Making Kernel Bypass Practical for the Cloud with Junction

Joshua Fried and Gohar Irfan Chaudhry, *MIT CSAIL;* Enrique Saurez, Esha Choukse, and Íñigo Goiri, *Azure Research – Systems;* Sameh Elnikety, *Microsoft Research;* Rodrigo Fonseca, *Azure Research – Systems;* Adam Belay, *MIT CSAIL*

https://www.usenix.org/conference/nsdi24/presentation/fried

This paper is included in the
Proceedings of the 21st USENIX Symposium on
Networked Systems Design and Implementation.

April 16–18, 2024 • Santa Clara, CA, USA

978-1-939133-39-7

Open access to the Proceedings of the
21st USENIX Symposium on Networked
Systems Design and Implementation
is sponsored by

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

Making Kernel Bypass Practical for the Cloud with Junction

Joshua Fried, Gohar Irfan Chaudhry, Enrique Saurez[†], Esha Choukse[†], Íñigo Goiri[†],
Sameh Elnikety[‡], Rodrigo Fonseca[†], Adam Belay
MIT CSAIL [†]*Azure Research – Systems* [‡]*Microsoft Research*

Abstract. Kernel bypass systems have demonstrated order of magnitude improvements in throughput and tail latency for network-intensive applications relative to traditional operating systems (OSes). To achieve such excellent performance, however, they rely on dedicated resources (*e.g.,* spinning cores, pinned memory) and require application rewriting. This is unattractive to cloud operators because they aim to densely pack applications, and rewriting cloud software requires a massive investment of valuable developer time. For both reasons, kernel bypass, as it exists, is impractical for the cloud.

In this paper, we show these compromises are not necessary to unlock the full benefits of kernel bypass. We present Junction, the first kernel bypass system that can pack thousands of instances on a machine while providing compatibility with unmodified Linux applications. Junction achieves high density through several advanced NIC features that reduce pinned memory and the overhead of monitoring large numbers of queues. It maintains compatibility with minimal overhead through optimizations that exploit a shared address space with the application. Junction scales to 19–62× more instances than existing kernel bypass systems and can achieve similar or better performance without code changes. Furthermore, Junction delivers significant performance benefits to applications previously unsupported by kernel bypass, including those that depend on runtime systems like Go, Java, Node, and Python. In a comparison to native Linux, Junction increases throughput by 1.6–7.0× while using 1.2–3.8× less cores across seven applications.

1 Introduction

Network-intensive applications have experienced remarkable performance improvements (*i.e.,* order of magnitude better tail latency and throughput) from kernel bypass systems [6, 15, 28, 29, 37, 43, 44, 67]. Their key idea is to map network queues into userspace, so applications can communicate directly with the NIC and avoid kernel overheads.

Junction is a new kernel bypass system that targets cloud applications (*e.g.,* microservices, serverless, etc.). Like previous kernel bypass systems, Junction delivers significant performance improvements, including higher throughput and greater CPU efficiency, as well as order of magnitude reductions in tail latency relative to traditional OSes. At the same time, Junction is the first kernel bypass system that retains compatibility with unmodified Linux binaries and is capable of achieving high density (*i.e.,* the ability to scale to thousands of instances on a machine). Junction delivers these benefits while maintaining strict isolation between applications with a narrower attack surface than existing cloud isolation schemes.

Prior kernel bypass systems make compromises that render them impractical for use in the cloud. For example, they require dedicated, busy-spinning cores and pinned memory, so very few instances can be packed on a machine. Moreover, they make significant changes to the programming model [6, 14, 21, 33, 52] that break compatibility and sacrifice the enormous investment made in existing software. Finally, most kernel bypass systems provide no isolation of their own, so they must be combined with virtualization—and its associated overheads (*e.g.,* VM exit costs, nested page tables, etc.)—to be deployed safely in a cloud setting.

Junction is able to retain the full performance benefits of kernel bypass without these compromises through a set of design contributions that target isolation, density, and compatibility. To achieve strong isolation while avoiding virtualization overheads, Junction runs each instance inside a normal Linux process and installs a filter that limits access to system calls. Within an instance, Junction runs as a library that shares an address space with the application. Because Junction is able to build all of its OS abstractions on top of kernel bypass hardware (*e.g.,* NIC queues, CPU features, etc.) it requires only minimal interactions with the kernel, just enough to enable resource multiplexing (≈ a dozen system calls).

To achieve high density, Junction efficiently multiplexes both cores and memory. For cores, Junction builds upon prior work on a dedicated scheduler core, but overcomes the previously unaddressed challenge of scaling to a large number of instances. To do so, Junction makes novel use of a NIC hardware feature that delivers packet arrival notifications on a dedicated queue instead of requiring each receive queue to be polled individually. For memory, Junction employs a variety of new techniques to reduce the footprint of packet buffers, including configuring NIC hardware to share a queue of receive buffers across multiple cores, as well as allowing multiple packets to be posted in each receive buffer. Junction also securely exposes the Linux page cache to share read-only memory across instances.

To achieve Linux compatibility, Junction provides its own implementation of the Linux Kernel system call interface. This was challenging to do in a way that maintains the performance benefits of kernel bypass. Junction exploits the fact that it runs in the same address space as the application to unlock optimizations that minimize the cost of compatibility. For example, Junction safely converts system calls into function calls, avoids transient execution mitigations,

System	Density			Compatibility				
	Mem. overhead (single core)	Mem. overhead per instance per core	Max instances (128GB RAM)	Ported Application	Lines of code			Compatible w/ existing clients
					Added	Removed	Modified	
eRPC [30]	24 MB	24 MB	# of cores	Masstree	551	0	0	✗
Demikernel [67]	167 MB	–	# of cores	Redis	926	819	213	✓
Caladan [15]	648 MB	9 MB	200	Memcached	393	539	637	✓
Junction	29 MB	0.47 MB	3,500	*No porting*	–	–	–	✓

Table 1: Existing kernel bypass systems require large amounts of dedicated host resources and invasive changes to applications.

accesses arguments directly without copying them, exploits undefined behavior to eliminate locking, and uses vector instructions without the need to save and restore register state. Junction also provides many OS features that are missing in existing kernel bypass systems (*e.g.,* signals, thread-local storage, randomness, file systems, timers, etc.) but are crucial to supporting cloud applications. It maintains a kernel bypass approach to delivering these features by exploiting modern CPU extensions to avoid traps into the kernel.

Results. Junction helps to pave a path toward practical deployment of kernel bypass in the cloud by showing it is possible to deliver high performance without sacrificing security, density, or compatibility. For example, Junction can run unmodified Linux binaries while matching or exceeding the performance of three state-of-the-art kernel bypass systems that require significant code changes. Moreover, Junction's reliance on kernel bypass hardware allows it to reduce the number of system calls in its attack surface by 69%–87% relative to two security-focused library OSes, and Junction's buffer management and queue polling optimizations allow it to pack thousands of instances on a machine. Finally, Junction is the first system to bring the performance benefits of kernel bypass to unmodified applications with complex language runtimes (*e.g.,* Python, Node, Go, Java, etc.). It improves throughput by 1.6–7.0× and reduces CPU use by 1.2–3.8× for seven applications relative to Linux. Junction is available as open source software: `https://github.com/JunctionOS/junction`.

2 Background & Motivation

Kernel bypass systems eliminate the kernel from the network datapath, and replace it with an optimized user-level networking stack that communicates directly with the NIC. In this section, we first discuss why the existing approach to kernel bypass has shortcomings that hinder adoption, especially in a cloud setting where density and compatibility are crucial issues. Next, we discuss current progress in making kernel bypass more general purpose. Finally, we discuss how a lack of security further compounds these issues.

Density challenges. Cloud providers commonly pack many instances on a machine to improve density [16, 55, 62, 68]. This is necessary because latency-sensitive applications have variability in their demand for resources, and utilization can only be kept high by filling in idle resources with best-effort applications. Furthermore, in serverless environments, it is typical for thousands of instances to remain active on a single machine to prevent cold start delays [1].

Unfortunately, kernel bypass systems today can only support a limited number of instances on a machine (left side of Table 1). One major problem is the widespread use of busy spinning and dedicated cores [30, 67]. Because this approach requires a minimum of one core per instance (and often many more), the maximum number of instances is limited by the number of cores.

New approaches to CPU scheduling that speed up core allocation can overcome this limitation and eliminate waste from busy spinning without sacrificing tail latency [15,42,50]. However, these systems rely on a dedicated core to make scheduling decisions, so their scalability is still limited. For example, Caladan is unable to scale beyond a few hundred instances because of bottlenecks in its scheduler core.

Finally, the memory footprint of kernel bypass systems is also a significant barrier to achieving high density. This is especially true for kernel bypass systems that handle TCP connections, like the Demikernel and Caladan, because they cannot guarantee the application will consume packet buffers the moment they arrive (*e.g.,* what if they arrive out of order?), so they must reserve a significant number of buffers to avoid packet drops. There is also an additional per-core cost for packet buffers in multicore kernel bypass systems because each core must post enough receive buffers to handle a worst case burst in traffic, which could be uneven across cores.

Compatibility challenges. Ideally, a kernel bypass system should support unmodified binaries. Existing kernel bypass systems instead require applications to be ported by developers (right side of Table 1). This is a significant barrier to adoption. For example, at the time of writing, three recent, state-of-the-art kernel bypass systems (eRPC, Demikernel, and Caladan) support just a handful of applications. Moreover, all of these applications are key-value stores written in C or C++, which are less challenging to port than typical cloud applications. A recent survey of serverless functions found that Node and Python were among the most popular languages [57], and both require a full language runtime with a complex set of OS dependencies that cannot be met by existing kernel bypass systems.

Breaking compatibility is not easy in the cloud because of

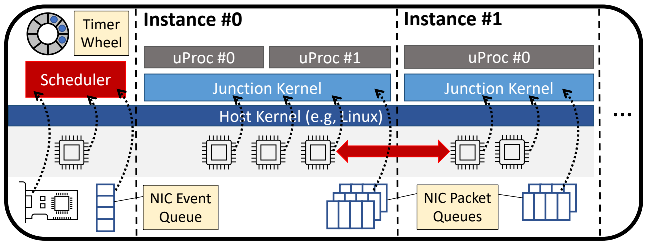


Figure 1: Junction's system architecture.

the significant investment developers have made in existing software. One obvious solution would be to focus on rewriting just the most popular applications. Unfortunately, this would have little impact on efficiency overall because, even together, these applications do not account for a large enough fraction of overall resource usage [31].

