused cores for use by other applications, while still keeping tail latency in check.

Modern services often experience very high request rates (millions of packets per second on a single server), and core allocation overheads make it infeasible to scale to per-request core reallocations. Instead, Shenango closely approximates this ideal, detecting load changes every five microseconds and adjusting core allocations over 60,000 times per second. Such a short adjustment interval requires new approaches to estimating load. We now discuss these challenges in more detail.

Core allocations impose overhead. The speed at which cores can be reallocated is ultimately limited by reallocation overheads: determining that a core should be reallocated, instructing an application to yield a core, etc. Existing systems impose too much overhead for microsecond-scale core reallocations to be practical: Arachne requires 29 microseconds of latency to reallocate a core [63], and IX requires hundreds of microseconds because it must update NIC rules for steering packets to cores [62].

Estimating required cores is difficult. Previous systems have used application-level metrics such as latency, throughput, or core utilization to estimate core requirements over long time scales [22, 34, 48, 63]. However, these metrics cannot be applied over microsecond-scale intervals. Instead, Shenango aims to estimate instantaneous load, but this is non-trivial. While requests arriving over the network provide one source of load, applications themselves can independently spawn threads.

3.1 Shenango's Approach

Shenango addresses these challenges with two key ideas. First, Shenango considers both thread and packet queuing delays as signals of compute congestion, and it introduces an efficient *congestion detection algorithm* that leverages these signals to decide if an application would benefit from more cores. This algorithm requires fine-grained, high-frequency visibility into each application's thread and packet queues. Thus, Shenango's second key idea is to dedicate a single, busy-spinning core to a centralized software entity called the *IOKernel* (§4). The IOKernel process runs with root privileges, serving as an intermediary between applications and NIC hardware queues. By busy-spinning, the IOKernel can examine thread and packet queues at microsecond-scale to orchestrate core allocations. Moreover, it can provide low-latency access to networking and enable steering of packets to cores in software, allowing packet steering rules to be quickly reconfigured when cores are reallocated. The result is that core reallocations complete in only 5.9 μs and require less than two microseconds of IOKernel compute time to orchestrate. These overheads support a core allocation rate that is fast enough to both adapt to shifts in load and quickly correct any mispredictions in our congestion detection algorithm.

Application logic runs in per-application *runtimes* (§5), which communicate with the IOKernel via shared memory (Figure 2). Each runtime is untrusted and is responsible for providing useful programming abstractions, including threads, mutexes, condition variables, and network sockets. Applications link with the Shenango runtime as a library, allowing kernel-like functions to run within their address spaces.

At start-up, the runtime creates multiple kernel threads (i.e., pthreads), each with a local runqueue, up to the maximum number of cores the runtime may use.

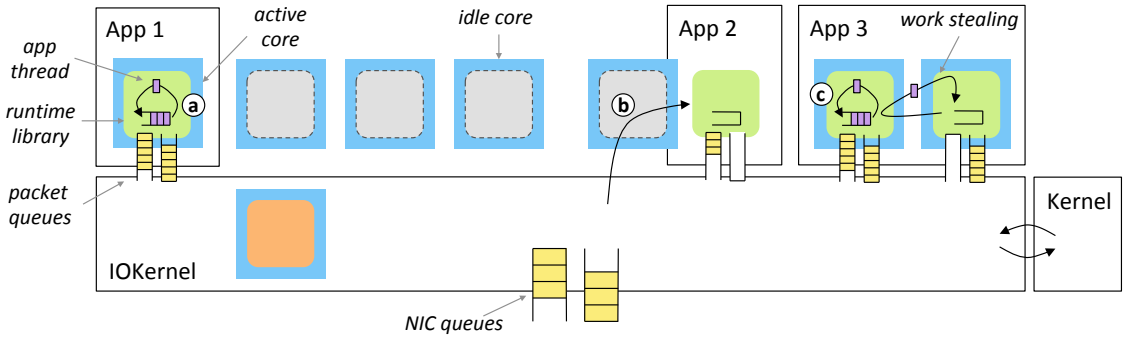

Figure 2: Shenango architecture. (a) User applications run as separate processes and link with our kernel-bypass runtime. (b) The IOKernel runs on a dedicated core, forwarding packets and allocating cores to runtimes. (c) The runtime schedules lightweight application threads on each core and uses work stealing to balance load.

Application logic runs in lightweight user-level threads that are placed into these queues; work is balanced across cores via work stealing. We refer to each per-core kernel thread created by the runtime as a *kthread* and to the user-level threads as *uthreads*. Shenango is designed to coexist inside an unmodified Linux environment; the IOKernel can be configured to manage a subset of cores while the Linux scheduler manages others.

4 IOKernel

The IOKernel runs on a dedicated core and performs two main functions:

1. At any given time, it decides how many cores to allocate to each application (§4.1.1) and which cores to allocate to each application (§4.1.2).
2. It handles all network I/O, bypassing the kernel. On the receive path, it directly polls the NIC receive queue and places each incoming packet onto a shared memory queue for one of the application's cores. On the transmission path, it polls each runtime's packet egress queues and forwards packets to the NIC (§4.2).

4.1 Core Allocation

The IOKernel must make core allocation decisions quickly because any time it spends on core allocations cannot be spent forwarding packets, thereby decreasing throughput. For simplicity, the IOKernel decouples its two decisions; in most cases, it first decides if an application should be granted an additional core, and then decides which core to grant.

4.1.1 Number of cores per application

Each application's runtime is provisioned with a number of *guaranteed cores* and a number of *burstable cores*. A runtime is always entitled to use its guaranteed cores without risk of preemption (oversubscription is not allowed), but it may use fewer (even zero) cores if it does not have enough work to occupy them. When extra cores are available, the IOKernel may allocate them as burstable cores, allowing busy runtimes to temporarily exceed their guaranteed core limit.

When deciding how many cores to grant a runtime, the IOKernel's objective is to minimize the number of cores allocated to each runtime, while still avoiding compute congestion (§3). To determine when a runtime has more cores than necessary, the IOKernel relies on runtime kthreads to voluntarily yield cores when they are unneeded. When a kthread cannot find any work to do, meaning its local runqueue is empty and it did not find stealable work from other active kthreads, it cedes its core and notifies the IOKernel (we refer to this as *parking*). The IOKernel may also preempt burstable cores at any time, forcing them to park immediately.

The IOKernel leverages its unique vantage point to detect incipient compute congestion by monitoring the queue occupancies of active kthreads. When a packet arrives for a runtime that has no allocated cores, the IOKernel immediately grants it a core. To monitor active runtimes for congestion, the IOKernel invokes the *congestion detection algorithm* at 5 μs intervals (Algorithm 1).

The congestion detection algorithm determines whether a runtime is overloaded or not based on two sources of load: queued threads and queued ingress packets. If any item is found to be present in a queue for two consecutive runs of the detection algorithm, it indicates that a packet or thread queued for at least 5 μs. Because queued packets or threads represent work that could be handled in parallel on another core, the runtime is deemed to be "congested," and the IOKernel grants it one additional core. We found that the duration of queuing is a more robust signal than the length of a queue, because using queue length requires carefully tuning a threshold parameter for different durations of requests [63, 74].

