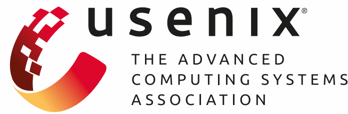


Shinjuku: Preemptive Scheduling for μsecond-scale Tail Latency

Kostis Kaffes, Timothy Chong, and Jack Tigar Humphries, *Stanford University;*
Adam Belay, *Massachusetts Institute of Technology;*
David Mazières and Christos Kozyrakis, *Stanford University*

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

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

Shinjuku: Preemptive Scheduling for μsecond-scale Tail Latency

Kostis Kaffes[1] Timothy Chong[1] Jack Tigar Humphries[1]
Adam Belay[2] David Mazières[1] Christos Kozyrakis[1]
[1] Stanford University [2] Massachusetts Institute of Technology

Abstract

The recently proposed dataplanes for microsecond scale applications, such as IX and ZygOS, use non-preemptive policies to schedule requests to cores. For the many real-world scenarios where request service times follow distributions with high dispersion or a heavy tail, they allow short requests to be blocked behind long requests, which leads to poor tail latency.

Shinjuku is a single-address space operating system that uses hardware support for virtualization to make preemption practical at the microsecond scale. This allows Shinjuku to implement centralized scheduling policies that preempt requests as often as every 5*µsec* and work well for both light and heavy tailed request service time distributions. We demonstrate that Shinjuku provides significant tail latency and throughput improvements over IX and ZygOS for a wide range of workload scenarios. For the case of a RocksDB server processing both point and range queries, Shinjuku achieves up to 6.6× higher throughput and 88% lower tail latency.

1 Introduction

Popular cloud applications such as web search, social networks, and machine translation fan out requests to hundreds of communicating services running on thousands of machines. End-to-end response times are then dominated by the slowest machine to respond [23]. Reacting to user actions within tens of milliseconds requires that each participating service process requests with *tail latency* in the range of ten to a few hundred microseconds [14]. Unfortunately, thread management in modern operating systems such as Linux is not designed for microsecond-scale tasks and frequently produces long, unpredictable scheduling delays resulting in millisecond-scale tail latencies [36, 37].

To compensate, researchers have developed network stacks, dataplanes, and full applications that bypass the operating system [44, 31, 39, 32, 16, 45, 22]. Most of these systems operate in a similar way: the NIC uses receive-side scaling (RSS) [21] to distribute incoming requests across multiple queues in a flow-consistent manner; a polling thread serves requests in each queue in a first-come-first-served manner (FCFS) without scheduling interruptions; optimizations such as zero copy, run-to-completion, adaptive batching, and cache-friendly and thread-private data structures reduce overheads. The resulting request scheduling is known as *distributed queuing and FCFS scheduling*, or *d-FCFS*.

d-FCFS is effective when request service times exhibit low dispersion [57], as is the case for get/put requests to simple in-memory key-value stores (KVS) such as Memcached [43]. d-FCFS fares poorly under high dispersion or heavy-tailed request distributions (e.g., bimodal, log-normal, Zipf, or Pareto distributions), as short requests get stuck behind older long ones assigned to the same queue. d-FCFS is also not work-conserving, an effect exacerbated by implementations based on RSS's flow-consistent hashing, which approximates true d-FCFS only with high numbers of client connections spreading requests out evenly over queues.

ZygOS [46] improved on d-FCFS by implementing low-overhead task stealing: threads that complete short requests steal work from threads tied up by longer ones. It approximates *centralized FCFS scheduling (c-FCFS)*, in which all threads serve a single queue. Work stealing is not free. It requires scanning queues cached on non-local cores and forwarding system calls back to a request's home core. However, if service times exhibit low dispersion and there are enough client connections for RSS to spread requests evenly across queues, stealing happens infrequently.

Unfortunately, c-FCFS is also inefficient for workloads with request times that follow heavy-tailed distributions or even light-tailed distributions with high dispersion. These workloads include search engines that score and rank a number of items depending on the popularity of search terms [13]; microservices and function-as-a-service (FaaS) frameworks [17]; and in-memory stores or databases, such as RocksDB [26], Redis [35], and Silo [54], that support both simple get/put requests

and complex range or SQL queries, and use slower non-volatile memory in addition to fast DRAM. Theory tells us that such workloads do best in terms of tail latency under *processor sharing (PS)* [57], where all requests receive a fine-grain, fair fraction of the available processing capacity.

To approximate PS, we need *preemption*, as built into any modern kernel scheduler including Linux. However, any service that uses one thread per request or connection and allows Linux to manage threads will experience *millisecond*-scale tail latencies, because the kernel employs preemption at millisecond granularities and its policies are not optimized for microsecond-scale tail latency [36, 37]. User-level libraries for cooperative threading can avoid the overheads of kernel scheduling [56]. However, it is difficult to yield often enough during requests with longer processing times and without many blocking I/O calls, which are precisely the requests impacting tail latency.

This paper presents *Shinjuku*, a single-address space operating system that implements *preemptive scheduling at the microsecond-scale* and improves tail latency and throughput for *both light- and heavy-tailed service time distributions*. Shinjuku deemphasizes RSS in favor of true centralized scheduling by one or more dedicated dispatcher threads with centralized knowledge of load and service time distribution. It leverages hardware support for virtualization—specifically posted interrupts—to achieve preemption overheads of 298 cycles in the dispatcher core and 1,212 cycles in worker cores. The single address space architecture allows us to optimize context switches down to 110 cycles.

Fast preemption enables scheduling policies that switch between requests as often as every 5 μsec when needed. We developed two policies. The first assumes no prior knowledge of request service times and uses preemption to select at fine granularity between FCFS or PS based on observed service times. The second policy assumes we can segregate requests with different service level objectives (SLO) in order to ensure good tail latency for both short and long requests. Both policies are work conserving and work well across multiple distributions of service times (light-tailed, heavy-tailed, bimodal, or multimodal). The two policies make Shinjuku the first system to support microsecond-scale tail latency for workloads beyond those with fixed or low-dispersion service time distributions.

We compare Shinjuku with IX [16] and ZygOS [46], two state-of-the-art dataplane operating systems. Using synthetic workloads, we show that Shinjuku matches IX's and ZygOS' performance for light-tailed workloads while it supports up to 5x larger load for heavy-tailed and multi-modal distributions. Using RocksDB, a popular key-value store that also supports range queries, we show that Shinjuku improves upon ZygOS by up to 6.6× in terms of throughput at a given 99th percentile latency. We show that Shinjuku scales well with the number of cores available, can saturate high speed network links, and is efficient even with small connection counts.

The rest of the paper is organized as follows. §2 motivates the need for preemptive scheduling at microsecond-scale. §3 discusses the design and implementation of Shinjuku. §4 presents a thorough quantitative evaluation while §5 discusses related work.

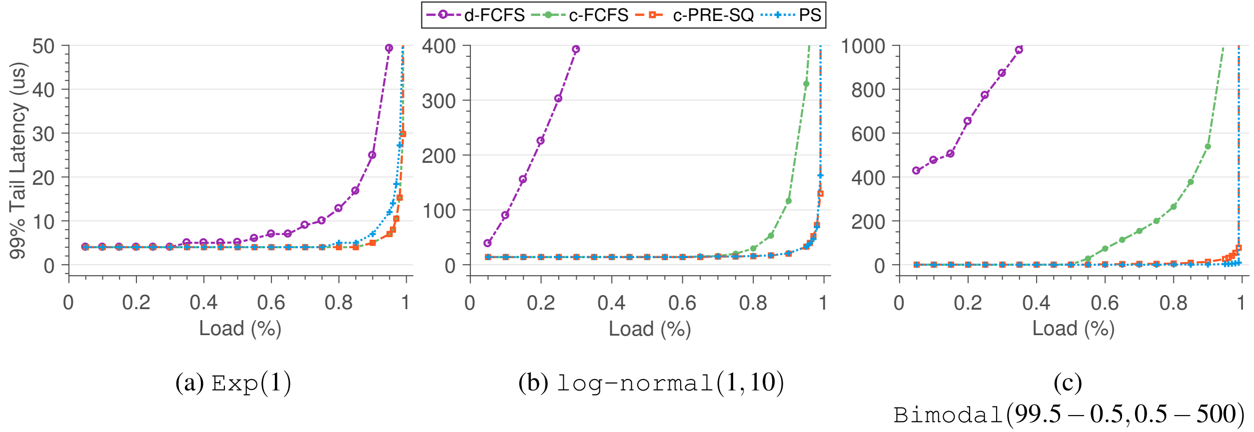
Shinjuku is open-source software. The code is available at `https://github.com/stanford-mast/shinjuku`.