Making kernel bypass general purpose. There have been several recent efforts to make kernel bypass more general purpose. For example, normally kernel bypass systems use run-to-completion to optimize for short requests [6], but this leaves them vulnerable to high tail latency when there is dispersion in request service times. Shinjuku [29] and Perséphone [9] solve this through efficient, fine-grained preemption and steering of short requests to separate cores respectively. Many kernel bypass systems also use "share nothing" designs that harm tail latency under load imbalance, an issue that ZygOS addresses through work stealing [47]. Demikernel provides unifying abstractions on top of different hardware backends (*e.g.,* RDMA vs. Ethernet) to reduce developer effort [67]. Finally, Arachne [50] and Shenango [42] show that threading can be made fast enough to be used with kernel bypass networking. Junction adopts several ideas from these systems while solving the previously unaddressed challenges of density and compatibility.

Security challenges. Most kernel bypass systems eschew isolation and must be run as root. As a result, they depend on other isolation mechanisms to be deployed safely in the cloud. The most viable option is to run each instance in a separate VM, but this increases overheads including extra TLB misses, VM exit costs, and a larger memory footprint caused by the guest kernel. In most cases, VMs also cannot take advantage of the page cache, which further limits their density.

3 Junction Overview

Figure 1 provides an overview of Junction and highlights its main components. Junction is designed to handle thousands of instances on a machine. An *instance* is an isolated container that runs one or more application binaries. From the host kernel's perspective, this container consists of a single process (called a kProc) with a fixed set of threads (called kThreads) that are statically initialized at startup time. The kThreads are scheduled on cores by a centralized scheduler (left side of Figure 1). An instance can load and run multiple binaries in its shared address space (placing each at different offsets in virtual memory). Each binary within an instance runs in a userspace process abstraction called a uProc.

A copy of the *Junction kernel* runs inside each instance and shares an address space with its uProcs. It directly handles system calls from uProcs and provides OS abstractions (*e.g.,* threading, networking, filesystems, signals, etc.) in userspace, similar to a library OS [10, 46]. The Junction kernel supports the Linux system call interface so that it can run existing software without modifications.

The Junction kernel uses kernel bypass hardware (both networking queues and CPU features) to provides its OS abstractions. As a result, most system calls can be handled entirely in userspace and it is only necessary to make system calls into the host kernel to multiplex resources (*i.e.,* cores and memory)—all other host kernel system calls can be blocked. Shifting OS functionality into userspace improves performance by reducing the frequency of boundary crossings, and limits the attack surface by allowing untrusted programs to exercise only very small parts of trusted host kernel code.

Throughout this paper, we assume Linux is used as the host kernel, but any standard OS could serve this purpose. Junction can coexist with normal Linux processes that do not use Junction. As a result, Junction can take full advantage of existing debugging and profiling tools; and the control plane and management functions of a standard Linux environment.

Networking and communication. Like other kernel bypass systems, Junction instances are provisioned with one NIC send and receive queue pair per kThread. This improves performance by allowing concurrent access to the NIC without synchronization. The Junction kernel provides a high-performance TCP/IP and UDP networking stack, which enable uProcs to communicate with the outside world. uProcs within the same instance can communicate with each other using standard inter-process communication (IPC) primitives (*e.g.,* pipes), but different instances on the same host may only communicate via loopback networking through the NIC.

Threading. The Junction kernel includes a high-performance, user-level threading library that uses work stealing to balance light weight user-level threads (uThreads) across kThreads. uProc threads (*i.e.,* those created when starting or via `clone3()`) are mapped to uThreads. uThreads are also used for various internal tasks like network protocol processing. Each kThread runs a scheduling loop that polls local queues for packets and timeouts, and runs pending uThreads.

Core scheduling. Junction relies on a microkernel-style scheduler to make core allocation decisions [15, 23, 42, 48]. It runs on a dedicated core and busy polls control signals to decide how and when cores should be allocated to each instance (indicated by the red double arrow in Figure 1). Instances can use as few as zero cores when idle, or more than one if demand justifies it (up to a per-instance limit). For each instance, the scheduler monitors timer expirations and queueing delays in both thread and network queues, which are made visible to the scheduler through shared memory. When the scheduler grants a core to an instance, it selects one of its idle kThreads and pins it to the core for the duration of the grant.

In addition to performing core allocation, the scheduler assists the threading library in implementing fine-grained timeslicing by sending user IPIs (UIPIs) [25] to cores where a running uThread has exceeded its timeslice quantum. This ensures that all uThreads can make progress, and that packet queues are drained in a timely manner. It is also beneficial for reducing tail latency when service time dispersion is high [29]. As an optimization, these interrupts are only sent when queued packets or runnable threads are waiting to be processed.

Because Junction aims to support thousands of instances, it employs novel techniques to ensure that control signals can be monitored in a scalable way. A *timer wheel* keeps track of the next timeout for each instance and a *NIC event queue* provides notifications of packets arrivals. These reduce how often the scheduler has to poll shared state inside each instance to determine if it could benefit from additional cores. We discuss these optimizations in more detail in §5.

4 Security

The conventional wisdom is to use virtual machines as the isolation boundary for the cloud to reduce interactions between untrusted code and the host kernel. However, virtualization still exercises a large amount of trusted code in the host kernel, resulting in a significant attack surface [3, 65]. Junction, by contrast, delivers OS abstractions directly on top of kernel bypass hardware, significantly reducing its reliance on the host kernel. In this section, we discuss Junction's security design in more detail.

4.1 Threat Model

We assume each instance can run arbitrary and potentially malicious code, so strong memory isolation is needed between instances and the host kernel. We are particularly concerned about a malicious user exploiting a bug in the host kernel through a system call or VM exit. Today's host kernels have wide attacks surfaces that can exercise millions of lines of code, and transient execution attacks potentially expose additional vulnerabilities in this code [4]. This poses a large security risk in cloud deployments [1, 17].

The Junction kernel, by contrast, runs as a separate copy in each instance. Each copy shares an address space and has fate sharing with the uProcs it handles, but is strictly isolated from other instances. Therefore, if a uProc corrupts its memory or exploits a bug in a system call implementation, it can only harm its own availability. Through the same reasoning, validating system call arguments or preventing transient executions attacks is not required for isolation with the Junction kernel. This unlocks many opportunities for optimization that are not available to the host kernel (§6.2). uProcs that are mutually trusting can run in the same instance at lower overhead, or in separate instances if isolation is needed.

Category	Syscalls
CPU scheduling	yield_core()
Memory management	mmap(), munmap(), madvise(), mprotect(), mremap()
Loading binaries	open(), close(), pread64()
Logging (optional)	write()
Process management	exit_group()

Table 2: System calls that Junction requires from the host kernel.

The NIC is trusted and we assume it can provide network virtualization and packet scheduling across multiple instances. NICs with these capabilities are commonly deployed in public clouds today (*e.g.,* Microsoft Catapult [12] and Amazon Nitro [2]).

4.2 Host Kernel Isolation

Isolation mechanisms. Junction relies on kProcs, which are normal Linux processes, for isolation of its instances. Each kProc installs a strict seccomp filter that limits access to a restrictive set of system calls. Because Junction uses the page cache to improve density, an instance is given read-only access to a `chroot` jail directory with needed binaries and shared libraries. Junction instances also have direct access to the NIC through a set of queues and a dedicated page for MMIO doorbell writes to PCIe. The NIC is aware of distinct Junction instances, and provides each instance with its own queues, pinned memory, and doorbell page.

Allowed system calls. Junction requires only 11 system calls, as shown in Table 2. Two of these are used only rarely: `write()` is used for debug logging to `stdout`, and `exit_group()` is used only once when the instance exits. Similar to Caladan [15], Junction uses a single blocking system call, `yield_core()`, to yield and allocate cores. This call is provided by a custom Linux Kernel module, and coordinates with the scheduler—informing it when a kThread yields voluntarily. The module allows the scheduler to wake a kThread on a specific core and unblock it from its `yield_core()` call.

Junction requires five system calls to manage memory. This is important for density because instances rarely need all of the memory they reserve [1]. First, Junction relies on `mmap()` to allocate anonymous memory and to map files. Second, it relies on `mprotect()` and `munmap()` to change mapping permissions and remove mappings respectively. Third, `mremap()` is needed to adjust existing mappings. Finally, `madvise()` is exposed to provide hints to the Linux Kernel, such as releasing unused memory without modifying the VMA, which has lower overhead than `munmap()`.

Page cache. To minimize the attack surface, we initially considered a design where Junction did not access the Linux Kernel file system at all, instead delivering files over the network (*e.g.,* 9P) or in memory. However, we realized that to achieve density, limited access to the Linux file system is necessary. This is because the host kernel is the only layer that can mediate access to the page cache, allowing different processes to share read-only mappings to the same disk blocks. Reducing the memory footprint in this way allows Junction to achieve higher density (§8.6).

Our goal, therefore, is to expose just enough of the file system to make it possible to leverage the page cache. Using the `chroot` jail, an instance can use `open()` to access allowed files in read-only mode and `close()` when done. `mmap()` is used to map the files into memory (*i.e.,* ELF segments), enabling use of the page cache. `pread64()`, which takes an offset and is designed to scale well across cores, is also made available to allow the contents of files to be read before they are mapped into memory. Before adopting its syscall filter, an instance scans the files and folders in its jail and caches the metadata (*e.g.,* file size), so it does not need access to any further file-system calls when running.

5 Optimizing for Density

Junction's goal is to deliver high networking throughput and low latency, like existing kernel bypass systems, while also packing significantly more instances on a machine. This required us to resolve several problems related to the use of a large number of NIC receive queues.

Normally, kernel bypass systems assign a separate receive queue to each core in order to avoid synchronization overheads so they can scale linearly with the number of cores. Although Junction differs in that it adjusts cores dynamically based on load, it still must assign enough receive queues to each instance so that one will be available for each kThread that might be running. As a result, the number of queues needed is the maximum number of cores per instance times the number of instances.

Modern NICs can easily scale to thousands of queues, but using them to pack many instances on a machine still poses significant challenges. First, buffer memory consumption is a key limit for density because each receive queue must post enough buffers to accommodate a worst-case burst in arriving packets, which is exacerbated by the unevenness in traffic across cores. Second, the cost of polling every queue in the core scheduler becomes prohibitive, with cache pollution causing a performance collapse. We now discuss our solution to each of these problems in more detail.

5.1 Minimizing Buffer Memory Consumption

Existing kernel bypass systems maintain large pools of memory for buffers that are used to send and receive packets through the NIC. These buffers must be available to the NIC for direct memory access, so the backing memory must be *pinned* to prevent the host kernel from swapping it out. Pinning buffers at the time of sending/receive packets is prohibitively expensive, so it is instead better to maintain a pool of pinned buffers. Sizing this pool requires consideration of several factors, including the number of open connections (*i.e.,* the total outstanding send and receive windows for a protocol like TCP), the worst-case delay in packet processing time, and the network round trip time. To handle variability in processing time and to absorb bursts, it is common to keep receive rings full with buffers so that packets are not dropped.