Implementing the queues as ring buffers enables a

Algorithm 1 Congestion Detection Algorithm

```
1:  for each application app do
2:      for each active kthread k of app do
3:          runq ← k's runqueue
4:          prev_runq ← k's runq last iteration
5:          inq ← k's ingress packet queue
6:          prev_inq ← k's inq last iteration
7:          if runq contains threads in prev_runq or
8:              inq contains packets in prev_inq then
9:              try to allocate a core to app
10:             break          ▷ go to next app in outer loop
```

simple and efficient detection mechanism. Detecting that an item is present in a queue for two consecutive intervals is simply a matter of comparing the current head pointer with the tail pointer from the previous iteration. Runtimes expose this state to the IOKernel in a single cache line of shared memory per kthread.

Intuitively, core allocation is capable of oscillatory behavior, potentially adding and parking a core every iteration. This is by design because slower adjustments would either sacrifice tail latency or prevent us from multiplexing cores over short timescales. Indeed, modern CPUs are capable of efficient enough context switching; Process Context Identifiers (PCIDs) allow page tables to be swapped without flushing the TLB. Linux takes about 600 nanoseconds to switch between processes, so it is fast enough to handle the core reallocation rates produced by the IOKernel. In §7.3 we evaluate the impact of different core allocation intervals on tail latency and CPU efficiency.

4.1.2 Which cores for each application

When deciding which core to grant to an application, the IOKernel considers three factors:

1. Hyper-threading efficiency. Intel's HyperThreads enable two hardware threads to run on the same physical core. These threads share processor resources such as the L1 and L2 caches and execution units, but are exposed as two separate logical cores [51]. If hyper-threads from the same application run on the same physical core, they benefit from cache locality; if hyper-threads from different applications share the same physical core, they can contend for cache space and degrade each others' performance. Thus, the IOKernel favors granting hyper-threads on the same physical core to the same application.

2. Cache locality. If an application's state is already present in the L1/L2 cache of a core it is newly granted, it can avoid many time-consuming cache misses. Because hyperthreads share the same cache resources, granting an application a hyper-thread pair of an already-running core will yield good cache locality. In addition, an application may experience cache locality benefits by running on a core that it ran on recently.[1] Thus, the IOKernel tracks current and past core allocations for runtimes.

3. Latency. Preempting a core and waiting for it to become available takes time, and wastes cycles that could be spent doing useful work. Thus, the IOKernel always grants an idle core instead of preempting a busy core, if an idle core exists.

Algorithm 2 Core Selection Algorithm

```
1:  function CanBeAllocated(core)
2:      if core is idle then return True
3:      app ← the app currently using core
4:      if n_idle_cores is 0 and app is bursting then
5:          return True
6:      return False
7:
8:  function SelectCore(app)
9:      for each active core c of app do
10:         c_hyper ← the hyper-thread pair core of c
11:         if CanBeAllocated(c_hyper) then
12:             return c_hyper
13:     c_recent ← core most recently yielded by app
14:     if CanBeAllocated(c_recent) then
15:         return c_recent
16:     if n_idle_cores > 0 then return any idle core
17:     app_bursting ← random bursting app
18:     return any core in use by app_bursting
```

The IOKernel's core selection algorithm (Algorithm 2) considers the three factors described above. A *core* is only eligible for allocation (function CanBeAllocated) if it is idle (line 2), or if there are no idle cores and the application using *core* is bursting (using more than its guaranteed number of cores) (line 4). Amongst the eligible cores, the selection algorithm SelectCore first tries to allocate the hyper-thread pair of a core the application is currently using (lines 9–12). Next, it tries to allocate the core that this application most recently used, but is no longer using (lines 13–15). Finally, the algorithm chooses any idle core if one exists, or a random core from a bursting application.

[1] This benefit is ephemeral; a core with a clock frequency of 2.2 GHz can completely overwrite a 3 MB L2 cache in as little as 60 μs.

Once the IOKernel has chosen a core to grant to an application, it must also select one of its parked kthreads to wake up and run on that core. For cache locality, it first attempts to pick one that recently ran on that core. If such a kthread is not available, the IOKernel selects the kthread that has been parked the longest, leaving other kthreads parked in case a core they ran on recently becomes available.

The runtime for SELECTCORE(APP) is linear in the number of active cores for APP (it checks whether each active core has an available hyper-thread). The congestion detection algorithm may invoke SELECTCORE up to once per active application in one pass, and the sum of active cores across active applications never exceeds the number of cores in the system. Thus the total cost of invoking the detection algorithm is linear in the total number of cores.

4.2 Dataplane

The IOKernel busy loops, continuously polling the incoming NIC packet queue and the outgoing application packet queues.

Packet steering. Because the IOKernel tracks which cores belong to each runtime, it can deliver incoming packets directly to a core running the appropriate runtime. In Shenango, each runtime is configured with its own IP and MAC address. When a new packet arrives, the IOKernel identifies its runtime by looking up the MAC address in a hash table. The IOKernel then chooses a core within that runtime using an RSS hash [4], and enqueues the packet to that core's ingress packet queue. Shenango may occasionally reorder packets (e.g., when the number of cores allocated to a runtime changes), but we found that packets in the same flow typically arrive in the same runtime ingress packet queue over short time intervals (§7.3). Our system could be extended to further optimize packet steering through techniques like Intel's Flow Director [8] or FlexNIC [42].

Polling transmission queues. Polling many egress queues in order to find packets to transmit can incur high CPU overhead, particularly in systems with many queues [68]. Because the IOKernel tracks which kthreads are active, it is able to only poll the outgoing runtime packet queues that correspond to active kthreads. This allows the CPU overhead of polling egress queues to scale with the number of cores in the system.

5 Runtime

Shenango's runtime is optimized for programmability, providing high-level abstractions like blocking TCP network sockets and lightweight threads. Our design scales to thousands of uthreads, each capable of performing arbitrary computation interspersed with synchronous I/O operations. By contrast, many previous kernel-bypass network stacks trade functionality for performance, forcing developers to use restrictive, event-driven programming models with APIs that differ significantly from Berkeley Sockets [2, 18, 40, 61].

Similar to a library OS [37, 60], our runtime is linked within each application's address space. After the runtime is initialized, applications should only interact with the Linux Kernel to allocate memory; other system calls remain available, but we discourage applications from performing any blocking kernel operations, as this could reduce CPU utilization. Instead, the runtime provides kernel-bypass alternatives to these system calls (in contrast to scheduler activations [11], which activates new threads to recover lost concurrency). As an additional benefit, memory and CPU usage, including for packet processing, can be perfectly accounted to each application because the kernel no longer performs these requests on their behalf.

Scheduling. The runtime performs scheduling within an application across the cores that are dynamically allocated to it by the IOKernel. During initialization, the runtime registers its kthreads (enough to handle the maximum provisioned number of cores) with the IOKernel and establishes a shared memory region for network packet queues. Each time the IOKernel assigns a core, it wakes one of the runtime's kthreads and binds it to that specific core.