2 Motivation

Background: We aim to improve the SLO of latency-critical services on a single server. For cloud services and microservices with high fan-out, Shinjuku must achieve *low tail latency* at the microsecond scale [14]. Low average or median latency is not sufficient [23]. While tail latency can be improved through overprovisioning, doing so is not economical for services with millions of users. To be cost-effective, Shinjuku must maintain low tail latency in the face of *high request throughput*. Finally, it must be *practical* for a wide range of workloads and support intuitive APIs that simplify development and maintenance of large code bases.

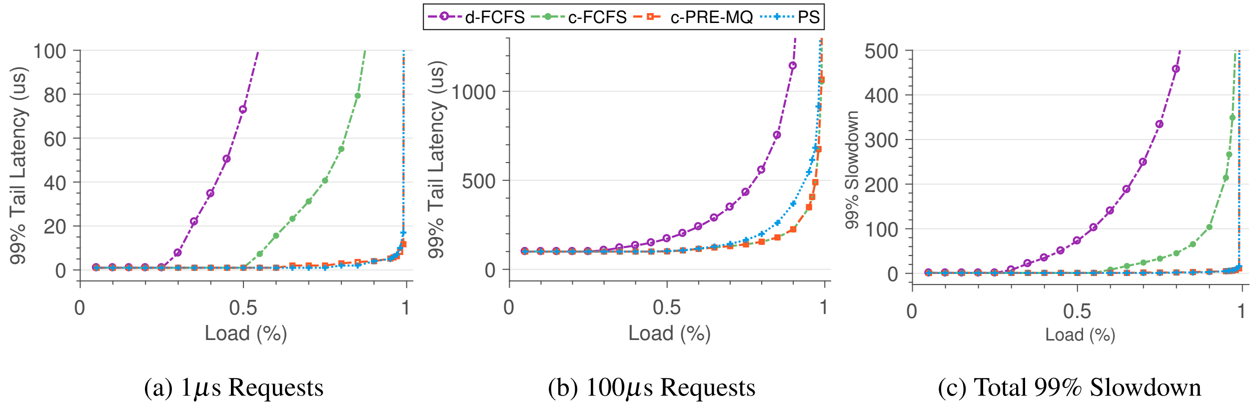
The key to achieving low tail latency and high throughput is effective request scheduling, which requires both *good policies* and *low-overhead mechanisms* that operate at the microsecond scale. Good policies are easy to achieve in isolation. Linux already supplies approximations of the optimal policies for workloads we target. Unfortunately, the Linux kernel scheduler operates at the millisecond scale because of preemption and context switch overheads and the complexity of simultaneously accommodating batch, background, and interactive tasks at different time scales [36, 37].

Recent proposals for user-level networking stacks, dataplanes, RPC protocols, and applications [22, 44, 16, 45, 31, 39, 32] sidestep the bloated kernel networking and thread management stacks in order to optimize tail latency and throughput. Most of these systems use RSS to approximate a d-FCFS scheduling policy [21], the IX dataplane being a canonical example [16]. ZygOS improves on IX by using work stealing to approximate c-FCFS [46]. Linux applications built with `libevent` [47] or `libuv` [5] also implement c-FCFS,



(a) $\texttt{Exp}(1)$ (b) $\texttt{log-normal}(1, 10)$ (c) $\texttt{Bimodal}(99.5-0.5, 0.5-500)$

Figure 1: Simulation results for different workloads and scheduling policies for a 16-core system.



(a) 1μs Requests (b) 100μs Requests (c) Total 99% Slowdown

Figure 2: Simulation results for $\texttt{Bimodal}(50-1, 50-100)$ for a 16-core system.

but at much higher overheads due to the use of interrupts for request distribution instead of RSS and polling.

Policy comparison: In order to quantify the differences between different scheduling policies, we developed a discrete event simulator. The simulator allowed us to configure parameters such as scheduling policy, number of host cores, system load, service and inter-arrival time distributions as well as various system-related overheads. Figure 1 compares idealized versions of scheduling policies—i.e., no stealing or preemption overhead—using the simulator. Plot (a) shows a light-tailed exponential distribution of service times with mean $\mu = 1$ μsec, representative of workloads such as the get/set requests of in-memory key-value stores. d-FCFS is arguably tolerable under such simple workloads, but suffers at moderate and high load as requests are not perfectly distributed across workers. c-FCFS is optimal under such workloads, while PS is slightly worse because it preempts even short requests. The PS time slice used for all simulations is 0.1 μsec.

d-FCFS is a poor option for heavy-tailed request distributions [42], as found in search engines [38] or induced by activities such as garbage collection or compaction [23, 6, 26]. Plot (b) shows performance under a heavy-tailed log-normal distribution with mean $\mu = 1$ μsec and standard deviation $\sigma = 10$ μsec. Any long request blocks every short request assigned to the same queue in d-FCFS. c-FCFS performs significantly better as a worker can service any request; short requests are only delayed when most workers simultaneously process older long requests, which is uncommon for the log-normal distribution.

c-FCFS performs significantly worse under a light-tailed bimodal distribution, commonly found in object stores and databases that mix simple get/put requests with complex range or relational queries [35, 26, 54]. Plot (c) shows such a distribution in which 99.5% of requests take $0.5\mu sec$ and 0.5% take $500\mu sec$. Compared to a heavy-tailed case, the bimodal distribution's long requests are not as long but far more frequent. PS han-

dles both cases in Plots (b) and (c) well by preempting long requests to interleave execution of short ones.

Figure 2 provides further insights by separating the performance of short and long requests in a bimodal workload with service times evenly split between 1 μsec and 100 μsec. This approximates a KVS in which half of the requests are get/put requests and the other half are range queries. The tail latency for the two request types is drawn separately in Plots (a) and (b), and Plot (c) shows the 99th percentile of the request slowdown for all requests, which is the ratio of a request's overall latency to its service time. This ratio is a useful metric for measuring how well we achieve our goal of reducing queuing time for all request types: if this ratio is small, it means that queuing time is small for all types and no requests are affected by the requests of different types.

Plot 2a shows that both d-FCFS and c-FCFS heavily penalize 1-μsec requests. Plot 2b shows that c-FCFS is marginally better than PS for 100 μsec requests, as it effectively prioritizes older, long requests that would be preempted by PS. Plot 2c shows that, in relative terms, the penalty c-FCFS inflicts on short requests dwarfs any benefit to long requests.

Shinjuku approach: Shinjuku implements the c-PRE policies (see §3.4) that achieve the best of both worlds between PS and c-FCFS as shown in Figures 1 and 2. The reason other recent systems cannot implement similar policies is that these policies require preemption at arbitrary execution points. Preemption typically involves interrupts and kernel threads whose overheads are incompatible with microsecond-scale latencies. Therefore, Shinjuku aims to achieve the following goals: 1) Implement low-overhead preemption and context switching mechanisms for user-level threads. 2) Use these mechanisms to build scheduling policies that work well across all possible distributions of service times for microsecond-scale workloads.

3 Shinjuku

Shinjuku[1] is a single-address space operating system for low-latency applications. Shinjuku is a significant departure from the common pattern in IX [16] and ZygOS [46], which rely heavily on RSS to distribute incoming requests to workers that process them without interruption. Instead, Shinjuku uses a centralized queuing and scheduling architecture and relies on low overhead and frequent preemption in order to ensure low tail latency for a wide variety of service time distributions.