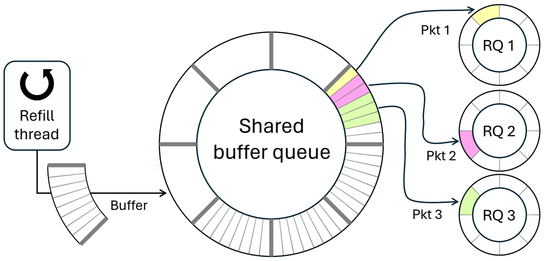


Figure 2: Junction reduces memory use through (A) a shared buffer queue that supplies per-core receive queues (RQs), and (B) packing many packets in each buffer (even when delivered to different RQs). A refill thread replenishes buffers.

Kernel bypass systems often provision pinned memory well in excess of the minimum amount needed. One major factor that contributes to this is the use of per-core receive queues. Traffic is distributed among receive queues using RSS, which is susceptible to skew and can lead to bursts on some queues. Each queue must have enough posted buffers to ensure that it can handle any traffic distributed to it. For example, kernel bypass network stacks can drive a single TCP flow at speeds greater than 100Gb/s. Because such a flow would be hashed to one receive queue at random, each queue must have enough buffers available to handle this high rate of traffic, despite the fact that only one queue's buffers will be consumed. Thus the amount of buffer memory needs to scale linearly with the number of cores an instance may be using.

A second factor that contributes to wasted memory is the need to support variably-sized packets. Normally, each packet that arrives consumes a single posted buffer, which must be at least MTU-sized. Small packets consume only a fraction of these buffers, so to ensure that enough bytes are available to buffer small packets that arrive at high rates, the buffer pool must be made proportionally larger.

As shown in Figure 2, Junction takes advantage of recent NIC hardware to overcome both of these problems. First, it uses a per-instance shared buffer queue to post buffers instead of posting them separately in each receive queue. This allows posted buffers to be shared amongst the receive queues, eliminating the need to scale the buffer pool with the number of cores. Second, it avoids fragmentation in large packet buffers by allowing many packets to be placed consecutively in each buffer, reducing the minimum memory consumed per-packet from MTU size (between 1500B to 9000B) to 256B.

Using a shared buffer queue requires coordination across cores, which can add overhead and limit scalability. The host cannot refill a slot in the buffer queue until the NIC has finished writing to it, which is detected by tracking arrivals at each receive queue. Additionally, because many packets can be stored in a single buffer, a buffer cannot be reused until all of its packets are freed. This is especially challenging because packets from a single buffer can be spread across many receive queues, requiring coordination across cores when freeing packets and refilling the shared buffer queue.

Prior work on ShRing [45] proposed hardware modifications to the NIC to enable efficient coordination. However, Junction reduces synchronization overhead with a software-only approach by using per-core reference counters for each slot in the shared buffer queue and each buffer in the buffer pool (*i.e.,* no cachelines are shared). Because buffer sizes can be large (we picked 16KB), the shared buffer queue can be kept small, resulting in low per-core memory overhead (about 512B). A high-priority *refill thread* is responsible for managing the shared buffer queue: it scans the counters and replenishes queue slots with free buffers. We provide more details on our approach and how it compares to ShRing in Appendix B.

5.2 Scalable Queue Polling

Using a spinning core for rapid core scheduling eliminates the need for each instance to spin-poll its own receive queues, allowing idle cores to be reallocated to applications that need them (§2). The scheduler polls shared memory locations in a loop to monitor queueing delays across each instance's network receive queues, thread runqueues, and timers. With large numbers of instances, the polling loop can take a long time to complete, leading to delays in wakeups and increases in latency. Furthermore, polling many locations pollutes the cache for the scheduler, leading to additional slowdowns. Junction avoids polling thread runqueues for inactive applications, but must still track pending timers and arriving packets as both may warrant an immediate core allocation for an idle instance.

Junction makes two main modifications to the scheduler core model to dramatically improve its scalability by reducing the amount of memory that the scheduler must inspect in each scheduling pass. First, it uses a set of NIC features in a novel way to avoid continuously polling idle network queues. Junction allocates a single *event queue* and a dedicated doorbell page for the scheduler core. Each time the scheduler observes an empty receive queue, it *arms* the queue by marking the index of the current head pointer and writing to the doorbell. When a packet arrives on an armed queue, the NIC writes an event into the event queue and disarms the queue. The scheduler continually polls the event queue and can react immediately when a packet arrives at an idle queue. This feature is available on modern Mellanox NICs.

Expiring timers may also require cores to be allocated to idle instances. High resolution timers are important for datacenter workloads; for example, they may used to detect when TCP segments need to be retransmitted, or an RPC has failed. In order to ensure that instances with timers are woken with minimal delay, Junction's scheduler employs a second optimization: a high resolution (16 μs) hierarchical timer wheel [61]. The timer wheel allows the scheduler to ignore instances with timers that expire far out in the future and monitor only instances that are either active or have immediately pending timeouts.

Subsystem	Hardware Feature	Syscall Alternative
Networking	NIC Queues	socket(),recv(),send()
Randomness	`RDRAND`, `RDSEED`	getrandom()
Threading (TLS)	`WRFSBASE`	arch_prctl()
Signals	`SENDUIPI`, `XSAVEC`	tgkill()
	`XRSTOR`, `UIRET`	rt_sigreturn()

Table 3: Kernel bypass hardware features used by Junction.

We also optimize the memory footprint of the data structures used by Junction's scheduler to ensure that the state from each instance occupies as few cache lines as possible. Additionally, state is arranged to avoid false sharing, *i.e.,* state needed only when an application is active is kept separate from state needed when an instance is idle. Together, these optimizations overcome the scalability bottlenecks of a centralized core scheduler and enable Junction's scheduler to manage thousands of active instances without latency issues.

6 Linux Compatibility

In line with experiences reported by other researchers, we found that it was tractable for a small team to implement enough of the Linux interface to run a wide range of applications [20, 22, 46]. Constraining our goal to only supporting cloud applications made this easier. For example, we could ignore desktop features like graphics, input devices, and sound, which would have required significantly more developer effort. In addition, most cloud applications are built for specific runtime systems and do not perform system calls directly. Therefore, targeting all the system calls needed by a particular runtime can enable a broad swath of compatibility. For example, we found Junction could run any Go program after implementing the set of system calls needed by its runtime.

We had to overcome two challenges to achieve Linux compatibility. First, we had to provide OS features that are not available in prior kernel bypass systems, but are necessary for cloud applications. Second, we had to minimize the overhead of compatibility in order to not squander the performance benefits of kernel bypass. We address the first challenge by shifting OS abstractions into userspace and by building them on top of kernel bypass hardware (Table 3). We address the second challenge by exploiting fate sharing and a shared address space to unlock performance optimizations. We discuss each solution in more detail next.

6.1 Adapting OS Features to Kernel Bypass

Loader and multiprocess support. The Junction kernel includes its own ELF loader to load uProcs inside an instance. The ELF loader is invoked automatically at startup to load the first uProc—a path to the program image is provided as a configuration parameter. The ELF loader can also be invoked later by the `execve()` system call, which enables loading of additional uProcs. During ELF loading, Junction creates a new uThread and populates its stack with environment variables, arguments, and an auxiliary vector. The auxiliary vector contains several important parameters that are needed to emulate a Linux process environment [35].

In an earlier prototype of Junction, the Junction kernel was instead deployed as a library that was linked with the application, similar to a Unikernel [24, 34]. We decided against this approach, however, because it required compiling a new binary for each program. This breaks compatibility with existing Linux binaries and also makes it more difficult to support multiple processes within an instance.

Multiprocess support is an important feature for some cloud applications (*e.g.,* microservices) because they rely on sidecars for RPC handling, logging, and other services. Sidecars are normally trusted by the application, so it is acceptable to run them together, potentially without memory isolation [56]. In Linux, a new process is created via `fork()`, which spawns a separate address space. However, Junction is a single address space OS, so it cannot fork to create new uProcs.

To launch multiple uProcs, Junction instead relies on `vfork()`, an optimized version of `fork()` that delays the creation of a new address space until `execve()` is called. Linux uses this to fuse the fork operation with the loading of the program, eliminating the need to clone the page table. However, Junction transparently co-opts the `vfork()` + `execve()` sequence to provide a different behavior. Instead of creating a new address space, it finds an empty location in the existing address space and loads the program there, allowing it to support multiple uProcs in one instance. To avoid collisions, a program must work correctly in any location in the address space, so at most one uProc can be compiled without position-independent code (PIC) enabled. Fortunately, for security reasons, PIC is enabled by default in most datacenters.

Threading. User-level threading packages are gaining traction because of their increased performance (*e.g.,* Java's Project Loom [49] and the Go Runtime [39]). Junction brings the same benefits to unmodified binaries by shifting all threading operations (*e.g.,* creating threads, acquiring mutexes, context switching, etc.) into userspace instead of going through the host kernel. Junction maintains a separate uThread runqueue in each kThread and uses packet arrivals, timeouts, signals, and other events to wake uThreads.

Junction provides threading support at two layers of abstraction. First, it supports the low-level Linux system calls that are required for threading (*e.g.,* `futex()`, `clone()`, etc.). This is needed to achieve compatibility with programs that use nonstandard threading libraries. Second, for greater efficiency, it overrides glibc's pthread library, and provides a custom implementation that is integrated directly with Junction and does not make use of these system calls. For example, `futex()` must normally do a hash table lookup to find if a thread is blocking on an address, but Junction can instead reference the mutex object directly to find the blocking uThread. Another challenge not addressed in prior kernel bypass systems is support for thread-local storage (TLS). Junction's solution is to rely on the `WRFSBASE` instruction to switch between thread-local regions during each uThread context switch. Without this instruction, performance would be significantly worse as the host kernel's `arch_prctl()` system call would have to be invoked at each context switch (§8.6).

Signals. Surprisingly, we found that many cloud applications depend on signals for normal operation. For example, Go uses signals to preempt and reschedule Go Routines, and the Hotspot JVM uses signals as an optimization to avoid explicit NULL pointer checks. This reflects two separate forms of signals supported by Linux: 1) those that are sent internally by an application (*e.g.,* via `tgkill()`) and 2) those that are generated by CPU exceptions (*e.g.,* page faults).

The difficulty in supporting signals is that their behavior is highly customizable (*e.g.,* setting handlers, masking signals, using alternate stacks, etc.). However, exposing `sigaction()` and `sigaltstack()` through the host kernel would significantly widen the attack surface. Signals also compose poorly with Junction's threading layer, because signals must normally wake or preempt a specific uThread, but the host kernel is only aware of kThreads.