Our runtime is structured around per-kthread runqueues and work stealing, similar to Go [6] and in contrast with Arachne's work sharing model [63]. Despite embracing this more traditional design, we found that it was possible to make our uthread handling extremely efficient. For example, because only the local kthread can append to its runqueue, uthread wakeups can be performed without locking. Inspired by ZygOS, we perform fine-grained work stealing of uthreads to reduce tail latency, which is particularly beneficial for workloads that have service time variability [61].

Our runtime also employs run-to-completion, allowing uthreads to run uninterrupted until they voluntarily yield, in most cases. This policy further reduces tail latency with light-tailed request patterns.[2] When a uthread yields, any necessary register state is saved on the stack, allowing execution to resume later. When the yield is cooperative, we can save less register state

[2] Preemption within an application, as in Shinjuku [38], could reduce tail latency for request patterns with high dispersion or a heavy tail; we leave this to future work.

because function call boundaries allow clobbering of some general purpose registers as well as all vector and floating point state [49]. However, any uthread may be preempted if the IOKernel reclaims a core; in this case all register state must be saved.

To find the next uthread to run after a yield, the scheduler first checks the local runqueue; if it is empty and there are no incoming packets or expired timers to process, it engages in work stealing. It first checks the core's hyper-thread sibling to exploit cache locality. If that fails, the scheduler tries to steal from a random kthread. Finally, the scheduler iterates through all active kthreads. It repeats these steps for a couple of microseconds, and if all attempts fail, the scheduler parks the kthread, yielding its core back to the IOKernel.

Networking. Our runtime is responsible for providing all networking functionality to the application, including UDP and TCP protocol handling. After a uthread yields or whenever the local runqueue is empty, each kthread checks its ingress packet queue for new packets to handle. Unlike previous systems, kthreads can also steal packets from remote ingress packet queues. This contrasts with ZygOS, which can steal application-level work above the TCP socket layer but must maintain flow consistent hashing of packets. Thus this stealing, along with the packet steering adjustments made by the IOKernel, can cause packet reordering over short timescales.

A variety of efficient techniques have been proposed to resequence packets [29, 30, 33]. Where ordering is required, our runtime provides a similar low overhead mechanism to reassemble the packet sequence in the transport layer. This resequencing involves acquiring a per-socket lock, but because packets from the same flow typically arrive at the same core over short time scales, cache locality is preserved and the overhead of acquiring the lock is small.

On the other hand, we found that there were significant advantages to relaxing ordering requirements and violating flow consistent hashing. ZygOS must send and receive packets from a given flow on the same core, so it relies on expensive IPIs to ensure timely processing of pending ingress packets and to ensure egress handling happens on the same core. By contrast, Shenango's approach enables more fine-grained load balancing of network flow processing, yielding better performance with imbalanced workloads (§7.3).

An earlier version of the runtime attempted to support zero-copy networking. However, we found this approach had serious drawbacks. First, it required API changes, breaking compatibility with Berkeley Sockets. Second, we were surprised to find it had a negative impact on performance. Upon further investigation, we discovered that our IOKernel's throughput was sensitive to the amount of resident buffering because DDIO (an Intel technology that pushes packet payloads directly into the LLC) places limits on the maximum number of cache lines that can be occupied by packet data. When that limit is exceeded, packet data is pushed to RAM, greatly increasing access latency. By copying payloads, we can encourage DDIO to reuse the same buffers, thus staying within its cache occupancy threshold. This bears similarity to the "leaky DMA" issue [70].

Because an application could potentially corrupt its runtime network stack, we assume security validation (e.g., bandwidth capping and network virtualization) will be efficiently handled out-of-band, in exactly the same manner as for virtual machine guest kernels [23, 27].

6 Implementation

Shenango's implementation consists of the IOKernel (§6.1), which runs as a separate, privileged process, and the runtime (§6.2), which users link with their applications. Shenango is implemented in C and includes bindings for C++ and Rust. The IOKernel is implemented in 2,244 LOC and the runtime is implemented in 6,155 LOC. Both components depend on a 4,762 LOC collection of custom library routines. The implementation currently supports 64-bit x86, and adapting it to other platforms would not require many changes. The IOKernel uses Intel Data Plane Development Kit (DPDK) [2], version 18.11, for fast access to NIC queues from user space. Our entire system runs in an unmodified Linux environment.

6.1 IOKernel Implementation

Shenango relies on several Linux kernel mechanisms to pin threads to cores and for communication between the IOKernel and runtimes. The IOKernel passes data via System-V shared memory segments that are mapped into each runtime. The runtime sets up a series of descriptor ring queues (inspired by Barrelfish's implementation of lightweight RPC [17]), including ingress packet queues, egress packet queues, and separate egress command queues (to prevent head-of-line blocking). It also designates a portion of the mapped-memory for outgoing network buffers. We currently place all ingress packet buffers in a single, read-only region shared with all runtimes. In the future, we plan to maintain separate buffers, using NIC HW filtering to segregate packets.

To assign a runtime kthread to a specific core, the IOKernel uses `sched_setaffinity`. The IOKernel maintains a shared `eventfd` file descriptor with each kthread. When a kthread cannot find more uthreads to

run, it notifies the IOKernel via a command queue message that it is parking and then parks itself by performing a blocking read on its `eventfd`. To unpark a kthread, the IOKernel simply writes a value into the `eventfd`. To preempt runtime kthreads when it needs to reassign a core, the IOKernel directs a `SIGUSR1` signal to the intended kthread using the `tgkill` system call. This prompts the kthread to park itself. A malicious kthread could refuse to park after a signal. While we have yet to implement mitigation strategies, the IOKernel could wait a few microseconds and then migrate an offending kthread to a shared core that is multiplexed by the Linux scheduler, so that other runtimes are not impacted.

6.2 Runtime Implementation

Our runtime includes support for lightweight threads, mutexes, condition variables, read-copy-update (RCU), high resolution timers, and synchronous TCP and UDP sockets. Like the IOKernel, the runtime makes use of a limited set of existing Linux primitives; it allocates memory with `mmap`, creates kthreads through calls to `pthread_create()`, and interacts with the IOKernel through shared memory, `eventfd` file descriptors, and signals. We implemented TCP from scratch according to the RFC [36]. Our TCP stack is interoperable with those of Linux and ZygOS and includes flow control and fast retransmit but omits congestion control.

To improve memory allocation performance, the runtime makes use of per-kthread caches [21], particularly when allocating thread stacks and network packet buffers. The runtime provides an RCU subsystem to support efficient access to read-mostly data structures [52]. The runtime detects a quiescent period after each kthread has rescheduled, allowing it to free any stale RCU objects. Internally, RCU is used for the ARP table and for the TCP and UDP socket tables.

Shenango provides bindings for both C++ and Rust with idiomatic interfaces (e.g., like `std::thread`) and support for lambdas and closures respectively. Most of the bindings are implemented as a thin wrapper around the underlying C library. However, our uthread support takes advantage of a unique optimization. We extended Shenango's spawn function to reserve space at the base of each uthread's stack for the trampoline data (captures, space for a return value, etc.), avoiding extra allocations.

Preemption. Upon receipt of a `SIGUSR1` sent by the IOKernel, the Linux kernel saves the CPU state into a trapframe on the thread stack and invokes the signal handler installed by the runtime. The signal handler immediately transfers to the scheduler context and parks, placing the preempted uthread back into the runqueue. The running uthread could eventually be stolen by another kthread or resume on the same kthread if it is re-granted a core.