[1]Shinjuku (新宿駅) is a major train station in Tokyo that serves millions traveling on 12 lines of various types and speeds.

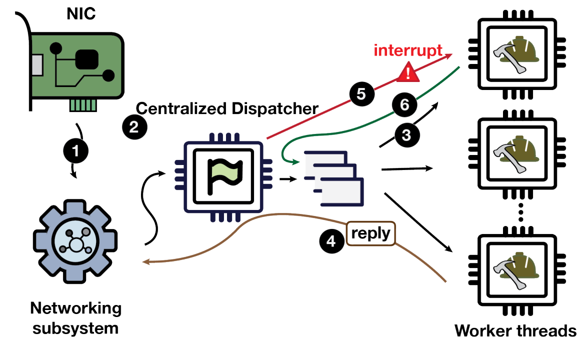


Figure 3: Shinjuku system design.

3.1 Design Overview

Figure 3 summarizes the key components in Shinjuku and the typical request flow. Incoming requests are first processed by the **networking subsystem** that handles all network protocol processing and identifies request boundaries ❶. The networking subsystem can be implemented using one or more dedicated cores or hyperthreads [22], a smartNIC [33, 53], or a combination of the two. By separating network processing from request scheduling, Shinjuku can be combined with a range of networking protocols that optimize for different conditions (UDP, TCP, ROCE [52], TIMELY [41], etc.) and various optimized network stacks [31, 28, 22]. The networking subsystem passes requests to a **centralized dispatcher** thread ❷ that will queue and schedule them to **worker threads**. The dispatcher generates a context for each incoming request in order to support preemption and rescheduling. In its simplest form, the dispatcher maintains a single queue for all pending requests. The dispatcher sends requests to worker threads ❸, each using a dedicated hardware core or hyperthread. Most requests will complete their execution without interruption. Network processing for any replies can take place either at the networking subsystem or the worker thread itself to optimize for latency. At a minimum, the worker thread notifies the networking subsystem to free any buffer space allocated for the incoming request ❹.

The dispatcher uses timestamps to identify long running requests that should be preempted based on the scheduling policy. Assuming there are queued requests, we preempt running requests after $5\mu sec$ to $15\mu sec$ for the workloads we studied (see §4), which is extremely frequent compared to the time slice in the Linux kernel. For example, the CFS scheduler has a target preemption latency of $6ms$ and a minimum one of 0.75ms. The dispatcher sends an interrupt to the worker thread ❺, which performs a context switch and receives a different request to run from the dispatcher. The long request is re-enqueued ❻ in the dispatcher and processed later us-

Preemption Mechanism	Sender Cost	Receiver Cost	Total Latency
Linux Signal	2084	2523	4950
Vanilla IPI	2081	2662	4219
IPI Sender-only Exit	2081	1212	2768
IPI Receiver-only Exit	298	2662	3433
IPI No Exits	298	1212	1993

Table 1: Average preemption overhead in cycles. Sender/receiver cost refers to cycles consumed in the sending/receiving core, including the receiver overhead of invoking an empty interrupt handler. Total cost includes interrupt propagation through the system bus. Hence, it is not equal to the sum of sender and receiver overhead. For "IPI Sender-only Exit" and "Vanilla IPI," the receiver starts interrupt processing before the sender returns from its VM exit.

ing steps ❷-❻ as many times as needed.

Since Shinjuku is a single-address space operating system, communication between its components occurs over shared memory. We use dedicated pairs of cache lines for each pair of communicating threads (see §3.5).

Similar to IX and ZygOS, Shinjuku leverages the Dune system for process virtualization [15]. With Dune, Linux and the Dune kernel module run in VMX root mode ring 0, where a hypervisor would run in a virtualized system. Shinjuku runs in VMX non-root mode ring 0, where a guest OS would run. This allows it to use very low overhead interrupts while separating the control from the data plane. The application context that uses Shinjuku can run in VMX non-root mode ring 0 or ring 3. For the results in §4, we run applications in VMX non-root mode ring 0 to avoid the address space crossings between Shinjuku and the application code. There is a separate instance of Shinjuku for each low-latency application running on the server.

3.2 Fast Preemption

To use preemptive scheduling at microsecond latencies, Shinjuku requires fast preemption. A naive approach would be for the dispatcher to notify workers using Linux signals. As we show in Table 1, however, signals incur high overheads for both the sender and the receiver (roughly 2.5 μsec on a 2GHz machine). They require user- to kernel-space transitions plus some kernel processing.

Preemption through interrupts. Direct use of interprocessor interrupts (IPIs) is potentially faster than signals. x86 processors implement IPIs using the Advanced Programmable Interrupt Controller (APIC). Each core has a local APIC and an I/O APIC is attached to the system bus. To send an IPI, the sending core writes registers in its local APIC which propagates the interrupt via the I/O APIC to the destination core's APIC, which in turn vectors execution to an interrupt handler.

We extended Dune to support IPIs by virtualizing the local APIC registers. When a non-root thread on core A writes its virtual APIC to send interrupt number V to core B, this causes a VM exit to Dune running in root mode. Dune writes V to core B's *posted interrupt descriptor*, and then uses the real APIC to send interrupt 242 to core B. That causes core B to perform a VM exit to an interrupt handler in Dune, which injects interrupt number V into non-root mode on resuming the application.

As Table 1 shows, this vanilla implementation of preemption using IPIs is slightly faster than Linux signals but still suffers from significant overheads due to the cost of VM exits in both the sender and the receiver.

Optimized interrupt delivery. We first focus on removing the VM exit on the receiving core B (the Shinjuku worker) using *posted interrupts*, an x86 feature for receiving interrupts without a VM exit. To enable posted interrupts, Dune on B configures its hardware-defined VM control structure (VMCS) to recognize interrupt 242 as the special *posted interrupt notification vector*. B also registers its posted interrupt descriptor with the VMCS. Core A still performs a VM exit upon writing the virtual APIC. Dune code on A writes V into B's posted interrupt descriptor and sends interrupt 242 to B. However, B then directly injects interrupt V without a VM exit. Table 1 shows that eliminating the receiver-side VM exit reduces receiver overhead by 54% (from 2662 to 1212 cycles). This allows frequent preemption of worker threads without significant reduction in useful worker throughput. This receiver overhead consists of modifications to hardware structures, and it cannot be significantly improved without hardware changes, such as support for lightweight user-level interrupts [51].

Optimized interrupt sending. Finally, we remove the VM exit on the sending core (dispatcher thread) by trusting the Shinjuku dispatcher with direct access to the real (non-virtual) APIC. Using the extended page table (EPT), we map both the posted interrupt descriptors of other cores and the local APIC's registers into the guest physical address space of the Shinjuku dispatcher. Hence, the dispatcher can directly send an IPI without incurring a VM exit. Table 1 shows that eliminating the sender-side VM exit reduces sender overheads down to 298 cycles (149ns in a 2GHz system). This improves

Mechanism	Linux process	Dune process
`swapcontext`	985	2290
No signal mask	140	140
No FP-restore	36	36
No FP-save	109	109

Table 2: Average overhead in clock cycles of different context-switch mechanisms for both an ordinary Linux process and the Dune process used by Shinjuku.

dispatcher scalability and allows it to serve more requests per second and/or more worker threads (cores).

Table 1 presents the result of combining the sender-side and receiver-side optimization for the interrupt delivery used to support preemption in Shinjuku. The low sender-side overhead (298 cycles) makes it practical to build a centralized, preemptive dispatcher that handles millions of scheduling actions per second. The low receiver-side overhead (1212 cycles) makes it practical to preempt requests as often as every 5μsec in order to schedule longer requests without wasting more than 10% of the workers' throughput.

3.3 Low-overhead Context Switch

When a request is scheduled to an idling core or upon preemption, we context switch between the main context in each worker and the request handling context. The direct approach would be to use the `swapcontext` function in the Linux `ucontext` library. According to Table 2, the overhead is significant in an ordinary Linux process and doubles when used in a Dune process. `swapcontext` requires a system call to set the signal masks during the switch, which requires a VM exit in Dune. The rest of the work in `swapcontext`—i.e., saving/restoring register state and the stack pointer—does not require system calls.