Instead, Junction uses UIPIs [25] to reduce the involvement of the host kernel. When a uThread invokes `tgkill()` to send an internal signal, the Junction kernel uses the `SENDUIPI` instruction to send a UIPI (if preemption is necessary). We discuss our implementation with UIPIs further in Appendix A.

CPU exceptions, however, still requires involvement from the host kernel because the CPU does not support user-level handling of these faults. Junction statically configures the host kernel with handlers for every possible CPU exception signal (using an alternate signal stack) before dropping privileges. When a uThread triggers a CPU exception, the host kernel sets up a trapframe on an alternate stack and invokes Junction's signal handler. Thankfully, UIPI's `UIRET` instruction obviates the need for the host kernel's `rt_sigreturn()` system call since it can atomically restore stack and instruction pointers. Though a system call is typically needed to unblock signals that are masked during delivery, Junction configures its Linux signal handlers to never alter the signal mask. This is appropriate because CPU exceptions cannot be masked.

When an interrupt or Linux signal is delivered, Junction uses its internal knowledge of uProcs and their signal configuration to route the signal to the right handler, which pushes the signal's trap frame onto a uThread's stack. This potentially includes waking a uThread or preempting a running uThread.

Other OS features. Junction provides support for networking sockets, IPC (*e.g.,* `pipe()` and loopback networking), waiting for events (*e.g.,* `select()`, `poll()`, `epoll()`, and `eventfd()`), a virtual filesystem, memory management (*e.g.,* `mmap()`), time keeping, and many more OS features not found in prior kernel bypass systems. We adopted several novel strategies to reduce overheads, discussed next in more detail.

6.2 Performance Optimizations

System call handling. The main mechanism for intercepting system calls in Linux is `seccomp`, which can generate a signal for each system call it intercepts. Junction uses this mechanism as an occasional fallback, but we found that it had a prohibitive level of overhead for intercepting all system calls. A more efficient alternative would be to patch occurrences of the `SYSCALL` instruction so they jump directly into the Junction kernel. Researchers recently discovered a clever trick to support this without exceeding the instruction length of `SYSCALL`, eliminating the need for binary rewriting [66].

However, Junction uses a different strategy to squeeze out even more performance. When a program is loaded, Junction's ELF loader transparently replaces glibc with a modified version. We found that nearly all system calls are performed through glibc so this was a good place to intercept them. The modified library is designed to call into Junction during each system call. This includes the use of a trampoline page that finds the location of Junction, which is randomized for ASLR.

One key benefit to this approach is that it allows each system call to be invoked like a normal function call with standard calling conventions. This is favorable for performance because some general purpose registers and all floating point and vector registers can be safely clobbered. As a result, unlike the Linux Kernel, the Junction kernel can be compiled with all optimizations enabled, including those that use vector instructions. This also means that significantly less register state has to be saved and restored when context switching between threads that are blocked. However, preempted threads still require all state to be restored. Because of fate sharing, there is also no need to apply transient execution attack mitigations during system calls [4]. Instead, these are only necessary when entering the host kernel.

Reducing compatibility overheads. Junction further reduces overheads by changing the way system calls are implemented. First, Junction does not need to mitigate time-of-check to time-of-use (TOCTOU) attacks, again because of fate sharing. As a result, Junction does not copy system call arguments. Second, the UNIX standard has many examples of undefined behavior. In the Linux Kernel these have to be implemented carefully to avoid compromising security, but in Junction they can be implemented in whatever way achieves the best performance.

A good example of this opportunity is that if a file descriptor is closed while `select()` is monitoring it, the behavior is undefined. The Linux Kernel still requires extra locking to prevent race conditions. On the other hand, Junction allows these race conditions to happen and avoids the cost of locking because it assumes a correctly written program will never trigger undefined behavior. However, Junction must still perform all standard argument checking (*e.g.,* is a file descriptor valid) because Linux programs depend on these behaviors.

7 Implementation

Junction is implemented in about 12,000 lines of C++23 code and runs on modern x86 CPUs. It is linked against Caladan's runtime (14,000 LOC), which it uses as a low-level library for networking and threading routines. Caladan provided useful support for centralized core scheduling. However, we had to heavily modify it to scale to more instances and to run under our restricted host kernel interface. Moreover, we completely replaced its `mlx5` driver (for modern Mellanox NICs) to enable shared buffer queues and multi-packet receive buffers (5,000 LOC); and we modified its kernel module to support UIPIs (500 LOC). The modified NIC driver replaces Linux's ibverbs and tightly integrates with the scheduler, allowing it to expose the NIC event queue.

Linux compatibility. Junction's current implementation supports 126 Linux system calls. We found this subset sufficient to run language frameworks including Python, Go, Node.js, and Java, as well as a variety of applications written in C, C++, and Rust. With the exception of Go, Junction runs all of these as unmodified Linux binaries. Go programs belong to a rare class of applications that make most of their system calls outside of libc (the only one we encountered). To deliver the best performance, we added a new OS target to the Go compiler that uses function calls and our trampoline instead of system calls. However, `seccomp` can still handle system calls even for unmodified Go binaries.

8 Evaluation

Our evaluation aims to answer the following questions:

1. How does Junction's performance compare to state-of-the-art kernel bypass systems (§8.2)?
2. How many active instances can Junction pack on a machine (§8.3)?
3. Can Junction achieve compatibility with unmodified cloud applications (§8.4)?
4. Can Junction be securely deployed in a multi-tenant cloud (§8.5)?
5. What factors contribute to Junction's better performance and higher density (§8.6)?

8.1 Methodology

Experimental setup. Most experiments are run on a server with an Intel Xeon 6354 3.6 GHz 18-core CPU, 64GB of RAM, and a 200GbE Mellanox ConnectX-6 NIC. Scaling

	Files	Net.	Mem.	Process	Threads/ Sync.	Signals	Rand.	Timers	Multi- threaded	Garbage Collected	Kernel Time (%)
Memcached (C)	6	20	4	12	4	2	1	1	✓	✗	86.4
Redis (C)	11	14	5	9	5	3	1	1	✗	✗	84.2
Masstree (C++)	4	14	5	6	4	4	0	1	✓	✗	83.2
nginx (C)	5	9	4	10	2	0	1	1	✗	✗	76.1
Node.js HTTP	7	12	6	14	4	2	1	0	✗	✓	44.7
Python HTTP	11	10	4	11	3	1	1	0	✗	✗	63.5
Go HTTP	4	9	4	6	4	3	1	1	✓	✓	55.3
Rocket (Rust)	7	13	5	7	4	0	1	0	✓	✗	44.7
Tomcat (Java)	15	18	5	15	5	4	1	2	✓	✓	51.6

Table 4: Characteristics of evaluated applications, including unique system calls used grouped by kernel subsystem, whether an application is multithreaded or uses garbage collection, and ratio of time spent in the kernel (when running in Linux).

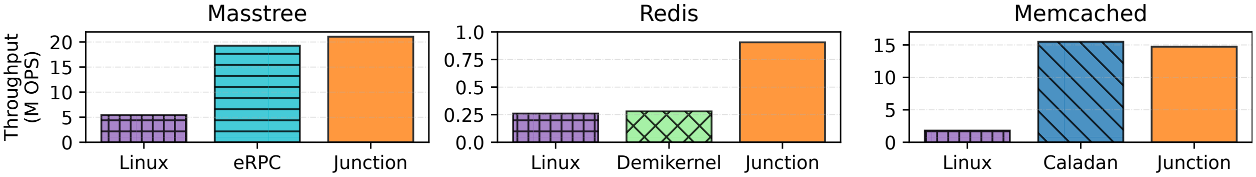


Figure 3: Performance comparison of Junction with state-of-the-art kernel bypass systems. Junction offers higher throughput than eRPC and Demikernel and is competitive with Caladan despite requiring no code modifications.

experiments run on a server with an Intel Xeon 5420+ 2.7GHz 28-core CPU and 128GB of RAM. Unless otherwise noted, we use a set of client machines connected to a 100GbE switch to generate load. The server runs Linux Kernel version 6.2, with the default mitigations enabled for CPU vulnerabilities. We use an open-loop kernel bypass load generator with Poisson arrivals [42] for latency measurements.

Systems. We compare Junction to three state-of-the-art kernel bypass systems: eRPC [30], Demikernel [67], and Caladan [15]. We also compare Junction to two state-of-the-art cloud isolation systems: Firecracker [1], a micro-VM isolation system, and gVisor [17], a secure container isolation system that also implements the Linux system call interface in userspace. Finally, we show performance relative to native Linux. We made every effort to tune each system for its maximum performance, and ensured that performance matched what is reported in other studies.

Applications. Table 4 characterizes the applications that we use throughout the evaluation, which include a range of web servers and various in-memory database and key-value stores.

8.2 Comparison to other kernel bypass systems

Figure 3 shows that Junction delivers performance to unmodified binaries that is on-par with or better than existing kernel bypass systems that require modifications to applications. The set of applications that have been ported to existing kernel bypass systems is limited and disjoint, making it difficult to use the same application to evaluate each system. Instead, we select one of the ported applications that was used originally to evaluate each kernel bypass system and compare to an unmodified Linux binary running in Junction and Linux. Table 1 shows the porting effort required for each of them. Except for Demikernel, which doesn't support multiple cores, we configure each system to use up to 16 cores.

eRPC. We compare against eRPC using the in-memory database called Masstree [36]. eRPC's port of Masstree replaces its included TCP/UDP server with its own wrapper around the database and a custom wire format. We compare the performance of eRPC running with RoCE versus Junction and Linux using standard TCP. We provision 128 client threads across 8 machines to issue a mixture of 50% GET and 50% PUT requests to the server (with four outstanding requests per thread). Because of eRPC's custom wire protocol, we use eRPC's client to measure eRPC's performance, and the Linux binary running in Junction to measure Linux and Junction. Junction achieves higher throughput (over 21 MOp/s) than eRPC (19.3 MOp/s), both significant improvements over the Linux baseline. We observe that eRPC spends up to 10% of its CPU time handling futexes while allocating memory, an overhead that Junction largely eliminates.

Demikernel. We benchmark Demikernel using Redis and its included redis-benchmark utility. We run the redis-benchmark client inside Junction to measure throughput of GET and SET requests over TCP connections. The Linux baseline achieves 260,000 RPS while consuming 1.5 CPU cores (this includes softirq time). Demikernel improves throughput by about 7% while using only 1 CPU core. Junction improves throughput by 248% over Linux, while also using only 1 CPU core. Junction's TCP stack is more efficient (*e.g.,* it uses a fast-path [32]), and we suspect this is the main contributor to its performance advantage relative to Demikernel in this setting.