During certain critical sections of runtime execution, preemption signals are deferred by incrementing a thread-local counter. These sections include the entire scheduler context, RCU and spinlock critical sections, and code regions that access per-kthread state. Supporting preemption of active uthreads poses some challenges. Pointers to thread-local storage (TLS) may become stale if a thread context starts executing on a different kthread. Unfortunately, gcc does not provide a way to disable caching these addresses. To our knowledge, Microsoft's C++ compiler is the only compiler to support this. As a workaround, we use our own TLS mechanisms for per-kthread data structures that are accessed outside of the scheduler context, and we currently require that applications disable preemption during accesses to thread-local variables (including glibc's `malloc` and `free`). We are considering extending the runtime to support TLS for each uthread, alleviating this burden on developers. However, the TLS data section would have to be kept small to prevent higher initialization overheads when spawning uthreads.

7 Evaluation

In evaluating Shenango, we aim to answer the following questions:

1. How do latency and CPU efficiency compare for Shenango and other systems across different workloads and service-time distributions? (§7.1)
2. How well can Shenango respond to sudden bursts in load? (§7.2)
3. What is the contribution of the individual mechanisms in Shenango to its observed performance? (§7.3)

Experimental setup. We used one dual-socket server with 12-core Intel Xeon E5-2650v4 CPUs running at 2.20 GHz, 64 GB of RAM, and a 10 Gbits/s Intel 82599ES NIC. We enabled hyper-threads and evaluated only the first socket, steering NIC interrupts, memory allocations, and threads. To reduce jitter, we disabled TurboBoost, C-states, and CPU frequency scaling. We generated load from six additional quad-core machines connected to the server through a Mellanox SX1024 switch and Mellanox ConnectX-3 Pro NICs. We used Ubuntu 18.04 with kernel version 4.15.0. We disabled kernel mitigations for Meltdown for consistency with prior results; future CPUs will support these mitigations in hardware [9].

Systems evaluated. We compare Shenango to Arachne, ZygOS, and Linux. *Arachne* is a state-of-the-art,

System	Kernel-bypass Net.	Lightweight Threading	Balancing Interval
Linux	✗	✗	4 ms
Arachne [63]	✗	✓	50 ms
ZygOS [61]	✓	✗	N/A
Shenango	✓	✓	5 μs

Table 1: Features of the systems we evaluated.

user-level threading system [63]. It achieves better tail latency and CPU efficiency than Linux by introducing a user-level core allocator that adjusts the cores assigned to each application over millisecond timescales. However, Arachne provides no network stack integration and applications typically rely on Linux kernel system calls for network I/O. *ZygOS* is a state-of-the-art, kernel-bypass network stack [61] that builds upon IX [18] to achieve better tail latency, adding fine-grained load balancing of application-level work between cores. However, it does not support threads, instead requiring developers to adopt a restrictive, event-driven API, and it can only run on a fixed set of statically provisioned cores. Finally, *Linux* is the most widely deployed of these systems in practice, but its performance, as previously studied, is limited by kernel overheads [18, 35]. Table 1 summarizes the salient differences between Shenango and these three systems.

For Arachne, we used the latest available source code [1] as of mid January 2019. We found that the default load factor of 1.5, a tuning parameter for the core allocator, yielded the best results in our experiments. For ZygOS, we similarly used the latest available source code [7]. We found that ZygOS was unstable with recent kernels, so we instead used Ubuntu 16.04 with kernel version 4.11.0.

Finally, for Linux, we used prior work [43, 45] and invested substantial effort in finding the best possible configuration. In many cases, the performance of Linux was unstable, making it challenging to measure. For example, we noticed signs of performance hysteresis, where measurement runs converged to different values despite identical configuration [77]. Increasing the number of active flows resolved this issue by allowing for more uniform RSS hashing. We ran batch tasks using `SCHED_IDLE` (a Linux scheduling policy intended for very low priority background jobs), though we found this did not improve performance much over using the lowest normal scheduler priority (niceness 19).

Applications. We evaluate *memcached* (v1.5.6), a popular key-value store that is well supported by all four systems.[3] We also wrote several new Shenango applications in Rust to measure different load patterns, taking advantage of language features like closures and move semantics. For example, we implemented a *spin-server* that emulates a compute-bound application by using the CPU for a specified duration before responding to each request. In addition, we implemented *loadgen*, a realistic load generator that can generate precisely-timed request patterns for our spin-server as well as for memcached. Combined, these two applications required 1,366 LOC. For comparing to other systems, we used variants of the ZygOS and Linux spin-servers in the ZygOS repository [7] and implemented our own spin-server for Arachne.

To support batch processing applications, we implemented a pthread shim layer for Shenango that enables it to run the entire PARSEC suite [19] without modifications. In our experiments, we use PARSEC's *swaptions* benchmark for batch processing. It computes prices of a portfolio using Monte Carlo simulations; each thread computes the price of a swaption with no synchronization or data dependencies between threads. Finally, we ported the *gdnsd* (v2.4.0) [3] DNS server, to demonstrate Shenango's UDP support. The source code for all of these applications is available on GitHub [5].

We used open-loop Poisson processes to model packet arrivals [69, 77]. Our experiments measure throughput and the 99.9$^{\text{th}}$ percentile tail response latency. All experiments use our Rust loadgen application to generate load over TCP, unless stated otherwise.

7.1 CPU Efficiency and Latency

In this section we evaluate the CPU efficiency and latency of memcached, the spin-server, and gdnsd. We use 6 client servers to generate load, enough to minimize client-side queuing delays. Each client uses 200 persistent connections (1200 total). We ramp up load gradually and measure each offered load over several seconds, so that bursts come only from the Poisson arrival process.

To ensure a fair comparison with ZygOS, which cannot support more than 16 hyperthreads with our NIC, we confine all systems to use 16 hyperthreads (8 cores) in total. Shenango must dedicate one core (2 hyperthreads) to running the IOKernel, so two fewer hyperthreads are available for applications; Arachne must dedicate one hyperthread to the core arbiter. For all but ZygOS, we also run swaptions, filling any unused cycles with lower-priority batch processing work. For ZygOS, we reserve all 16 hyperthreads for the latency-sensitive application, as required to achieve peak throughput.

Memcached. We use the USR workload from [13]: requests follow a Poisson arrival process and consist of 99.8% GET requests and 0.2% SET requests. For Shenango, we limit memcached to using at most 12 hyperthreads, because this yields the best performance

[3] We don't run LRU cache maintenance/eviction and slab rebalancing for Arachne because Arachne's memcached implementation does not support them.