Table 2 evaluates context switch optimizations. First, we skip setting the signal mask which eliminates the system call and brings Dune to parity with ordinary Linux. This introduces the limitation that all tasks belonging to the same application need to share the same signal mask. Next, we exploit that the main worker context does not use floating (FP) instructions. When switching from a request context to the worker context, we must save FP registers as they may have been used in request processing, but we do not need to restore them for the worker context. When switching from the worker context to a request context, we skip saving FP registers and just restore them for the request context. Shinjuku uses the last two options in Table 2 for context switching in worker cores. The overall cost ranges from 36 to 109 cycles (18 to 55ns for a 2GHz system).

3.4 Preemptive Scheduling

The centralized dispatcher and fast preemption and context switch mechanisms allow Shinjuku to implement preemptive scheduling policies. We developed two policies that differ on whether we can differentiate *a priori* between requests types. The policies rely on frequent preemption to provide near-optimal tail latency for any workload, approximating c-FCFS for low dispersion workloads and PS for all other cases.

Single queue (SQ) policy: This policy assumes that we do not differentiate *a priori* between request types and that there is a single service-level agreement (SLO) for tail latency. This is the case, for example, in a search service where we cannot know *a priori* which requests will have longer service times. All incoming requests are placed in a single FCFS queue. When a worker is idle, the dispatcher assigns to it the request at the head of the queue. If requests are processed quickly, this policy operates as centralized FCFS. The dispatcher uses timestamps to identify any request running for more than a predefined quantum (5 to 15μsec in our experiments) and, assuming the queue is not empty, preempts it. The request is placed back in the queue and the worker is assigned the request at the current head of the queue. The *c-PRE-SQ* policy evaluated through simulation in Figure 1 is this single queue policy.

Multi queue (MQ) policy: This policy assumes that the network subsystem can identify different request types. For example, it can parse the request header for KVS like Redis and RocksDB and separate simple get/put requests from complex range query requests [33] or use different ports for different request types. Linux already supports peeking into packets with eBPF [2]. Each request type can have a different tail latency SLO. The dispatcher maintains one queue per request type. If only one queue has pending requests, this policy operates just like the single queue policy described above. If more than one queue is non empty, the dispatcher must select a queue to serve when a worker becomes idle or a request is preempted. Once the queue is selected, the dispatcher always takes the request at the head.

The queue selection algorithm is inspired by BVT [24], a process scheduling algorithm for latency sensitive tasks. In BVT, each process has a *warp factor* that quantifies its priority compared to other processes. For Shinjuku, we need a similar warp factor that favors requests with smaller target latency in the short term, but also considers aging of requests with longer latency targets. Since Shinjuku schedules requests and not long running processes with priorities like BVT, the selec-

tion algorithm shown below uses as input the target SLO latency for each queue (e.g., target 99th percentile latency). For the request at the head of each queue, the algorithm uses timestamps to calculate the ratio of the time it has already spent in the system (queuing time) to the SLO target latency for this request type. The queue with the highest such ratio is selected. The algorithm initially favors short requests that can only tolerate short queuing times, but eventually selects long requests that may have been waiting for a while. The per-queue SLO is a user-set parameter. In our experiments, we set it by running each request type individually using the single queue policy and use the observed 99% latency. This captures the requirement that the performance of a request type should not be affected by the existence of requests with different service time distributions.

1 Queue Selection Policy

```
1:  procedure QUEUESELECTION(QUEUES):
2:      max ← 0
3:      max_queue ← −1
4:      time ← timestamp()
5:      for queue in queues do
6:          cur_ratio ← (time−queue[0].timestamp) / queue.SLO
7:          if cur_ratio > max then
8:              max ← cur_ratio
9:              max_queue ← queue
10:     return max_queue
```

A preempted request can be placed either at the tail of its queue to approximate PS or at the head of the queue to approximate c-FCFS. This choice can be set by the application or based on online measurements of service time statistics. The rule of thumb we use is that for multi-modal or heavy-tailed workloads, the requests should be placed at the tail of the queue, while for light-tailed ones at the head. Frequent preemption is needed even with light-tailed distributions in order to allow Shinjuku to serve the queues for other request types. The *c-PRE-MQ* policy evaluated through simulation in Figure 2 is this multi-queue policy, where both request types are placed at the head of their corresponding queues when preempted.

3.5 Implementation

The current Shinjuku implementation is based on Dune and requires the VT-x virtualization features in x86-64 systems [20]. Dune can be ported to other architectures with similar virtualization support. Our modifications to Dune involve 1365 SLOC. The Shinjuku dispatcher and worker code are 2535 SLOC. The network subsystem we used in §4 is based on IX [16]. All the aforementioned codebases are in C.

API: To use Shinjuku, applications need to register three callback functions: the `init()` function that initializes global application state; the `init_per_core(int core_num)` function that initializes application state for each worker thread (e.g. local variables or configuration options); the `reply * handle_request(request *)` function that handles a single application-level request and returns a pointer to the reply data.

Context management: We use a modified version of the Linux `ucontext` library for context management. The context structure consists of a machine-specific representation of the saved state, the signal mask, a pointer to the context stack, and a pointer to the context that will be resumed when this context finishes execution. The dispatcher allocates context objects and stack space for each request from a memory pool. They are freed by the dispatcher when the request context completes execution and is returned by a worker thread.

Inter-thread communication: In adition to preemption, we use a low-overhead, shared memory communication scheme similar to that used in [50]. Each pair of threads, running on dedicated cores or hyperthreads, communicates over shared pairs of cache lines, one for each direction of communication. The sending thread fills the cache line with the data it wants to send, e.g. request or context pointers, as shown in Figure 3. Then, it sets the value of the byte the receiver polls to notify it that the cache line is ready for reading. This approach requires two cache line state transitions, one from shared to exclusive state which takes place when the sender writes the data and one from exclusive to shared state when the receiver reads the data. The average roundtrip latency for a message sent and received over a cache line is 211 cycles. The dispatcher's minimum work for sending a message is approximately 70 cycles, i.e. 35ns in a 2GHz machine. This sets a theoretical upper bound of 28 MRPS for the number of requests the dispatcher can handle, assuming all it has to do is to place pointers to the requests in shared memory locations and notify idling workers.

3.6 Discussion

Hardware constraints: §4 shows that a single dispatcher thread can process at least 5M requests per second and comfortably saturate a full socket with 12 cores and 24 hyperthreads. To scale a single application to higher core and/or socket counts, we must improve the dispatcher throughput. The approach we use is to have each dispatcher thread handle a subset of the worker threads and steer requests to different dispatchers using the NIC RSS feature. A relatively simple hardware fea-

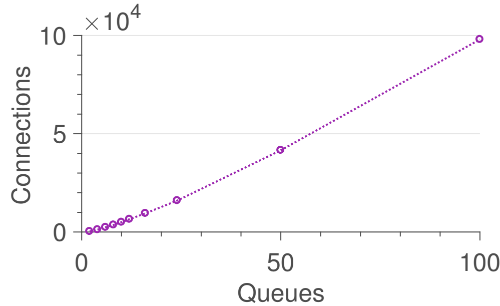


Figure 4: Number of concurrent connections needed for load imbalance among queues to be less than 10% with probability greater than 90%.

ture that would vastly improve the dispatcher scalability would be a low-overhead message passing mechanism among different cores [34, 51]. Ideally, such a mechanism would offer two variations, a preemptive one that would be used for scheduling and a non-preemptive one where messages are added to per core queues and would be used for work assignment.

Connection counts: IX and ZygOS use RSS to distribute requests to workers. Using a Monte-Carlo simulation, we calculate the connection count needed for RSS to keep imbalance below 10% with high probability as we increase the number of cores. As shown in Figure 4, they need 16,000 connections (clients or flows) to avoid imbalance on a server with 24 hyperthreads. High connection counts are common for public facing services (e.g., public load balancer or HTTP server), but not for internal ones. The DCTCP project [9] found at most a few hundred connections to back-end servers over each 1 msec window. In contrast, Shinjuku uses RSS to distribute requests to dispatchers. Since each dispatcher can manage tens of cores, Shinjuku is not subject to the requirement of high connection (clients or flows) counts discussed in §2. For example, 300 connections are sufficient to load balance across 2 dispatchers. When a single dispatcher suffices, Shinjuku will operate efficiently even with a single connection.