Caladan. We use memcached [13] to compare performance to Caladan [15], with additional detail on tail latency in Fig-

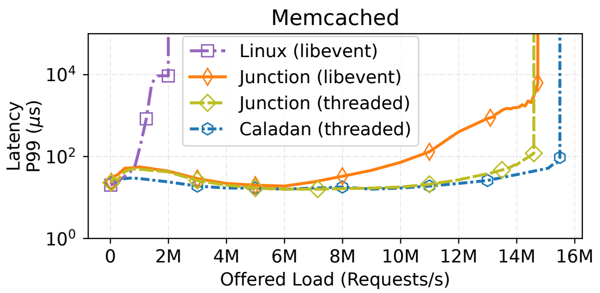


Figure 4: Latency for native memcached binaries (libevent) running in Linux and Junction, as well as two ported versions that replace libevent with per-connection threads (threaded) written against POSIX and Caladan interfaces. Junction can nearly match the throughput of Caladan in both cases, but the added overheads of libevent harm latency at higher throughputs.

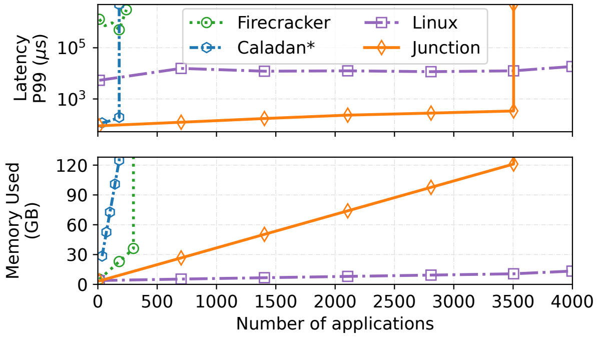


Figure 5: Performance and resource consumption while varying the number of single-threaded Rocket web servers packed into one machine. Load is fixed at 150K RPS and spread evenly across all instances. Junction supports up to 3,500 active instances while using 3.33× less memory per instance than Firecracker and keeping tail latency 35× lower than Linux.

ure 4. This comparison highlights the performance costs of using unmodified binaries because the two systems share a user threading library and TCP stack. We found that the while the native Linux binary in Junction could nearly match the throughput of Caladan, its tail latency degraded more quickly. This is because the Caladan version of memcached entirely removes `libevent` and `epoll()`, which were used to multiplex large numbers of connections across a smaller set of threads. We performed the same set of modifications to the Linux binary and evaluated its performance on Junction. The modified version (threaded) shows that Junction can match the latency profile of Caladan, suggesting that limiting the number of threads is important for OSes with high kernel crossing overheads, but can be detrimental otherwise due to head-of-line blocking. Thus, an optimal version of memcached for Junction would merely spawn a thread for each connection. Nevertheless, this result shows that Junction can offer dramatic improvement for low-latency applications even without modifications.

8.3 Density

We now demonstrate that Junction can densely pack many active instances on a machine and deliver low tail latency. Figure 5 shows an experiment where we provision increasing numbers of instances of a multi-threaded HTTP server written in Rust called Rocket. We use a host with 128GB of RAM. Each instance is provisioned with 8 threads. We offer a total load of 150,000 RPS to the machine, evenly divided across all instances. At each instance count, we show the aggregate tail latency experienced across all instances and the total memory consumption. We compare Junction to a Linux baseline and to Firecracker (designed to have a low per-instance memory footprint while providing an isolated VM environment). We also show *Caladan**, which mixes Junction with Caladan's network stack that is not optimized for density; this scales to only 180 instances. Linux consumes hardly any memory per instance (<1MB) and can scale far beyond Junction, but has poor tail latency. Firecracker's memory overhead is 8.6× lower than Caladan's but suffers from even poorer tail latency. Junction scales to 3,500 instances before running out of memory, with p99 latency below 350 μs, a 35× improvement over Linux.

8.4 Compatibility

We demonstrate that Junction achieves broad compatibility for cloud workloads by benchmarking a suite of HTTP servers written in several popular languages/frameworks. This collection includes (1) nginx [51], a popular load-balancer and proxy written in C, (2) Node.js's built-in web server [40], (3) a simple web server written in Python [60], (4) Go's built-in HTTP package [39], (5) the Rocket framework in Rust [53], and (6) the Apache Tomcat framework in Java [58]. These applications cannot be supported by any existing kernel bypass system without a large porting effort. We compare them to native Linux, and to both gVisor and Firecracker. We configure each application to use up to 8 cores, though several are single-threaded. All are configured to deliver small static responses (14–600 bytes); nginx is the only one configured to read its response from a file (stored in a RAM-backed file system). We use 200 concurrent connections with connection keep-alives to avoid the cost of additional TCP handshakes.

Figure 6 shows the p99 latency and total CPU utilization across varying load for each application. For all applications, Junction provides superior throughput, latency, and CPU efficiency. Relative to Linux, Junction improves throughput by 1.62-3.69× and uses 19-65% less CPU when handling Linux's peak loads. Junction shows even larger gains against Firecracker and gVisor.

8.5 Attack Surface

Junction's use of kernel bypass reduces the host kernel attack surface relative to existing security-focused library

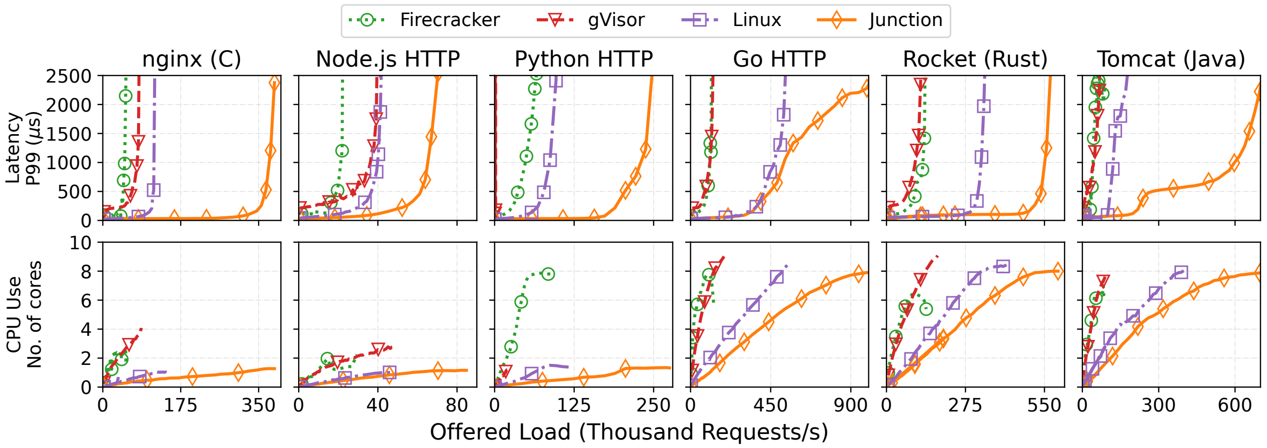


Figure 6: Service response-time and corresponding CPU usage at varying offered loads. Junction offers higher throughput and lower tail latency while also using fewer cores compared to other systems.

	Total	Mem.	Threads/ Procs	Sync	I/O	Misc.
Junction	11	5	2	0	4	0
gVisor (Sentry) [19]	64	7	11	2	34	10
gVisor (Gofer) [18]	57	4	11	1	36	5
Drawbridge	36	3	7	7	12	7

Table 5: Comparison of allowed syscalls to the host kernel for Junction and two other library operating systems.

	Unique syscalls	Total syscalls (/s)	`VMEXIT`s (/s)
Junction	4	9,603	n/a
Firecracker [1]	5	68,162	102,027
gVisor [17]	21	99,879	13,084
Linux	14	34,087	n/a

Table 6: Kernel crossings (per second) when running Rocket for 10s with an offered load of 10,000 RPS.

OSes. To demonstrate this, Table 5 shows the total number of syscalls required to run Junction, compared to gVisor and Drawbridge [46]. Drawbridge is a library OS that provides applications with a Windows 7 interface (in total over 100,000 API calls) using 36 system calls. gVisor is broken down into its two components, the Sentry and Gofer; the Sentry implements much of the system call interface but proxies access to files through the Gofer to add defense in depth. Because Junction uses kernel bypass NIC and CPU features to implement OS functionality, it requires 3.2–7.6× fewer syscalls to run.

Table 6 further demonstrates Junction's lack of reliance on the host kernel by showing profiling output from `strace` while running an 8-threaded Rocket instance at 10K RPS (after initialization). We report host kernel interactions through both system calls and VMExits. 99% of Junction's syscalls are to `yield_core()`, which is called when the application idles between requests, enabling other applications to run. The remaining 1% of calls are exclusively allocating and releasing memory. Linux, gVisor, and Firecracker all rely heavily on system calls that read from, send to, and block on file descriptors. Firecracker and gVisor interact with a Linux TAP device and virtio queues, while the Linux instance interacts with sockets. Both gVisor and Firecracker use `ioctl()` to interact with KVM. gVisor also heavily uses futexes and timers.

8.6 Performance Analysis

To better understand Junction's performance, we evaluated the impact of several of our design choices and mechanisms.

Performance optimizations. Figure 7 demonstrates how various aspects of Junction's design contribute to its performance. Both (1) using the Linux TAP driver to send and receive packets (as gVisor and Firecracker do) and (2) using seccomp filters to trap and intercept syscalls severely limit throughput for Junction instances. Per-core kernel bypass queues allow it to scale to significantly higher rates. Additional optimizations to enable compatibility with unmodified binaries further improve performance: both the use of hardware instructions to support TLS (`WRFSBASE`) and the optimization to allow it to clobber floating point state during system calls. These techniques allow Junction to nearly match the performance of Caladan (*i.e.,* the same TCP stack without compatibility).

Pinned memory. We quantify the improvements from adopting shared buffer queues and multi-packet buffers on the memory footprint of a Junction instance in Figure 8a, which shows the amount of memory consumed by 8-threaded instances with buffer pools sized to accommodate peak throughput. Enabling shared buffer queues shrinks the footprint by 35%, and enabling multi-packet buffers shrinks the footprint by an additional 48%, yielding a 33MB footprint per instance.

Scaling core allocation. We evaluate the two techniques discussed in §5.2 to scale the core allocator in Figure 8b. This experiment uses the same scenario as Figure 5. Enabling notifications from the NIC to a centralized queue allows the scheduler to scale to handle an additional 4× the number of

Page Cache Sharing	Total CPU Utilization
None	65%
+ Sharing Junction Kernel	60%
+ Sharing common libraries	59%
+ Sharing Rocket binary	36%

Table 7: Impact on CPU consumption of varying page cache sharing strategies for Rocket in Junction with 1500 instances.