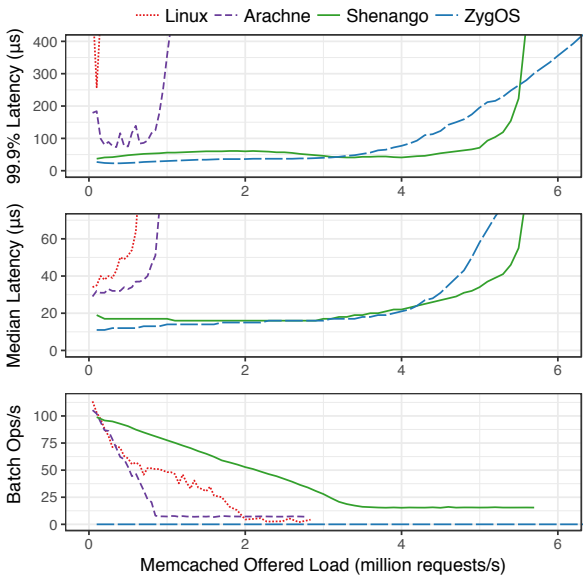

Figure 3: Shenango maintains consistently low median and 99.9% latency, comparable to those of ZygOS, while allowing unused cycles to be used by a batch processing application.

for memcached. Figure 3 shows how 99.9th percentile latency for memcached, median latency for memcached, and throughput for the batch application (y-axes) change as we increase the load offered to memcached (x-axis). We only show data points for which achieved load is within 0.1% of offered load.

Shenango can handle over five million requests per second while maintaining a median response time of 37 μs and 99.9th percentile response time of 93 μs. Despite busy polling on all 16 hyperthreads, ZygOS maintains similar response times only up to four million requests per second. ZygOS does scale to support higher throughput than Shenango, though at a high latency penalty. Shenango achieves lower throughput because at the very low service times of memcached (< 2 μs), the IOKernel becomes a bottleneck. We discuss options for scaling out the IOKernel further in Section 8. For all other systems, memcached is bottlenecked by CPU.

Similar to previous studies [18, 61], when there is no batch work running, we achieve about 800,000 requests per second with memcached in Linux before 99th percentile latency spikes (not shown). However, we found that Linux's latency degrades significantly due to the presence of batch work, especially at the 99.9th percentile. For example, at 0.4 million requests per second, the 99.9th percentile latency without batch work is only 83 μs compared to over 2 ms with batch work. Arachne improves upon Linux, maintaining 99.9th percentile latency below 200 μs with batch work. However, even without batch work, both systems suffer significantly from their use of the Linux network stack; kernel bypass enables both Shenango and ZygOS to achieve much lower median latency and much higher peak throughput for memcached.

Shenango outperforms the other systems in terms of throughput for the batch application at all but the lowest loads. At very low load, Linux achieves the most batch throughput because it does not reserve any hyperthreads for the IOKernel or the core arbiter. As the load offered to memcached increases, Shenango's batch throughput decreases linearly and then plateaus once the batch task is restricted to only the two remaining hyperthreads. Memcached throughput still increases beyond this point, however, because Shenango becomes more efficient near peak load, spending fewer cycles on core reallocations and work stealing.

In aggregate, our memcached results illustrate that Shenango has key advantages over previous systems. Shenango can achieve tail latencies similar to ZygOS while at the same time sparing significantly more cycles for batch work than all three systems, despite reserving two hyperthreads for the IOKernel.

Spin-server. To evaluate Shenango's ability to handle service-time variability in the presence of a batch processing application, we ran our spin-server with three service-time distributions, each with a mean of 10 μs: *constant*, where all requests take equal time; *exponential*; and *bimodal*, where 90% of requests take 5 μs and 10% take 55 μs.

Figure 4 shows the resulting 99.9th percentile latency and batch throughput as we vary the load on the spin-server. All systems fall short of the theoretical maximum throughput achievable by an M/G/16/FCFS simulation, due to overheads such as packet processing. Compared to ZygOS, Shenango achieves slightly higher throughput for the spin server, even though two out of Shenango's 16 hyperthreads are dedicated to running the IOKernel. Shenango's tail latency is similar to that of ZygOS, but because ZygOS must provision all cores for the spin server in order to achieve peak throughput, it does not achieve any batch throughput.

At the 99.9th percentile, Linux's tail latency varies drastically, at times reaching several milliseconds, even at low load. Arachne achieves higher throughput than Linux for both applications, demonstrating the benefit of granting applications exclusive use of their cores. Surprisingly, we observe that Arachne's tail latency is slightly higher at the lowest loads than at moderate load. We suspect that this is due to misestimation of core requirements. Granting too few cores for up to 50 ms at a time can result in high latencies for many requests, par-

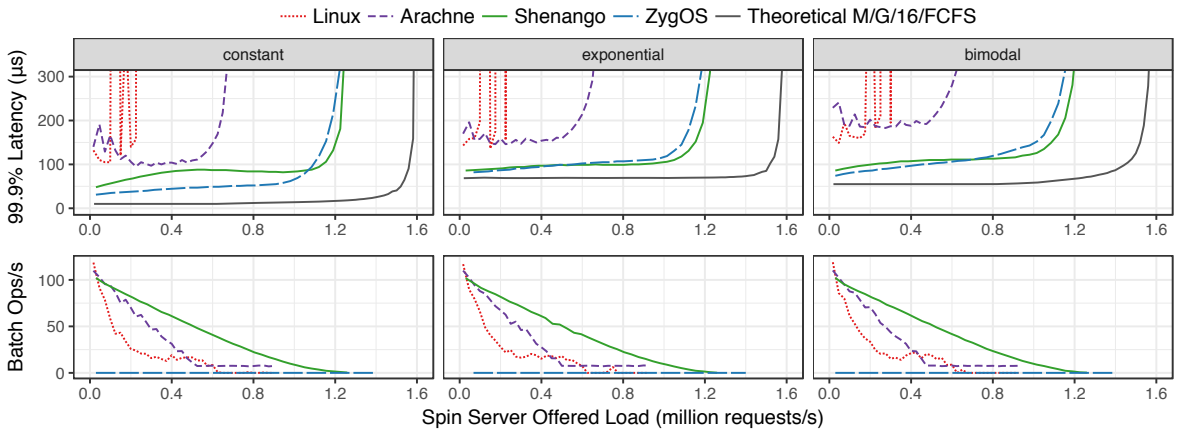

Figure 4: Shenango maintains low 99.9% latency across a variety of service time distributions (mean of 10 μs) and linearly trades off batch processing throughput for latency-sensitive throughput. Linux and Arachne suffer from poor latency and low throughput, while ZygOS must dedicate all cores to the latency-sensitive spin server in order to achieve peak throughput, resulting in no batch throughput.

ticularly at low loads when there are few cores allocated to absorb the extra load. We also found that decreasing Arachne's core allocation interval to 1 ms or 100 μs yielded similar or worse performance for both the spin server and batch application, suggesting that Arachne's load estimation mechanisms are not well-tuned for small core allocation intervals. In contrast, in this experiment Shenango reallocates cores up to 60,000 times per second, enabling it to adjust quickly to bursts in load and maintain much lower tail latency, while granting unused cycles to the batch application.

DNS. We evaluate UDP performance by running *gdnsd* and swaptions simultaneously for Linux and Shenango; we did not port *gdnsd* to ZygOS or Arachne. Linux gdnsd can drive up to 900,000 requests per second with 41 μs median latency and sub-millisecond 99.9$^{\text{th}}$ percentile latency before starting to drop packets. Shenango gdnsd is capable of scaling to 5.7 million requests per second (a 6.33$\times$ improvement) with 36 μs median latency and 73 μs 99.9$^{\text{th}}$ percentile latency. We omit a graph due to space constraints.