Alternative scheduling policies: Shinjuku can support more scheduling policies in addition to the two we presented. In future work, we will explore integrating Shinjuku with datacenter-wide profiling tools [49] and online experimentation tools [55] in order to dynamically infer the service time distributions and adjust the policy accordingly. We will also explore microsecond-scale scheduling policies that are locality- and heterogeneity-aware [30, 27]. For example, consider an application which creates a large memory footprint before responding to a client request. In such cases, we will want to avoid preempting and context switching as multiple cache lines will have to move to a different core, which can be very expensive.

Control plane: Online services experience load variations, such as a diurnal load patterns and spikes. Hence, it makes sense to adjust over time the number of workers a Shinjuku process uses. Shenango [7] solves this problem by adjusting core allocation between applications in microsecond timescales. We plan to explore the possibility of integrating the two systems.

Security model: The Dune kernel module [15] uses hardware support for virtualization to isolate a Shinjuku process from the Linux kernel and any other process, ordinary Linux or Shinjuku based. Linux can also remove cores and network queues from a Shinjuku process at any time. Within a Shinjuku process, the application code must trust the Shinjuku runtime and, if the application contexts execute in VMX non-root ring 0, the Shinjuku runtime must trust the application code. For example, the fact that APIC registers are mapped in the process address space means that one process could launch a denial-of-service attack on another process by issuing a large number of interrupts to a specific core.

We measured the cost of a ring 3 $\rightarrow$ ring 0 $\rightarrow$ ring 3 transition to be only 84 cycles. Future versions of Shinjuku will run application code in ring 3 while the Shinjuku runtime will be running in VMX non-root ring 0 eliminating this attack vector with very small overhead. Moreover, with this approach, bugs in application code will only cause contexts to crash, not affecting the runtime system.

Synchronization in user code: Online services are designed to run well on multiple cores. They synchronize across requests, but synchronization is short and infrequent to achieve scalability. Scalable applications will perform with Shinjuku regardless of whether we disable or allow preemption around read/write locks. We currently disable interrupts during any non thread-safe code, using a `call_safe(fn)` API call to simplify application porting. The runtime overhead of the instructions that are used to disable interrupts is only a few clock cycles and they do not affect the Linux kernel's abilities to reclaim the cores. Memory allocation code is a special case that often optimizes away locks using thread-local storage. We preload our own version of the C and C++ libraries that disable interrupts (and hence preemption) during the execution of allocation functions. If these functions take a long time, it will affect the tail latency observed with Shinjuku.

Any application that frequently uses coarse-grain or contested locks within requests will scale poorly regard-

less of scheduling policy on any system, including Shinjuku.

4 Evaluation

We compare Shinjuku to IX [16] and ZygOS [46], two recent systems that use d-FCFS and approximate c-FCFS respectively to improve tail latency. All three systems are built on top of Dune [15]. We use the latest IX and ZygOS versions available at [4].

4.1 Experimental Methodology

Machines: We use a cluster of 6 client and one server machines, connected through an Arista 7050-S switch with 48 10GbE ports. The client machines each include two Intel Xeon E5-2630 CPUs operating at 2.3GHz. Their NICs are a mixture of Intel 82599ES and Solarflare SFC9020 10GbE NICs. The server machine that runs IX, ZygOS, or Shinjuku includes two Intel E5-2658 CPUs operating at 2.3GHz, 128GB of DRAM, and an Intel 82599ES 10Gb NIC. All machines run Ubuntu LTS 16.0.4 with the 4.4.0 Linux kernel. Hyperthreading is always enabled unless noted. NICs are configured as half-duplex by the IX and ZygOS drivers and we use the same setting for Shinjuku. To perform scalability experiments, we also use the server machine with a 40Gb Intel XL710-QDA2 NIC and an identical E5-2658 two-socket machine as the client.

Each of the two server CPUs has 12 cores and 24 hyperthreads. However, ZygOS and IX can only support up to 16 hyperthreads as their network drivers are limited to 16 RSS RX queues. Hence, we use an 8-core (16-hyperthread) configuration for most experiments. Shinjuku always uses two of the available hyperthreads for the networking subsystem and dispatcher. Hence, our results use the notation `Shinjuku(x)` to specify that Shinjuku uses `x-2` hyperthreads for workers and a total of `x` hyperthreads. The notation `IX(x)` and `ZygOS(x)` specify that IX and ZygOS use `x` hyperthreads, all for d-FCFS or c-FCFS processing respectively.

Networking: We use the following networking subsystem with Shinjuku. A single hyperthread, co-located on the same physical core with the dispatcher, polls the NIC queue and processes raw packets. It performs UDP processing, identifies requests, and optionally parses the request header to identify types. The Shinjuku workers process network replies. This simple subsystem is sufficient to evaluate Shinjuku. Since Shinjuku decouples network processing from request scheduling, we can combine Shinjuku in the future with alternative systems that implement other transport protocols and use optimizations such as multithreaded stacks [31, 22] or stacks that offload networking to a SmartNIC [33, 18, 53, 19]. The latter will free x86 hyperthreads for Shinjuku workers. If the SmartNIC is connected to the processor chip through a coherent interconnect like Intel's UPI, we can also offload the Shinjuku dispatcher to the NIC cores.

IX supports both UDP and TCP networking. We use it with UDP and a batch size of 64. ZygOS supports only TCP networking [4], but is configured to use exactly one TCP segment per request and reply. Hence, ZygOS requests have some additional service time for TCP processing ($< 0.25\mu sec$), but are otherwise similar to UDP-based IX and Shinjuku requests.

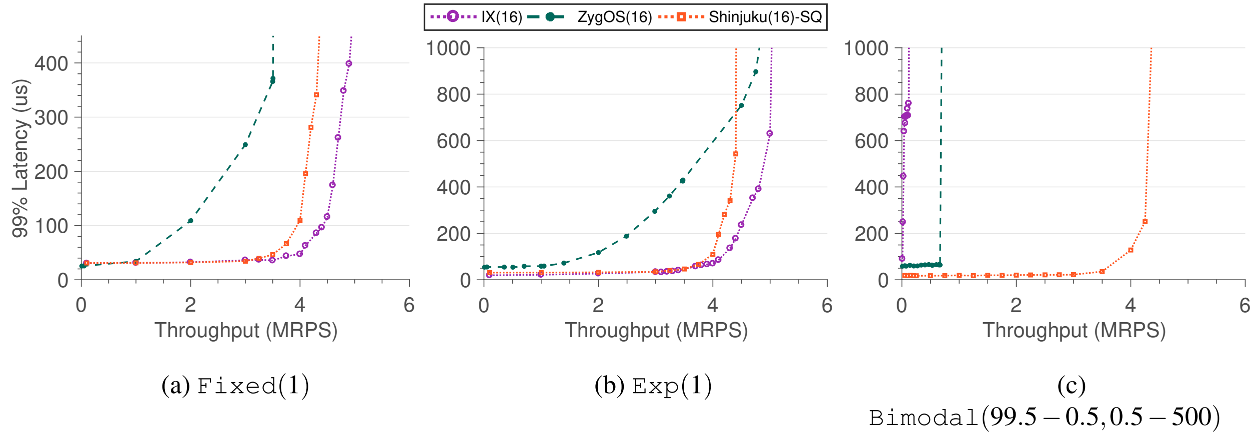
Workloads: We use one synthetic and one real workload. The synthetic workload is a server application where requests perform dummy work that we can control in order to emulate any target distribution of service times. This synthetic server allows us to derive insights about how the three systems compare across a large application space.