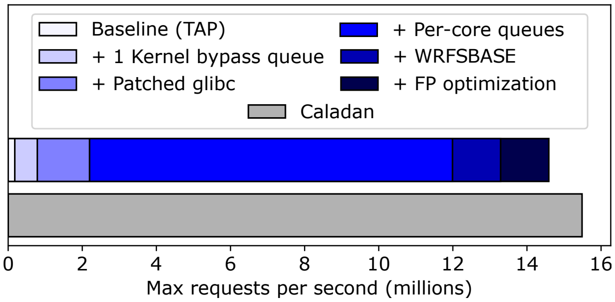
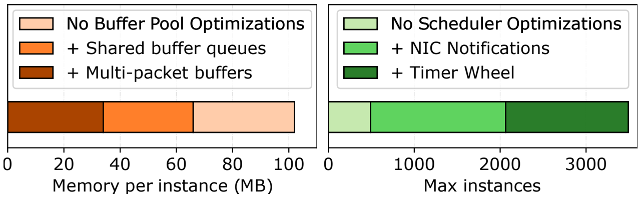


Figure 7: Contributions of several elements of Junction's design to its performance (using memcached).



(a) Memory optimizations.

(b) Scheduler scalability.

Figure 8: Contributions of techniques to improve density (allowing more instances to be packed on a machine).

instances; the timer wheel scales it by an additional 1.69×.

Impact of page cache sharing We investigate the impact of Junction's page cache sharing (unsupported by VMs) in Table 7 by varying which binaries and dynamically linked libraries can be deduplicated. This experiment shows that in extreme case where a large number of identical binaries are operating at once, allowing page cache sharing results in a 44% decrease in CPU consumption. This is because data in instruction caches can be shared across instances, which suggests that operators that favor statically-linked binaries may be squandering performance under dense deployments.

9 Discussion

Hardware features. The CPU features Junction relies on are available on all modern Intel and AMD x86 CPUs. The only exception is UIPIs, which were recently introduced in Intel's Sapphire Rapid CPUs, so their availability is more limited. On older hardware, Junction falls back to the host kernel (`tgkill()` and `rt_sigreturn()`) for signals. Junction also relies on features in recent Mellanox NICs (ConnectX-5 and later) to reduce buffer memory consumption and make polling scalable, but can still support other NICs with potentially higher memory use and less scalability.

In the future, Junction could benefit from additional hardware support. For example, Junction could take advantage of the ability to handle CPU exceptions in userspace without involving the host kernel, as discussed in §6.1 and below.

Host kernel platform. Building Junction on Linux has many advantages. However, better security might be achievable by adopting a clean-slate approach, creating a purpose-built kernel for Junction's restricted host kernel interface. Such a kernel could be extremely minimal and even formally verified, reducing the chances of an exploit. uKVM provides a *monitor* similar to this idea for the Solo5 library OS [64], and we plan to investigate this for Junction in future work.

Another interesting tradeoff that such a platform could allow us to investigate is whether Junction should use Intel's VT-x and EPT extensions. An advantage could be that it would enable Junction to manipulate its page table and handle CPU exceptions directly, further reducing its reliance on the host kernel. Indeed, Dune demonstrates that these extensions can be configured in a way that maintains a process-like environment (*i.e.,* not a full VM) [5]. However, this would incur extra costs when entering and exiting the host kernel.

Performance isolation. Providing performance isolation across tenants is a challenging and important goal for cloud providers. Tenants can contend for CPU time, memory, network bandwidth, and microarchitectural CPU resources (caches, memory bandwidth, etc.). Junction is in a good position to handle this because it builds on Caladan, which already controls multiple forms of microarchitectural CPU interference [15]. While not in our current prototype, Junction's centralized core scheduler can easily implement priority scheduling and enforce CPU shares, and Linux cgroups can be used to enforce memory allocation limits. Finally, network bandwidth allocation can be offloaded to the NIC (§4).

10 Conclusion

This paper presented Junction, a system that retains the performance benefits of kernel bypass while scaling to thousands of instances and maintaining compatibility with existing Linux applications. Junction uses kernel bypass hardware (NIC queues and CPU features) to reduce its reliance on the host kernel, and it restricts the host kernel interface to enable efficient resource multiplexing with a minimal attack surface. It exploits a variety of advanced NIC features and scheduler optimizations to enable higher density. It also adapts OS subsystems to a kernel bypass setting and reduces system call overheads to maintain compatibility without sacrificing performance. Our evaluation shows that Junction can bring large benefits to existing applications without modifications, delivering superior tail latency, throughput, CPU efficiency, and density relative to state-of-the-art kernel bypass and cloud isolation systems.

Acknowledgements

We thank our shepherd Kostis Kaffes, the anonymous reviewers, Frans Kaashoek, Robert Morris, and other members of the MIT PDOS group for their helpful feedback. This work was funded in part by a Facebook Research Award; a Google Faculty Award; the DARPA FastNICs program under contract #HR0011-20-C-0089; the NSF under award CNS-2104398 and CNS-2212099; and VMware.

A UIPI Support

Junction uses UIPIs to support preemptive core allocation, timeslicing, and realtime POSIX signals between uThreads in the same instance. UIPIs are a recent Intel CPU feature that allow IPIs to be directly sent and received in userspace. At the time of writing, UIPI support is not yet a part of the mainline Linux Kernel, but Intel has proposed a patchset to add support for it [38]. Junction's design (*e.g.,* the assumption that one kThread runs on a core at a time) unlocks simplifications to host kernel support for UIPI that diverge from this patchset. Therefore, we implemented UIPI support from scratch as a small Linux Kernel module (§7) that is integrated with the central core scheduler.

Hardware interface. UIPIs build upon Intel's existing posted interrupt hardware, which was used previously to deliver interrupts directly into VMs without involving the host kernel. UIPIs instead allow IPIs to be sent between *normal OS threads* (kThreads in Junction) without going through the host kernel. Each interrupt receiver (*i.e.,* a thread or vCPU) is associated with a posted interrupt descriptor in memory, which serves as a link between the interrupt sender and receiver. When the receiver is scheduled on a core, the host kernel configures the core with several pieces of information: (1) the address of the receiver's descriptor, (2) a physical interrupt vector to use for posted interrupts, and (3) the handler (or guest interrupt vector) that should be invoked when a posted interrupt is received. At the same time, the host kernel writes information about the core into the descriptor (*i.e.,* its APIC ID and the physical interrupt vector). This allows a sender to send interrupts by only referencing a receiver's descriptor, which can be inspected by the CPU at send time to find up-to-date information about which core is running the receiver.

The ability to send UIPIs is restricted by target tables managed by the host kernel, which enumerate the addresses of descriptors that a thread is allowed to send to. To send a UIPI, a program invokes the `SENDUIPI` instruction with an operand that indexes this table. When this instruction is invoked, hardware writes a pending interrupt flag to the target descriptor. It then inspects the descriptor to determine whether or not to send an IPI—if another interrupt is pending or the receiver is not running, it skips sending an IPI. Otherwise, it uses the information in the descriptor to deliver an IPI to the receiver's core. When a core receives an interrupt on its posted interrupt vector, it inspects its current descriptor for pending interrupts and delivers them to the receiver.

UIPI Driver. UIPI support for Junction is managed by its UIPI kernel module. The kernel module receives notifications from the core scheduling kernel module whenever a core is reallocated, so it can appropriately program the core for the next running kThread. Junction does not need to send IPIs to inactive Junction kthreads, so we opt to use a single interrupt descriptor for each core, instead of using one for each kThread. Each Junction instance is provisioned with its own target table for `SENDUIPI` with one entry for each core's descriptor. Rows that correspond to cores actively allocated to this Junction instance are marked as valid, and all other rows are invalidated. The core scheduler's target table gives it permission to interrupt any core on the machine so it can use `SENDUIPI` when it needs to send an IPI. To correctly implement support for UIPIs, the driver must interpose on all context switches and system call entry and exit points for Junction kthreads. It does so by registering itself for callbacks through the Linux Kernel's lightweight tracepoint system.

How Junction uses UIPIs. Both the scheduler and the Junction kernel can send UIPIs to *kick* cores that are running kThreads and force them into the Junction kernel's handler. The handler determines the reason for the IPI and performs the appropriate action (*e.g.,* rescheduling uThreads, scanning network queues, or yielding the core).

Junction's core scheduler uses IPIs to notify kThreads when their core is being revoked or when a uThread running on the core has exceeded its timeslice. This works well when the scheduler needs to send just a single IPI at a time. However, because the scheduler implements microsecond-scale scheduling, it often needs to send multiple IPIs. While the interrupt controller supports IPI multicasting, a feature that reduces the cost of sending multiple IPIs, there is no instruction that exposes this functionality to userspace.

Our workaround is to take advantage of the fact that any interrupt sent on a core's posted interrupt vector can trigger user interrupt handlers on the receiving side. We mapped the interrupt descriptors for each core into the scheduler's memory so it can directly post the interrupt information, and exposed a custom system call through the UIPI kernel module that sends a multicast interrupt on the posted interrupt vector (we use the same vector on all cores). Upon receipt of the interrupt, hardware observes the interrupt in the descriptor and transfers control to the userspace interrupt handler. This approach still requires the scheduler to perform a system call, but this cost is amortized across multiple interrupts. If necessary, the APIC could be exposed directly to the scheduler in the future since it is a trusted component. The receiver side still benefits because it avoids interacting with the kernel.

Register saving. uThreads that are interrupted by UIPIs must save their registers as well as *extended* CPU state, which includes all vector registers and can total many kilobytes. x86 provides an instruction pair, XSAVE and XRSTOR, that can be used in usermode to save and restore extended CPU states. It also provides two variants of XSAVE that aim to reduce overhead. XSAVEOPT avoids saving states that were not modified since the previous XRSTOR, while XSAVEC reduces memory fragmentation by saving to a compacted memory layout. Both instructions save only *active* states. Intel warns that XSAVEOPT is not recommended for user applications, because it is possible for the processor to mistakenly correlate save/restore pairs between different applications [26]. We invested some effort to ensure that XSAVEOPT was only used when it was paired with a correct XRSTOR, however, we found that XSAVEC was just as fast as XSAVEOPT, so Junction uses this instruction when saving extended states.

Interactions with system calls. POSIX signals delivered during a blocking interruptible system call should cause the system call to return immediately with an error, so that signal handlers can run without delay. Junction must support this at two levels. First, system calls that block inside the Junction kernel (*e.g.,* blocking network socket reads) must be unblocked by signals. Each uThread in Junction has an atomic flag that is used to coordinate blocking with wakeups from signal senders. Second, a uProc system call that results in a host kernel system call (*e.g.,* mmap()) must also be unblocked when a signal is sent. Junction's UIPI kernel module handles this by redirecting the local core's posted interrupt vector whenever a Junction kThread enters a Linux system call. This causes future interrupts on this vector (*i.e.,* those sent by SENDUIPI or multicast) to be delivered to the kernel module. Because the centralized core scheduler assigns kThreads to cores with exclusive grants, the kernel module is able to easily locate and wake up the blocked kThread.