7.2 Resilience to Bursts in Load

In this experiment, we generate TCP requests with 1 μs of fake work, and measure the impact of sudden load increases on tail latency. We offer a baseline load of 100,000 requests per second for one second, followed by an instantaneous increase to an elevated rate. After an additional second at the new rate, the load drops back to the baseline rate. Any unused cores are allocated to batch processing, keeping overall CPU utilization at 100%.

Figure 5 shows the 99.9$^{\text{th}}$ percentile tail latency and throughput for Arachne and Shenango (computed over 10 ms windows). We exclude Linux because, under these conditions, it has milliseconds of tail latency even at the lowest offered load, and we exclude ZygOS because it cannot adjust core allocations. By contrast, Arachne can eventually meet the loads offered in the experiment, up to 1 million requests per second. However, because of its slow core allocation speed, it can take over 500 milliseconds to add enough cores to adapt after a load transition, causing it to accumulate a backlog of pending requests. As a result, Arachne experiences milliseconds of tail latency, even after relatively modest shifts in load. By contrast, Shenango reacts so quickly that it incurs almost no additional tail latency, even when handling an extreme load shift from 100,000 to 5 million requests per second.

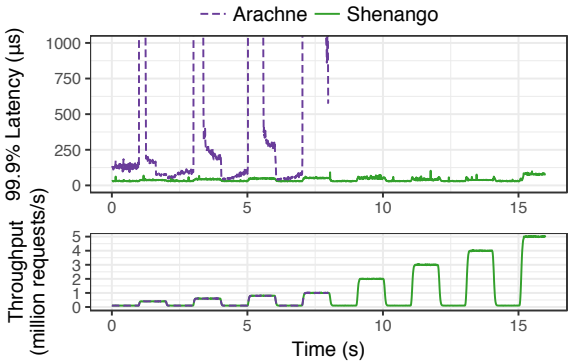

Figure 5: Under sudden changes in load, low tail latency is only possible with a short core allocation interval.

7.3 Microbenchmarks

We now evaluate the individual components of Shenango with microbenchmarks.

Thread library. Shenango depends on efficient thread scheduling to support high-level programming abstractions at low cost. Here we compare Shenango's latency for common threading operations to Linux pthreads and to Go and Arachne's optimized user space threading implementations (Table 2). These benchmarks are

	pthreads	Go	Arachne	Shenango
Uncontended Mutex	30	24	55	37
Yield Ping Pong	593	109	79	52
Condvar Ping Pong	1,900	281	203	100
Spawn-Join	12,996	462	595	148

Table 2: Nanoseconds to perform common threading operations (fastest highlighted in green). Shenango performs best for all but mutexes.

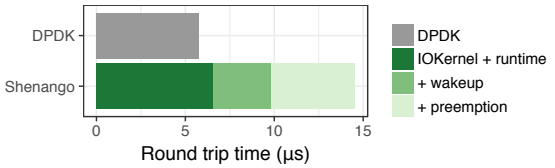

Figure 6: Traversing the network stack, waking a kthread, and preempting a kthread each add only a few μs of overhead to a packet's RTT in Shenango.

written in C++ and configure each system to use a single core. Shenango outperforms all three systems in all but one benchmark because of its preallocated stacks, atomic-free wakeups, and care to avoid saving registers that can safely be clobbered. In Go, mutexes are slightly faster because its compiler can inline them.

Network stack and core allocation overheads. We evaluate the baseline latency of our network stack and the overhead of waking and preempting cores with a simple C/C++ UDP echo benchmark. The client is a minimal DPDK client. On the server side, we compare a minimal DPDK server to three variants of Shenango which are configured so that: (1) the runtime core busy-spins, (2) the runtime core does not busy-spin and must be reallocated on every packet arrival, and (3) a batch application fills all cores and must be preempted on every packet arrival. Figure 6 shows that the runtime and the IOKernel add little latency over using raw packets in DPDK. Waking sleeping kthreads and preempting running kthreads, however, do incur some overhead, due to the use of Linux system calls (§6.1). While we were pleasantly surprised to find that the overhead of these Linux mechanisms is acceptable, we believe they can be reduced in the future.

Packet load balancing. Shenango allows packet handling to be performed on any core; here we evaluate this approach. To challenge our system's load balancing, we replicate the central graph of Figure 4 but vary the number of client connections used. With only 24 connections, RSS distributes flows unevenly across cores. Figure 7 shows that by allowing cores to steal packet processing work, including TCP protocol handling, Shenango is able to maintain good performance even with an unbalanced workload. In contrast, ZygOS's latency degrades significantly because it only allows work stealing at the application layer and performs all packet processing on the core

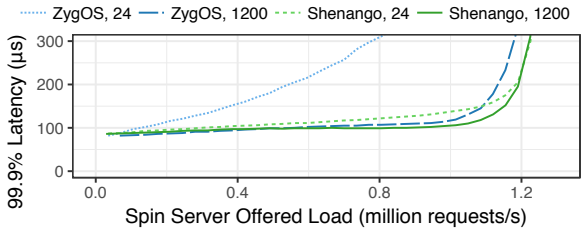

Figure 7: By work stealing packet handling, Shenango can load balance more effectively than ZygOS and maintain almost as good performance with 24 client connections as with 1200.

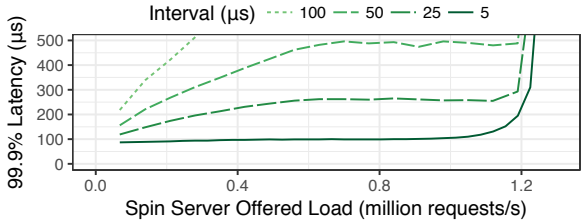

Figure 8: Shenango's tail latency degrades with larger core allocation intervals.

on which a packet arrives. At the same time, the costs of Shenango's fine-grained work stealing remain quite low. With 1200 connections, less than 0.07% of packets arrive at Shenango's ingress network stack out of order. With 24 connections, this percentage increases at moderate loads but remains below 3%. The result is that the application spends less than 0.5% of its cycles resequencing packets.

Core allocation interval. A major strength of Shenango is its ability to make μs-scale adjustments to the allocation of cores to runtimes. To illustrate the impact of core allocation speed on Shenango's performance, we replicate the central graph of Figure 4 but vary the interval between core allocations. Figure 8 demonstrates that a short interval between adjustments is required to maintain low tail latency. Such frequent reallocations do impact CPU efficiency; the batch application performs up to 6% fewer operations per second (of the max possible) with a 5 μs interval than with a 25, 50, or 100 μs interval. However, we do not think these efficiency savings are worth the tail latency increase of at least 150 μs. We did not use a smaller interval because, at faster rates, latency is only marginally improved but more cycles are wasted parking threads.

8 Discussion

We found, in practice, that the IOKernel can support packet rates of up to 6.5 million incoming and outgoing packets per second. This is sufficient to saturate a 10 Gbits/s NIC with 114 byte TCP packets or a 40 Gbits/s NIC with typical Ethernet MTU-sized packets. We note our evaluation of Shenango does not consider multisocket, NUMA machines. One option may be to run multiple instances of the IOKernel, one per socket.