We also use RocksDB version 5.13 [26], a popular and widely deployed key-value store developed by Facebook. The IX, ZygOS, and Shinjuku servers handle RocksDB queries that may be simple get/put requests or range scans. We configure RocksDB to keep all data in DRAM in order to evaluate all three systems under the lowest latency requirements possible. If some RocksDB requests had to access data in Flash, the variability of service times would be even higher, and the preemptive Shinjuku would perform even better than the non-preemptive IX and ZygOS.

We developed an **open loop** load generator similar to mutilate [36] that transmits requests over either TCP or UDP. The load generator starts a large number of connections in a set of client machines, while it measures latency from a single unloaded machine. Unless otherwise noted, we use 1920 persistent TCP connections (ZygOS) and 1920 distinct UDP 5-tuples (IX and Shinjuku). Using fewer connections significantly affected the performance of IX and ZygOS (see §3.6).

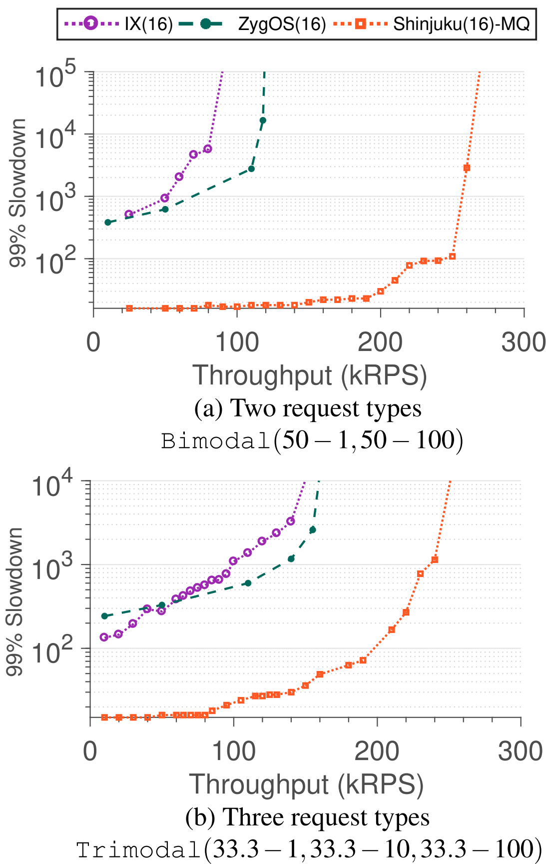
4.2 Synthetic Workload Comparison

Figure 5 compares Shinjuku to IX and ZygOS for three service time distributions. Figure 5a uses a fixed service time of $1\mu sec$, while Figure 5b uses an exponential distribution with a mean of $1\mu sec$. These two cases are ideal for IX that uses d-FCFS. IX benefits further from its ability to batch similarly sized requests. Shinjuku (SQ) performs close to IX, despite exclusively using two hyperthreads for networking and the dispatcher and despite preempting requests that exceed $5\mu sec$. In this case, Shinjuku places preempted requests at the head of the queues. Moreover, preemption is fast and for light-



(a) `Fixed`(1) (b) `Exp`(1) (c) `Bimodal`$(99.5-0.5, 0.5-500)$

Figure 5: Systems comparison with synthetic workloads. Shinjuku uses the single queue policy.



(a) Two request types `Bimodal`$(50-1, 50-100)$

(b) Three request types `Trimodal`$(33.3-1, 33.3-10, 33.3-100)$

Figure 6: Systems comparison with multi-modal synthetic workloads. Shinjuku uses the multi-queue policy.

tailed workloads only a few requests will be preempted allowing Shinjuku to outperform ZygOS for both scenarios. ZygOS also has a high stealing rate (60%) even for homogeneous workloads which exacerbates its stealing overheads. A similar performance drop was also observed in the original ZygOS paper [46].

Figure 5c uses a `Bimodal`$(99.5-0.5, 0.5-500)$ service time distribution where 99.5% of the requests have a $0.5\mu sec$ service time and 0.5% $500\mu sec$. Shinjuku with the single queue policy is vastly better than both IX and ZygOS, achieving up to **50% lower tail latency at low load** and **5x better throughput** for a given 300μs tail latency target. IX and ZygOS lack preemption, hence the 0.5% of long requests determine the overall 99th percentile tail latency as short requests are frequently blocked behind them. The task stealing in ZygOS improves upon IX but is not sufficient to deal with the high dispersion in service times. In contrast, Shinjuku preempts long requests and places them at the tail of the single queue to allow short requests to complete quickly.

Figure 6 evaluates the three systems with multiple request types, a key experiment that was missing from the original IX and ZygOS papers. We use Shinjuku's multi-queue policy which assumes knowledge of the request types (e.g., from packet inspection). Figure 6a uses a `Bimodal`$(50-1, 50-100)$ workload, while Figure 6b uses a `Trimodal`$(33-1, 33-10, 33-100)$ workload. In all cases, Shinjuku places preempted requests to the head of their corresponding queues. Both figures show the 99th percentile of request slowdown (overall request latency / service time) with a logarithmic y-axis. The preemptive, multi-queue policy allows Shinjuku to outperform IX and Zygos by having **94% lower slowdown at low load and over 2x higer throughput (RPS)**. In addition to the frequent preemption that avoids head-of-line blocking, Shinjuku benefits from its ability to select the type of requests (long vs. short) to serve next based on their ratio of queuing time to target latency.

4.3 Shinjuku Analysis

How important is frequent preemption? Figure 7a

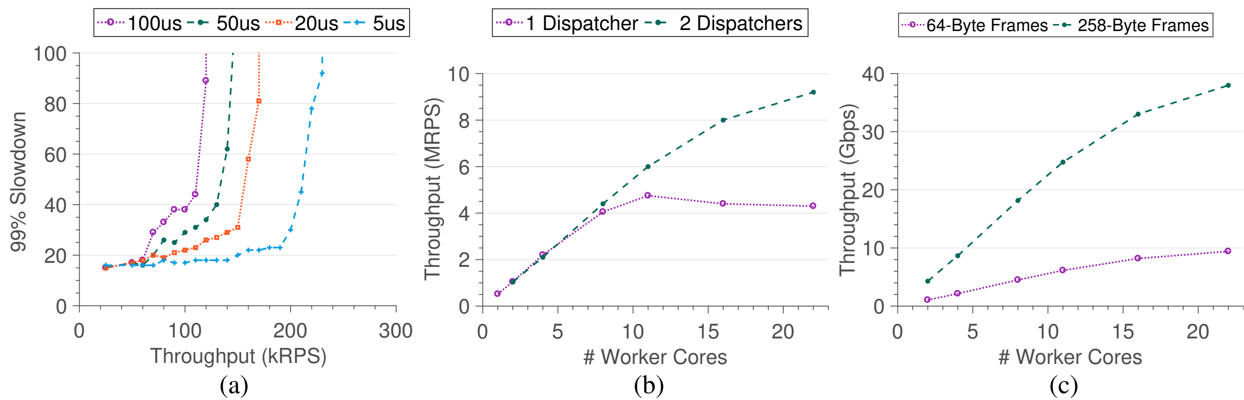
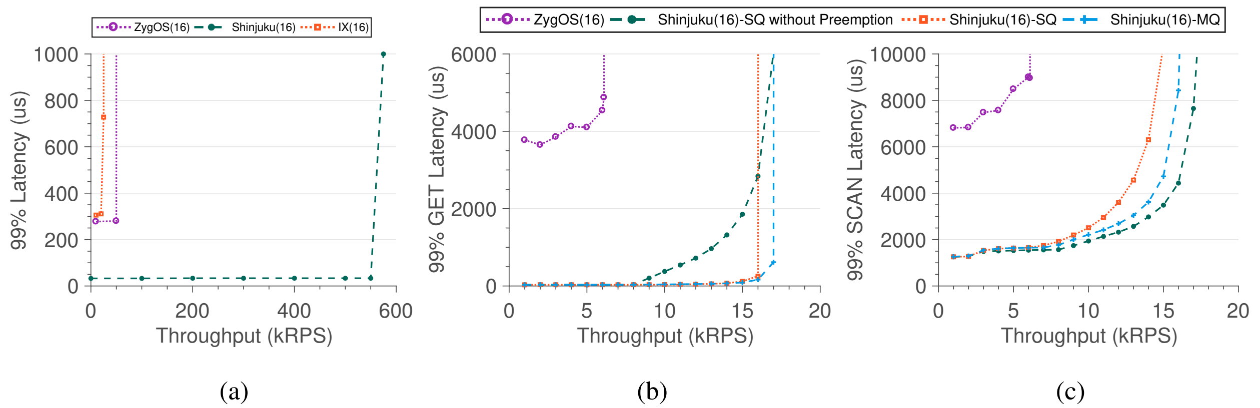


Figure 7: (a) Two request types $\texttt{Bimodal}(50-1, 50-100)$ with varying preemption time slice. (b) Shinjuku throughput (Million RPS) as we scale the worker cores. (c) Shinjuku throughput (Gbps) as we scale the worker cores.