Evaluation. We examined the impact of UIPIs on core allocation, timeslicing, and inter-thread signalling. We found that it only marginally improved latency and efficiency when replacing Linux signals for preempting kThreads to reallocate cores. We also have yet to find any real-world applications that use signals frequently enough to benefit from the speedups of UIPIs. We did observe, however, a significant reduction in overhead for Junction's uThread timeslicing. In Figure 9, we measure this overhead for a range of scheduler quantums with UIPIs and Linux signals. We observe that UIPIs reduce timeslicing overhead by an average of 2.35×. In practice, this unlocks the ability to timeslice uThreads at much finer granularity, which is beneficial for reining in the tail latency of microsecond-scale workloads with high service time dispersion [9, 27, 29].

B Buffer Management

Keeping the buffer memory footprint of a kernel bypass application low is critical for density. To address this, Junction leverages two NIC hardware features simultaneously—shared buffer queues and multi-packet receive buffers.

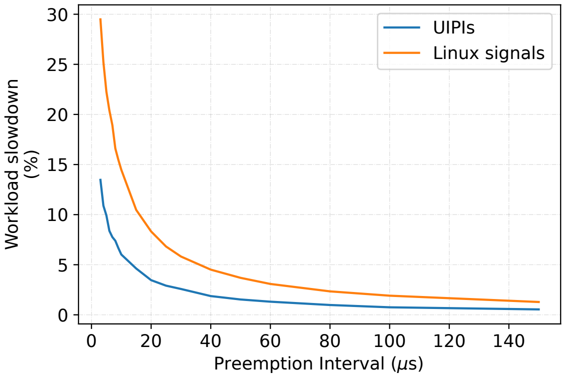


Figure 9: Comparison of preemption overheads for UIPIs versus Linux signals for a synthetic workload performing square root computations. UIPIs on average reduce slowdowns by 2.35×.

Using a shared buffer queue with per-core receive queues introduces the need to coordinate across cores. Posting buffers to the shared buffer queue involves two steps: writing the buffer address into an available descriptor in the queue, and advancing the index pointer forward to notify the NIC that the descriptor is ready. When a core receives a packet on its receive queue, it can immediately refill the corresponding descriptor in the shared buffer queue with a new buffer. However, it must wait to update the index pointer because the preceding buffers may have not yet been received by other cores. Therefore, if a core is slow to poll its receive queue, it could stall the entire buffer posting process.

Using multi-packet buffers adds additional complexity, since posted contiguous buffers are split into many smaller packets and distributed to multiple cores. In addition to coordinating updates to the index pointer, cores must coordinate (A) when to refill an individual descriptor, which cannot happen until all of its individual packets are delivered; and (B) when to mark a buffer as eligible to be reposted, which cannot happen until packet processing has freed each individual packet in that buffer.

Prior work on ShRing also explored using these hardware features, but for a different goal of optimizing DDIO cache usage [45]. The authors argued for hardware changes to enable efficient coordination. They proposed out-of-order posting to shared buffer queues, to prevent a busy core from delaying freeing slots. Instead, other cores can continue to supply buffers to the NIC without waiting. This change is coupled with a change to reduce the high cost of reference counting buffers. Their solution is for the hardware to batch multiple packets into a single buffer, but not deliver multiple packets within one buffer to multiple cores. Instead, a buffer becomes associated with a core when the first packet is delivered, and only future deliveries to that core can use the buffer.

To the contrary, Junction shows that low coordination overhead can be achieved with existing hardware (*i.e.,* without these changes). First, Junction relies on both UIPIs and work stealing across kThread receive queues to ensure that processing delays cannot starve the shared buffer queue. Second, Junction's per-core reference counting reduces synchronization overhead by allowing kThreads to update reference counters without frequently modifying shared cache lines.

When a kThread receives a packet, it increments a core-local counter corresponding to the packet's descriptor by the number of bytes consumed. Once the sum of the per-core counters is equal to the full size of the buffer, no further data can be written, so it is safe to replenish the slot with a new buffer. Similarly, after a packet is dequeued from the NIC, it undergoes protocol processing and is delivered to the application. Eventually, a core frees a packet by incrementing its local counter corresponding to the buffer backing the packet. Once all the packets in a buffer are free, the buffer can be reused. The scheduler core monitors the number of posted buffers and sends a notification over shared memory when the queue needs to be refilled. This notification triggers a high-priority refill thread that is responsible for managing the shared buffer queue, which collects the reference counter sums and refills the queue. This thread's logic is shown in algorithm 1.

Algorithm 1: Refill Pool Algorithm

```
Function RefillPool():
    // Refill free slots in the shared queue
    while true do
        index ← CompletedBufs mod $|RQ|$
        cnt ← 0
        foreach cpu do
            cnt += SlotCompletions[cpu][index]
        end
        if $cnt = PKTS\_PER\_BUF$ then
            ResetSlotRefs(index)
            BusyBufs.append(SharedQ[index])
            SharedQ[index] ← FreeBufs.pop()
            CompletedBufs += 1
        else
            break
        end
    end
    // Find free buffers
    foreach buf in BusyBufs do
        index ← index_of(buf)
        cnt ← 0
        foreach cpu do
            cnt += BufCompletions[cpu][index]
        end
        if $cnt = PKTS\_PER\_BUF$ then
            ResetBufRef(index)
            FreeBufs.push(buf)
            BusyBufs.erase(buf)
        end
    end
```

C Additional Benchmarks

C.1 Threading & Networking

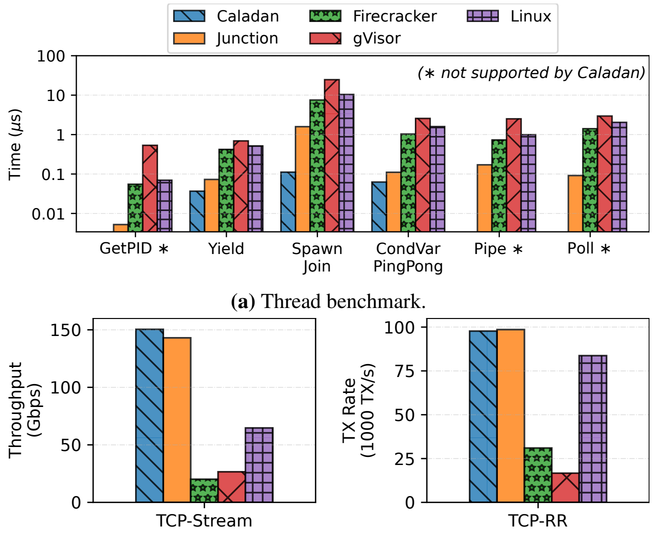
In this section, we evaluate the performance of Junction's threading and networking primitives.

Threading microbenchmarks. Figure 10a shows a set of microbenchmarks that measure the performance of several common threading operations. We use the same binary for each system except Caladan, which required a custom implementation due to its lack of compatibility. Each operation is performed in a loop for 1,000,000 iterations to determine the time per operation. The GetPID benchmark measures the baseline cost of performing a syscall (`getpid()`). In Junction, syscalls are replaced with function calls, resulting in low overheads ($\approx$ 10 ns). Linux, Firecracker, and gVisor pay a penalty for switching between user and kernel mode, while gVisor pays additional penalties for its system call interception techniques.

Yield and SpawnJoin measure overheads of the threading subsystems. Relative to Caladan, Junction can context switch between threads nearly as fast, but pays additional costs for thread creation and teardown because of POSIX compatibility and support for TLS—each thread must allocate and initialize thread local variables when created. CondVar and Pipe measure the costs of synchronizing two threads using condition variables and pipes; the Poll benchmark does the same but uses `poll()` on non-blocking pipes. In most cases, Junction fares at least an order of magnitude better than the systems that rely on kernel crossings. Caladan's performance on the CondVar benchmark is slightly better than Junction's in part due to its ability to inline synchronization functions (while in Junction these calls must traverse the trampoline page).

PARSEC. We now show how these threading primitives translate to Junction's end-to-end performance in the PARSEC benchmark suite [7], which consists of a set of compute-bound applications that exhibit high degrees of parallelism with varying synchronization strategies. Figure 11 shows the time taken to execute each PARSEC benchmark relative to Linux. As expected, due to the low overall frequency of system calls, Junction, Firecracker, and gVisor achieve mostly comparable performance to Linux. However, in benchmarks where there is heavier thread synchronization, Junction outperforms Firecracker and gVisor because of its better load balancing and more efficient system calls.

TCP Microbenchmarks. We evaluate the performance of Junction's kernel bypass networking and TCP/IP stack using the standard netperf benchmark (TCP-Stream and TCP-RR). For each experiment, we use *a single flow* and configure each system with a single thread (but do not restrict SoftIRQ processing on other cores). We use the same binary for every system but Caladan, for which we wrote a compatible imple-



(a) Thread benchmark.

(b) Networking benchmark.

Figure 10: Performance comparison with threading and networking microbenchmarks. On most threading benchmarks, Junction improves performance nearly 10× relative to non-kernel bypass systems. Junction is also able to to sustain TCP throughputs of up to 146Gbps with a single flow on a single core, a 2.2× improvement over Linux. Junction's low overhead network stack enables it to achieve high transaction rates (100,000 messages per second), a 3.2–6.0× speedup over Firecracker and gVisor.

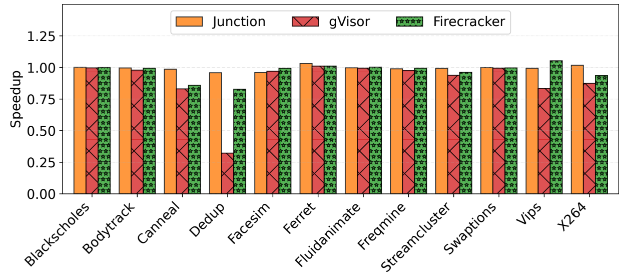


Figure 11: PARSEC benchmark speedup, normalized to Linux. Junction is able to offer comparable performance to Linux, while other systems perform worse in situations with heavier synchronization.

mentation. This benchmark runs between two machines that are connected back-to-back with a 200GbE link.

The TCP-Stream benchmark measures data transfer rates. Figure 10b shows that Junction and Caladan are able to achieve 146 and 154 Gbps respectively, greater than 2.2× the performance of the next best system (Linux). Junction's improvements are even more pronounced relative to Firecracker and gVisor, improving throughput 5.4–7.1×.

TCP-RR benchmarks the request/response rate which is determined by the round trip latency between two TCP stacks. Both Junction and Caladan achieve close to 100,000 transactions per second, indicating a round trip time of 10 μs. Linux adds marginal overhead with round trip times of 11 μs. Firecracker and gVisor, however, have 32 μs and 60 μs latencies respectively. They both use TAP devices for networking, resulting in additional scheduling and processing hops on top of Linux.