Each IOKernel instance could exchange messages with the others, perhaps enabling coarse-grained load balancing between sockets. Such a design would enable our IOKernel to scale out further. We observed that the majority of IOKernel overhead was in forwarding packets rather than in orchestrating core allocations. Therefore, we also plan to explore hardware offloads, such as new NIC designs that can efficiently expose information about queuing buildups to the IOKernel.

9 Related Work

Two-level scheduling: In two-level scheduling (first proposed in [71]), a first-level spatial scheduler allocates cores to applications and a second-level scheduler handles threads on top of the allocated cores. Scheduler activations [11] provide a kernel mechanism to enable two-level scheduling; this work inspired recent systems such as Tessellation [22, 47], Akaros [65], and Callisto [32]. All of these systems decouple core allocation from thread scheduling. Shenango introduces a new approach to two-level scheduling by combining the first scheduler level directly with the NIC.

User-level threading: Several systems have multiplexed user space threads across one or more cores. Examples include Capriccio [73], Lithe [58], Intel's TBB [64], μThreads [14], Arachne [63], and the Go runtime [6]. Shenango's runtime borrows many techniques from these prior works, including work stealing [20]. However, to our knowledge, no prior system is designed to tolerate core allocations and revocations at the granularity of μs.

Dynamic resource allocation: When deciding how to allocate threads or cores across applications, previous systems have employed resource controllers that monitor performance metrics, utilization, or internal queue lengths (e.g., Tessellation [22], PerfIso [34], Arachne [63], SEDA [74], and IX [62]). However, because these metrics are gathered over several milliseconds or even seconds, they are too coarse-grained to manage tail latency. Furthermore, using core utilization to estimate core requirements is only possible in systems in which cores remain allocated to applications even while they are idle or busy-spinning [34, 63]; this approach wastes CPU cycles.

Several scheduling optimizations have been proposed to reduce tail latency. For example, Heracles [48] adjusts CPU isolation mechanisms (e.g., cache partitioning), Elfen Scheduling [75] strategically disables hyperthreading lanes, and Tail Control [44] improves upon work stealing. We are interested in exploring ways of integrating these techniques with Shenango in the future.

Kernel-bypass networking: Many systems bypass the kernel to achieve low-latency networking by using RDMA, SR-IOV, or libraries such as DPDK [2] or netmap [66]. Examples include MICA [46], IX [18], Arrakis [59], mTCP [35], Sandstorm [50], FaRM [25], HERD [39], RAMCloud [57], SoftNIC [31], ZygOS [61], Shinjuku [38], and eRPC [40]. IX and eRPC process packets in batches and may provide higher throughput than Shenango for workloads with short, uniform service times and many connections to balance load across cores. ZygOS is most similar to Shenango; it builds on IX by adding work stealing to improve load balancing within an application. However, none of these systems can dynamically reallocate cores across applications at a fine granularity. Instead, they statically partition cores across applications, or else use an external control plane to reconfigure core assignments over large timescales.

10 Conclusion

This paper presented Shenango, a system that can simultaneously maintain CPU efficiency, low tail latency, and high network throughput on machines handling multiple latency-sensitive and batch processing applications. Shenango achieves these benefits through its IOKernel, a dedicated core that integrates with networking to drive fine-grained core allocation adjustments between applications. The IOKernel makes use of a congestion detection algorithm that can react to application overload in μs timescales by tracking queuing backlog information for both packets and application threads. This design allows Shenango to significantly improve upon previous kernel bypass network stacks by recovering cycles wasted on busy spinning because of the provisioning gap between minimum and peak load. Finally, our per-application runtime makes these benefits more accessible to developers by providing high-level programming abstractions (e.g., lightweight threads and synchronous network sockets) at low overhead.

11 Acknowledgments

We thank our shepherd KyoungSoo Park, the anonymous reviewers, John Ousterhout, Tom Anderson, Frans Kaashoek, Nickolai Zeldovich, and other members of PDOS for their useful feedback. We thank Henry Qin for helping us evaluate Arachne. Amy Ousterhout was supported by an NSF Fellowship and a Hertz Foundation Fellowship. This work was funded in part by a Google Faculty Award and by NSF Grants CNS-1407470, CNS-1526791, and CNS-1563826.

References

[1] Arachne: Towards Core-Aware Scheduling. https://github.com/PlatformLab/Arachne.

[2] DPDK Boosts Packet Processing, Performance, and Throughput. http://www.intel.com/go/dpdk.

[3] gdnsd – an authoritative-only dns server. http://gdnsd.org/.

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

[5] Shenango. https://github.com/shenango.

[6] The Go Programming Language. https://golang.org/.

[7] ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. https://github.com/ix-project/zygos.

[8] Intel 82599 10 GbE Controller Datasheet. https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/82599-10-gbe-controller-datasheet.pdf, 2016.

[9] Intel Analysis of Speculative Execution Side Channels. Technical report, January 2018.

[10] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data Center TCP (DCTCP). In *SIGCOMM*, 2010.

[11] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *TOCS*, 1992.

[12] D. Ardelean, A. Diwan, and C. Erdman. Performance Analysis of Cloud Applications. In *NSDI*, 2018.

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

[14] S. Barghi. uThreads: Concurrent User Threads in C++(and C). https://github.com/samanbarghi/uThreads.

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

[16] L. A. Barroso, J. Clidaras, and U. Hölzle. The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines. *Synthesis Lectures on Computer Architecture*, 2013.

[17] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The Multikernel: A new OS architecture for scalable multicore systems. In *SOSP*, 2009.

[18] A. Belay, G. Prekas, M. Primorac, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. The IX Operating System: Combining Low Latency, High Throughput, and Efficiency in a Protected Dataplane. *TOCS*, 2017.

[19] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[20] R. D. Blumofe and C. E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. *JACM*, 1999.

[21] J. Bonwick and J. Adams. Magazines and Vmem: Extending the Slab Allocator to Many CPUs and Arbitrary Resources. In *USENIX ATC*, 2001.

[22] J. A. Colmenares, G. Eads, S. Hofmeyr, S. Bird, M. Moretó, D. Chou, B. Gluzman, E. Roman, D. B. Bartolini, N. Mor, et al. Tessellation: Refactoring the OS around Explicit Resource Containers with Continuous Adaptation. In *DAC*, 2013.

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

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

[25] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. FaRM: Fast Remote Memory. In *NSDI*, 2014.

[26] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. Dark Silicon and the End of Multicore Scaling. In *ISCA*, 2011.

[27] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. M. Caulfield, E. S. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. G. Greenberg. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *NSDI*, 2018.

[28] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker. Network Requirements for Resource Disaggregation. In *OSDI*, 2016.

[29] Y. Geng, V. Jeyakumar, A. Kabbani, and M. Alizadeh. JUGGLER: A Practical Reordering Resilient Network Stack for Datacenters. In *EuroSys*, 2016.

[30] S. Ghorbani, Z. Yang, P. Godfrey, Y. Ganjali, and A. Firoozshahian. DRILL: Micro Load Balancing for Low-latency Data Center Networks. In *SIGCOMM*, 2017.