(a) (b) (c)

Figure 8: RocksDB (a) Shinjuku, IX, and ZygOS 99.5% GET 0.5% SCAN(1000). (b + c) Shinjuku Performance - 50% GET(b) 50% SCAN(5000)(c).

varies the preemption interval for a $\texttt{Bimodal}(50-1, 50-100)$ synthetic workload. Shinjuku uses the multi-queue policy. The shorter the preemption interval, the better Shinjuku performs as the impact of $100\mu sec$ requests on $1\mu sec$ is reduced. Shinjuku performs well even at the very frequent $5\mu sec$ preemption interval.

How does Shinjuku scale? Figures 7{b,c} examine how Shinjuku scales with more workers. We issue short requests with 1μsec fixed service time to stress the dispatcher. We also use the Intel XL710-QDA2 40Gb NICs so that networking is not a bottleneck. Since each worker thread can saturate its core, we turn off hyperthreading and pin each worker thread to a physical core. We use the two hyperthreads in the 12th physical core for the dispatcher and the networking threads. Figure 7b shows that a single dispatcher thread scales almost linearly to 11 worker cores, which is the socket size in our server. A second dispatcher thread allows Shinjuku to schedule across the 22 worker cores on both sockets for a single application. Shinjuku can schedule 5M and 9.5M RPS with 1 and 2 dispatchers respectively. Figure 7c measures the outgoing network throughput of Shinjuku using two dispatchers. Shinjuku saturates the 40Gb NIC when reply frames are as short as 258 bytes.

These two figures validate that a single Shinjuku application can scale to high core counts and high line rates even with short 1μsec service times.

4.4 RocksDB Comparison

We use RocksDB with a simple server we ported to IX, ZygOS, and Shinjuku. Client requests are looked up in a RocksDB database created on an in-memory file system (/tmpfs/) with random key-value pairs. We use two request types: GET requests for a single key-value pair that execute within $6\mu sec$; SCAN requests that scan 1,000 or 5,000 key-value pairs and require $240\mu sec$ or $1,200\mu sec$ respectively. We use memory-mapped plain tables as the backing files to avoid memory copies and access to block devices. Shinjuku uses a preemption time slice of $15\mu sec$ and places preempted requests at the head of their corresponding queues for the multi-queue policy and at the tail for the single-queue policy.

Figure 8a compares IX, ZygOS, and Shinjuku with the single queue policy for a 99.5-0.5 mix of GET and

SCAN(1000) requests. Shinjuku provides a vast improvement over ZygOS in tail latency (88% decrease) and throughput (6.6x improvement). Frequent preemption in Shinjuku allows GET requests to avoid long queuing times due to SCAN requests. IX performs even worse due to the combination of highly imbalanced request service times and d-FCFS scheduling.

Preemption and queue selection policy matter: Figures 8b and 8c use a 50-50 workload between GET and SCAN(5000) requests. In addition to comparing with ZygOS, we modified the Shinjuku dispatcher to show the impact of using Shinjuku without preemption, the single-queue preemptive policy, and the multi-queue preemptive policy. IX is omitted because its latency is outside the range of our plot. The results show that Shinjuku without preemption (SQ without preemption) favors the longer SCAN requests over the shorter GET requests. The addition of preemption (SQ) fixes this problem and allows both request types to achieve fair throughput and low tail latency. The multi-queue policy (MQ) improves SCAN requests as it avoids excessive queuing for them as well. ZygOS performs significantly worse even than Shinjuku without preemption. ZygOS uses distributed queuing and is susceptible to head-of-line blocking for requests within the same connection. This supports our decision to decouple network processing and request scheduling in Shinjuku.

5 Related Work

Optimized network stacks: There is significant work in optimizing network stacks, including polling based processing (DPDK [3]), multi-core scalability (mTCP [31]), modularity and specialization (Sandstorm [40]), and OS bypass (Andromeda [22]). Shinjuku is orthogonal to this work as it optimizes request scheduling after network protocol processing.

Dataplane operating systems: Several recent systems optimize for throughput and tail latency by separating the OS dataplane from the OS control plane, an idea originating in Exokernel [25]. IX [16], Arrakis [45], MICA [39], Chronos [32], and ZygOS [46] fall in this category. Shinjuku improves on these systems by introducing preemptive scheduling that allows short requests to avoid excessive queuing.

Task scheduling: Li *et al.* [38] control tail latency by reducing the amount of resources dedicated to long-running requests that violate the SLO. Haque *et al.* [29] take the opposite approach and devote more resources to stragglers so that they finish faster. Interestingly, both approaches work well. However, these approaches are applicable to millisecond-scale workloads and require workloads that are dynamically parallelizable. Shinjuku allows the development of efficient scheduling policies for requests 3 orders of magnitude shorter than what this line of work can handle.

Flow scheduling: PIAS [12] is a network flow scheduling mechanism that uses hardware priority queues available in switches to approximate the Shortest Job First (SJF) scheduling policy and prioritize short flows over longs ones. We do not follow a similar approach in Shinjuku as SJF is optimal in terms of minimizing average but not tail latency [57]. Moreover, in order to be effective, PIAS requires some form of congestion control to keep the queue length short. This is not practical in non-networked settings where the runtime does not control the application.

Exit-less interrupts: The idea of safe, low-overhead interrupts was introduced in ELI for fast delivery of interrupts to VMs [10]. ZygOS [46] uses inter-processor interrupts for work stealing but does not implement preemptive scheduling. Shinjuku uses Dune [15] to optimize processor-to-processor interrupts.

User-space thread management: Starting with scheduler activations [11], there have been several efforts to implement efficient, user-space thread libraries [8, 56, 48, 1]. They all focus on cooperative scheduling. Shinjuku shows that preemptive scheduling is practical at microsecond-scales and leads to low tail latency and high throughput.

6 Conclusion

Shinjuku uses hardware support for virtualization to make frequent preemption practical at the microsecond scale. Hence, its scheduling policies can avoid the common pitfall of non-preemptive policies where short requests are blocked behind long requests. Shinjuku provides low tail latency and high throughput for a wide range of distributions and request service times regardless of the number of client connections. For the RocksDB KVS, we show that Shinjuku improves upon the recently published ZygOS system by 6.6x in throughput and 88% in tail latency.

Acknowledgements

We thank our shepherd, Irene Zhang, and the anonymous NSDI reviewers for their helpful feedback. We also thank John Ousterhout, Adam Wierman, and Ana Klimovic for providing feedback on early versions of this paper. This work was supported by the Stanford Platform Lab and by gifts from Google, Huawei, and Samsung.

References

[1] libfiber: A user space threading library supporting multi-core systems. https://github.com/brianwatling/libfiber, 2015.

[2] ebpf - extended berkeley packet filter. http://prototype-kernel.readthedocs.io/en/latest/bpf/ , 2016.

[3] Data plan development kit. http://www.dpdk.org/, 2018.

[4] Ix-project: Protected dataplane for low latency and high performance. https://github.com/ix-project/ , 2018.

[5] libuv: Cross-platform asynchronous i/o. https://libuv.org/, 2018.

[6] Memcached. https://memcached.org/ , 2018.

[7] Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, Boston, MA, 2019. USENIX Association.