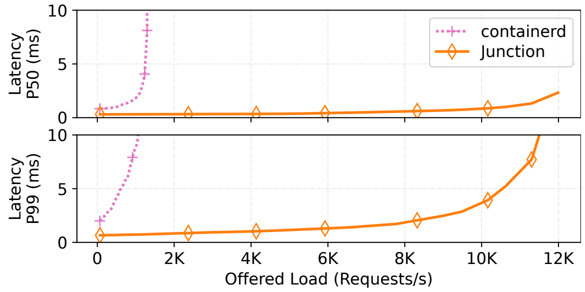


Figure 12: Benefits of using Junction in an end-to-end experiment with the serverless framework faasd. Accelerating each component in the system using Junction leads to compounding benefits for tail latency and throughput, with 3.5× lower tail latency and 10× higher total system throughput.

C.2 FaaS Benchmark

We also evaluate the end-to-end performance of Junction with a cloud framework to study the impact of adopting it across a distributed system instead of just one application. We use faasd [11], an open source serverless orchestration framework based on OpenFaaS [41]. faasd uses Linux containers to sandbox untrusted user applications, deployed by containerd [8]. It includes two orchestration services written in Go: a front-end load balancer and a per-host provider that communicates with containerd. To demonstrate the performance benefits of Junction, we replace containerd with *junctiond*, a C++ component that manages local instances of Junction. We also run the two orchestration services inside Junction. We evaluate the setup using invocations of a serverless function from vSwarm [59, 63] that encrypt a 600 byte input with AES. We do not evaluate cold-starts here, but we separately profiled the startup costs for a single-threaded Junction instance and found that Junction takes 3.4ms to initialize. These experiments used two machines with 10 core Xeon 4114 CPUs running at 2.2GHz, 48GB of RAM, and 100GbE NICs.

Figure 12 shows the median and tail latency across varying request rates offered via the front-end load balancer. Junction can sustain up to 10× more throughput while lowering the latency by ∼ 2× at the median and ∼ 3.5× at the tail. This reflects the compounding end-to-end benefit of using Junction across multiple components running in separate instances. More discussion of the faasd architecture and this experiment are available in [54].

References

[1] Alexandru Agache, Marc Brooker, Andreea Florescu, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight Virtualization for Serverless Applications. In *NSDI*, 2020.

[2] Amazon Web Services. The Security Design of the AWS Nitro System: AWS Whitepaper. Technical report, November 2022.

[3] Anjali, Tyler Caraza-Harter, and Michael M. Swift. Blending Containers and Virtual Machines: A Study of Firecracker and GVisor. In *VEE*, 2020.

[4] Jonathan Behrens, Adam Belay, and M. Frans Kaashoek. Performance evolution of mitigating transient execution attacks. In *EuroSys*, 2022.

[5] Adam Belay, Andrea Bittau, Ali José Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe User-level Access to Privileged CPU Features. In *OSDI*, 2012.

[6] 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 *OSDI*, 2014.

[7] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *PACT*, 2008.

[8] containerd. containerd overview, 2023.

[9] Henri Maxime Demoulin, Joshua Fried, Isaac Pedisich, Marios Kogias, Boon Thau Loo, Linh Thi Xuan Phan, and Irene Zhang. When Idling is Ideal: Optimizing Tail-Latency for Heavy-Tailed Datacenter Workloads with PerséPhone. In *SOSP*, 2021.

[10] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *SOSP*, 1995.

[11] faasd. A lightweight & portable FaaS engine, 2023.

[12] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *NSDI*, 2018.

[13] Brad Fitzpatrick. Distributed caching with memcached. *Linux journal*, 2004.

[14] Linux Foundation. Data Plane Development Kit (DPDK), 2015.

[15] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating Interference at Microsecond Timescales. In *OSDI*, 2020.

[16] Alexander Fuerst, Stanko Novaković, Íñigo Goiri, Gohar Irfan Chaudhry, Prateek Sharma, Kapil Arya, Kevin Broas, Eugene Bak, Mehmet Iyigun, and Ricardo Bianchini. Memory-Harvesting VMs in Cloud Platforms. In *ASPLOS*, 2022.

[17] Google. gVisor Documentation, 2019.

[18] Google. gVisor Gofer Syscalls, 2022.

[19] Google. gVisor Sentry Syscalls, 2023.

[20] Boncheol Gu, Andre S Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, et al. Biscuit: A framework for near-data processing of big data workloads. In *ISCA*, 2016.

[21] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. MegaPipe: A New Programming Interface for Scalable Network I/O. In *OSDI*, 2012.

[22] Steven Hand, Andrew Warfield, Keir Fraser, Evangelos Kotsovinos, and Daniel J Magenheimer. Are Virtual Machine Monitors Microkernels Done Right? In *HotOS*, 2005.

[23] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner, and Christos Kozyrakis. GhOSt: Fast & Flexible User-Space Delegation of Linux Scheduling. In *SOSP*, 2021.

[24] Takayuki Imada. Mirageos unikernel with network acceleration for iot cloud environments. In *ICCBDC*, 2018.

[25] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 2B*, December 2022.

[26] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 1*, June 2023.

[27] Rishabh Iyer, Musa Unal, Marios Kogias, and George Candea. Achieving Microsecond-Scale Tail Latency Efficiently with Approximate Optimal Scheduling. In *SOSP*, SOSP '23, New York, NY, USA, 2023.

[28] Eun Young Jeong, Shinae Woo, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. MTCP: A Highly Scalable User-Level TCP Stack for Multicore Systems. In *NSDI*, 2014.

[29] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive Scheduling for µSecond-Scale Tail Latency. In *NSDI*, 2019.

[30] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Datacenter RPCs Can Be General and Fast. In *NSDI*, 2019.

[31] Svilen Kanev, Juan Pablo Darago, Kim M. Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David M. Brooks. Profiling a warehouse-scale computer. In *ISCA*, 2015.

[32] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr. Sharma, Arvind Krishnamurthy, and Thomas Anderson. TAS: TCP Acceleration as an OS Service. In *EuroSys*, 2019.

[33] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. OSv: Optimizing the Operating System for Virtual Machines. In *USENIX ATC*, 2014.

[34] Simon Kuenzer, Vlad-Andrei Bădoiu, Hugo Lefeuvre, Sharan Santhanam, Alexander Jung, Gaulthier Gain, Cyril Soldani, Costin Lupu, Ştefan Teodorescu, Costi Răducanu, et al. Unikraft: fast, specialized unikernels the easy way. In *EuroSys*, 2021.

[35] H.J. Lu, Michael Matz, Jan Hubicka, Andreas Jaeger, and Mark Mitchell. System V Application Binary Interface. *AMD64 Architecture Processor Supplement*, 2018.

[36] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *EuroSys*, 2012.

[37] Ilias Marinos, Robert N.M. Watson, and Mark Handley. Network Stack Specialization for Performance. In *SIGCOMM*, 2014.

[38] Sohil Mehta. x86 User Interrupts support, 2021. Available at https://lore.kernel.org/lkml/20210913200132.3396598-1-sohil.mehta@intel.com/.

[39] Jeff Meyerson. The Go programming language. *IEEE software*, 31(5):104–104, 2014.

[40] Node.js. Node.js, 2023.

[41] OpenFaaS. Serverless functions made simple, 2023.

[42] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high cpu efficiency for latency-sensitive datacenter workloads. In *NSDI*, 2019.

[43] 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), aug 2015.

[44] 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 *OSDI*, 2015.

[45] Boris Pismenny, Adam Morrison, and Dan Tsafrir. ShRing: Networking with Shared Receive Rings. In *OSDI*, 2023.

[46] Donald E Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C Hunt. Rethinking the library OS from the top down. In *ASPLOS*, 2011.

[47] George Prekas, Marios Kogias, and Edouard Bugnion. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In *SOSP*, 2017.

[48] George Prekas, Mia Primorac, Adam Belay, Christos Kozyrakis, and Edouard Bugnion. Energy proportionality and workload consolidation for latency-critical applications. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, page 342–355, New York, NY, USA, 2015. Association for Computing Machinery.

[49] Ron Pressler. On the performance of user-mode threads and coroutines, 8 2020. Accessed: September 2023.

[50] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. Arachne: Core-Aware Thread Management. In *OSDI*, 2018.

[51] Will Reese. Nginx: the high-performance web server and reverse proxy. *Linux Journal*, 2008.

[52] Luigi Rizzo. netmap: a novel framework for fast packet I/O. In *USENIX Security*, 2012.

[53] Rocket. Rocket, 2016.

[54] Enrique Saurez, Joshua Fried, Gohar Irfan Chaudhry, Esha Choukse, Íñigo Goiri, Sameh Elnikety, Adam Belay, and Rodrigo Fonseca. Junctiond: Extending FaaS Runtimes with Kernel-Bypass, March 2024.

[55] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: Flexible, scalable schedulers for large compute clusters. In *EuroSys*, 2013.

[56] Zhiming Shen, Zhen Sun, Gur-Eyal Sela, Eugene Bagdasaryan, Christina Delimitrou, Robbert Van Renesse, and Hakim Weatherspoon. X-containers: Breaking down barriers to improve performance and isolation of cloud-native containers. In *ASPLOS*, 2019.

[57] Mike Stemle. The State of Serverless, 2023.

[58] Tomcat. Tomcat, 1999.

[59] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, and Edouard Bugnionand Boris Grot. Benchmarking, analysis, and optimization of serverless function snapshots. In *ASPLOS*, 2021.

[60] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009.

[61] George Varghese and Anthony Lauck. Hashed and hierarchical timing wheels: efficient data structures for implementing a timer facility. *IEEE/ACM transactions on networking*, 5(6):824–834, 1997.

[62] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-Scale Cluster Management at Google with Borg. In *EuroSys*, 2015.

[63] vSwarm. Serverless benchmarking suite, 2023.

[64] Dan Williams and Ricardo Koller. Unikernel Monitors: Extending Minimalism Outside of the Box. In *HotCloud*, 2016.

[65] Dan Williams, Ricardo Koller, and Brandon Lum. Say goodbye to virtualization for a safer cloud. In *HotCloud*, 2018.

[66] Kenichi Yasukata, Hajime Tazaki, Pierre-Louis Aublin, and Kenta Ishiguro. zpoline: a system call hook mechanism based on binary rewriting. In *USENIX ATC*, 2023.

[67] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S. Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, Pedro Henrique Penna, Max Demoulin, Piali Choudhury, and Anirudh Badam. The Demikernel Datapath OS Architecture for Microsecond-Scale Datacenter Systems. In *SOSP*, 2021.

[68] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. CPI^2: CPU performance isolation for shared compute clusters. In *EuroSys*, 2013.