[31] S. Han, K. Jang, A. Panda, S. Palkar, D. Han, and S. Ratnasamy. SoftNIC: A Software NIC to Augment Hardware. Technical Report UCB/EECS-2015-155, Univ. California, Berkeley, 2015.

[32] T. Harris, M. Maas, and V. J. Marathe. Callisto: Co-Scheduling Parallel Runtime Systems. In *EuroSys*, 2014.

[33] K. He, E. Rozner, K. Agarwal, W. Felter, J. Carter, and A. Akella. Presto: Edge-based Load Balancing for Fast Datacenter Networks. In *SIGCOMM*, 2015.

[34] C. Iorgulescu, R. Azimi, Y. Kwon, S. Elnikety, M. Syamala, V. R. Narasayya, H. Herodotou, P. Tomita, A. Chen, J. Zhang, and J. Wang. PerfIso: Performance Isolation for Commercial Latency-Sensitive Services. In *USENIX ATC*, 2018.

[35] E. Jeong, S. Wood, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *NSDI*, 2014.

[36] P. Jon. Transmission Control Protocol: DARPA Internet Program Protocol Specification. Technical report, RFC-793, DARPA, 1981.

[37] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application Performance and Flexibility on Exokernel Systems. In *SOSP*, 1997.

[38] K. Kaffes, T. Chong, J. T. Humphries, A. Belay, D. Mazières, and C. Kozyrakis. Shinjuku: Preemptive Scheduling for μsecond-scale Tail Latency. In *NSDI*, 2019.

[39] A. Kalia, M. Kaminsky, and D. Andersen. Using RDMA Efficiently for Key-Value Services. In *SIGCOMM*, 2014.

[40] A. Kalia, M. Kaminsky, and D. Andersen. Datacenter RPCs can be General and Fast. In *NSDI*, 2019.

[41] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat. Chronos: Predictable Low Latency for Data Center Applications. In *SOCC*, 2012.

[42] A. Kaufmann, S. Peter, N. K. Sharma, T. E. Anderson, and A. Krishnamurthy. High Performance Packet Processing with FlexNIC. In *ASPLOS*, 2016.

[43] J. Leverich and C. Kozyrakis. Reconciling High Server Utilization and Sub-millisecond Quality-of-Service. In *EuroSys*, 2014.

[44] J. Li, K. Agrawal, S. Elnikety, Y. He, I. A. Lee, C. Lu, and K. S. McKinley. Work Stealing for Interactive Services to Meet Target Latency. In *PPoPP*, 2016.

[45] J. Li, N. K. Sharma, D. R. Ports, and S. D. Gribble. Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency. In *SoCC*, 2014.

[46] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *NSDI*, 2014.

[47] R. Liu, K. Klues, S. Bird, S. Hofmeyr, K. Asanovic, and J. Kubiatowicz. Tessellation: Space-Time Partitioning in a Manycore Client OS. In *HotPar*, 2009.

[48] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis. Heracles: Improving Resource Efficiency at Scale. In *ISCA*, 2015.

[49] H. Lu, M. Matz, J. Hubicka, A. Jaeger, and M. Mitchell. System V Application Binary Interface. *AMD64 Architecture Processor Supplement*, 2018.

[50] I. Marinos, R. N. Watson, and M. Handley. Network Stack Specialization for Performance. In *SIGCOMM*, 2014.

[51] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton. Hyper-Threading Technology Architecture and Microarchitecture. *Intel Technology Journal*, 2002.

[52] P. E. McKenney, S. Boyd-Wickizer, and J. Walpole. RCU Usage in the Linux Kernel: One Decade Later. Technical report, 2013.

[53] D. Meisner, C. M. Sadler, L. A. Barroso, W. Weber, and T. F. Wenisch. Power Management of Online Data-Intensive Services. In *ISCA*, 2011.

[54] Mellanox Technologies. HP and Mellanox Benchmarking Report for Ultra Low Latency 10 and 40Gb/s Ethernet Interconnect. http://www.mellanox.com/related-docs/whitepapers/HP_Mellanox_FSI%20Benchmarking%20Report%20for%2010%20%26%2040GbE.pdf, 2012.

[55] Mellanox Technologies. RoCE vs. iWARP Competitive Analysis. http://www.mellanox.com/related-docs/whitepapers/WP_RoCE_vs_iWARP.pdf, 2017.

[56] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling Memcache at Facebook. In *NSDI*, 2013.

[57] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, and S. Yang. The RAMCloud Storage System. *TOCS*, 2015.

[58] H. Pan, B. Hindman, and K. Asanović. Composing Parallel Software Efficiently with Lithe. *PLDI*, 2010.

[59] S. Peter, J. Li, I. Zhang, D. R. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The Operating System is the Control Plane. *OSDI*, 2014.

[60] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt. Rethinking the Library OS from the Top Down. In *ASPLOS*, 2011.

[61] G. Prekas, M. Kogias, and E. Bugnion. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In *SOSP*, 2017.

[62] G. Prekas, M. Primorac, A. Belay, C. Kozyrakis, and E. Bugnion. Energy Proportionality and Workload Consolidation for Latency-critical Applications. In *SoCC*, 2015.

[63] H. Qin, Q. Li, J. Speiser, P. Kraft, and J. Ousterhout. Arachne: Core-Aware Thread Management. In *OSDI*, 2018.

[64] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. 2007.

[65] B. Rhoden, K. Klues, D. Zhu, and E. Brewer. Improving Per-Node Efficiency in the Datacenter with New OS Abstractions. In *SoCC*, 2011.

[66] L. Rizzo. netmap: a novel framework for fast packet I/O. In *USENIX ATC*, 2012.

[67] S. M. Rumble, D. Ongaro, R. Stutsman, M. Rosenblum, and J. K. Ousterhout. It's Time for Low Latency. In *HotOS*, 2011.

[68] A. Saeed, N. Dukkipati, V. Valancius, V. The Lam, C. Contavalli, and A. Vahdat. Carousel: Scalable Traffic Shaping at End Hosts. In *SIGCOMM*, 2017.

[69] B. Schroeder, A. Wierman, and M. Harchol-Balter. Open Versus Closed: A Cautionary Tale. In *NSDI*, 2006.

[70] A. Tootoonchian, A. Panda, C. Lan, M. Walls, K. J. Argyraki, S. Ratnasamy, and S. Shenker. ResQ: Enabling SLOs in Network Function Virtualization. In *NSDI*, 2018.

[71] A. Tucker and A. Gupta. Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors. In *SOSP*, 1989.

[72] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at Google with Borg. In *EuroSys*, 2015.

[73] R. Von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Capriccio: Scalable Threads for Internet Services. In *SOSP*, 2003.

[74] M. Welsh, D. Culler, and E. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *SOSP*, 2001.

[75] X. Yang, S. M. Blackburn, and K. S. McKinley. Elfen Scheduling: Fine-Grain Principled Borrowing from Latency-Critical Workloads Using Simultaneous Multithreading. In *USENIX ATC*, 2016.

[76] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes. CPI^2: CPU performance isolation for shared compute clusters. In *EuroSys*, 2013.

[77] Y. Zhang, D. Meisner, J. Mars, and L. Tang. Treadmill: Attributing the Source of Tail Latency through Precise Load Testing and Statistical Inference. In *ISCA*, 2016.