[8] Atul Adya, Jon Howell, Marvin Theimer, William J. Bolosky, and John R. Douceur. Cooperative task management without manual stack management. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference*, ATC '02, pages 289–302. USENIX Association, 2002.

[9] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, pages 63–74, New Delhi, India, 2010. ACM.

[10] Nadav Amit, Abel Gordon, Nadav Har'El, Muli Ben-Yehuda, Alex Landau, Assaf Schuster, and Dan Tsafrir. Bare-metal performance for virtual machines with exitless interrupts. *Commun. ACM*, 59(1):108–116, December 2015.

[11] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, SOSP '91, pages 95–109, Pacific Grove, California, USA, 1991. ACM.

[12] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Hao Wang. Information-agnostic flow scheduling for commodity data centers. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 455–468, Oakland, CA, 2015. USENIX Association.

[13] L. A. Barroso, J. Dean, and U. Holzle. Web search for a planet: The google cluster architecture. *IEEE Micro*, 23(2):22–28, March 2003.

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

[15] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe user-level access to privileged cpu features. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 335–348, Hollywood, CA, USA, 2012. USENIX Association.

[16] 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 *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 49–65, Broomfield, CO, 2014. USENIX Association.

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

[18] Broadcom. Ps225. https://www.broadcom.com/products/ethernet-connectivity/network-adapters/ps225 , 2017.

[19] Cavium. Liquidio smartnic. https://www.cavium.com/product-liquidio-adapters.html , 2018.

[20] Intel Corp. Intel virtualization technology. https://www.intel.com/content/www/us/en/virtualization/virtualization-technology/intel-virtualization-technology.html , 2018.

[21] Microsoft Corp. Receive side scaling. http://msdn.microsoft.com/library/windows/hardware/ff556942.aspx , 2018.

[22] Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Arefin, Anshuman Gupta, Brian Fahs, Dima Rubinstein, Enrique Cauich Zermeno, Erik Rubow, James Alexander Docauer, Jesse Alpert, Jing Ai, Jon Olson, Kevin DeCabooter, Marc de Kruijf, Nan Hua, Nathan Lewis, Nikhil Kasinadhuni, Riccardo Crepaldi, Srinivas Krishnan, Subbaiah Venkata, Yossi Richter, Uday Naik, and Amin Vahdat. Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 373–387, Renton, WA, 2018. USENIX Association.

[23] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56:74–80, 2013.

[24] Kenneth J. Duda and David R. Cheriton. Borrowed-virtual-time (bvt) scheduling: Supporting latency-sensitive threads in a general-purpose scheduler. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, SOSP '99, pages 261–276, Charleston, South Carolina, USA, 1999. ACM.

[25] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 251–266, Copper Mountain, Colorado, USA, 1995. ACM.

[26] Facebook. Rocksdb. http://rocksdb.org/ , 2018.

[27] Yi Guo, Jisheng Zhao, Vincent Cave, and Vivek Sarkar. Slaw: A scalable locality-aware adaptive work-stealing scheduler for multi-core systems. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 341–342, Bangalore, India, 2010. ACM.

[28] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. Softnic: A software nic to augment hardware. Technical Report UCB/EECS-2015-155, EECS Department, University of California, Berkeley, May 2015.

[29] Md E. Haque, Yong hun Eom, Yuxiong He, Sameh Elnikety, Ricardo Bianchini, and Kathryn S. McKinley. Few-to-many: Incremental parallelism for reducing tail latency in interactive services. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 161–175, Istanbul, Turkey, 2015. ACM.

[30] Md E. Haque, Yuxiong He, Sameh Elnikety, Thu D. Nguyen, Ricardo Bianchini, and Kathryn S. McKinley. Exploiting heterogeneity for tail latency and energy efficiency. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, pages 625–638, Cambridge, Massachusetts, 2017. ACM.

[31] 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, Seattle, WA, 2014. USENIX Association.

[32] 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*, SoCC '12, pages 9:1–9:14, San Jose, California, 2012. ACM.

[33] Antoine Kaufmann, SImon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurthy. High performance packet processing with flexnic. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 67–81, Atlanta, Georgia, USA, 2016. ACM.

[34] Sanjeev Kumar, Christopher J. Hughes, and Anthony Nguyen. Carbon: Architectural support for fine-grained parallelism on chip multiprocessors. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 162–173, San Diego, California, USA, 2007. ACM.

[35] Redis Labs. Redis. `https://redis.io/`, 2018.

[36] Jacob Leverich and Christos Kozyrakis. Reconciling high server utilization and sub-millisecond quality-of-service. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 4:1–4:14, Amsterdam, The Netherlands, 2014. ACM.

[37] 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, pages 9:1–9:14, Seattle, WA, USA, 2014. ACM.

[38] Jing Li, Kunal Agrawal, Sameh Elnikety, Yuxiong He, I-Ting Angelina Lee, Chenyang Lu, and Kathryn S. McKinley. Work stealing for interactive services to meet target latency. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, pages 14:1–14:13, Barcelona, Spain, 2016. ACM.

[39] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. Mica: A holistic approach to fast in-memory key-value storage. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 429–444, Seattle, WA, 2014. USENIX Association.

[40] Ilias Marinos, Robert N.M. Watson, and Mark Handley. Network stack specialization for performance. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 175–186, Chicago, Illinois, USA, 2014. ACM.

[41] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. Timely: Rtt-based congestion control for the datacenter. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 537–550, London, United Kingdom, 2015. ACM.

[42] Jayakrishnan Nair, Adam Wierman, and Bert Zwart. The fundamentals of heavy-tails: Properties, emergence, and identification. In *Proceedings of the ACM SIGMETRICS/International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '13, pages 387–388, Pittsburgh, PA, USA, 2013. ACM.

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

[44] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen Rumble, Ryan Stutsman, and Stephen Yang. The ramcloud storage system. *ACM Trans. Comput. Syst.*, 33(3):7:1–7:55, August 2015.

[45] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 1–16, Broomfield, CO, 2014. USENIX Association.

[46] 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*, SOSP '17, pages 325–341, Shanghai, China, 2017. ACM.

[47] N. Provos and N. Mathewson. libevent: An event notification library. `http://libevent.org`, 2018.

[48] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. Arachne: Core-aware thread management. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 145–160, Carlsbad, CA, 2018. USENIX Association.

[49] Gang Ren, Eric Tune, Tipp Moseley, Yixin Shi, Silvius Rus, and Robert Hundt. Google-wide profiling: A continuous profiling infrastructure for data centers. *IEEE Micro*, pages 65–79, 2010.

[50] Sepideh Roghanchi, Jakob Eriksson, and Nilanjana Basu. Ffwd: Delegation is (much) faster than you think. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 342–358, Shanghai, China, 2017. ACM.

[51] Daniel Sanchez, Richard M. Yoo, and Christos Kozyrakis. Flexible architectural support for fine-grain scheduling. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 311–322, Pittsburgh, Pennsylvania, USA, 2010. ACM.

[52] Mellanox Technologies. Rdma over converged ethernet. `http://www.mellanox.com/related-docs/whitepapers/roce_in_the_data_center.pdf`, 2014.

[53] Mellanox Technologies. Bluefield multicore system on chip. `http://www.mellanox.com/related-docs/npu-multicore-processors/PB_Bluefield_SoC.pdf`, 2017.

[54] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 18–32, Farminton, Pennsylvania, 2013. ACM.

[55] Kaushik Veeraraghavan, Justin Meza, David Chou, Wonho Kim, Sonia Margulis, Scott Michelson, Rajesh Nishtala, Daniel Obenshain, Dmitri Perelman, and Yee Jiun Song. Kraken: Leveraging live traffic tests to identify and resolve resource utilization bottlenecks in large scale web services. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 635–651, Savannah, GA, 2016. USENIX Association.

[56] Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer. Capriccio: Scalable threads for internet services. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 268–281, Bolton Landing, NY, USA, 2003. ACM.

[57] Adam Wierman and Bert Zwart. Is tail-optimal scheduling possible? *Oper. Res.*, 60(5):1249–1257, September 2012.