IX: A Protected Dataplane Operating System for High Throughput and Low Latency

Adam Belay[1] George Prekas[2] Ana Klimovic[1] Samuel Grossman[1]
Christos Kozyrakis[1] Edouard Bugnion[2]
[1]Stanford University [2]EPFL

Abstract

The conventional wisdom is that aggressive networking requirements, such as high packet rates for small messages and microsecond-scale tail latency, are best addressed outside the kernel, in a user-level networking stack. We present IX, a *dataplane operating system* that provides high I/O performance, while maintaining the key advantage of strong protection offered by existing kernels. IX uses hardware virtualization to separate management and scheduling functions of the kernel (control plane) from network processing (dataplane). The dataplane architecture builds upon a native, zero-copy API and optimizes for both bandwidth and latency by dedicating hardware threads and networking queues to dataplane instances, processing bounded batches of packets to completion, and by eliminating coherence traffic and multi-core synchronization. We demonstrate that IX outperforms Linux and state-of-the-art, user-space network stacks significantly in both throughput and end-to-end latency. Moreover, IX improves the throughput of a widely deployed, key-value store by up to 3.6× and reduces tail latency by more than 2×.

1 Introduction

Datacenter applications such as search, social networking, and e-commerce platforms are redefining the requirements for systems software. A single application can consist of hundreds of software services, deployed on thousands of servers, creating a need for networking stacks that provide more than high streaming performance. The new requirements include high packet rates for short messages, microsecond-level responses to remote requests with tight tail latency guarantees, and support for high connection counts and churn [2, 14, 46]. It is also important to have a strong protection model and be elastic in resource usage, allowing other applications to use any idling resources in a shared cluster.

The conventional wisdom is that there is a basic mismatch between these requirements and existing networking stacks in commodity operating systems. Consequently, some systems bypass the kernel and implement the networking stack in user-space [29, 32, 40, 59, 61]. While kernel bypass eliminates context switch overheads, on its own it does not eliminate the difficult tradeoffs between high packet rates and low latency (see §5.2). Moreover, user-level networking suffers from lack of protection. Application bugs and crashes can corrupt the networking stack and impact other workloads. Other systems go a step further by also replacing TCP/IP with RDMA in order to offload network processing to specialized adapters [17, 31, 44, 47]. However, such adapters must be present at both ends of the connection and can only be used within the datacenter.

We propose IX, an operating system designed to break the 4-way tradeoff between high throughput, low latency, strong protection, and resource efficiency. Its architecture builds upon the lessons from high performance middleboxes, such as firewalls, load-balancers, and software routers [16, 34]. IX separates the control plane, which is responsible for system configuration and coarse-grain resource provisioning between applications, from the dataplanes, which run the networking stack and application logic. IX leverages Dune and virtualization hardware to run the dataplane kernel and the application at distinct protection levels and to isolate the control plane from the dataplane [7]. In our implementation, the control plane is the full Linux kernel and the dataplanes run as protected, library-based operating systems on dedicated hardware threads.

The IX dataplane allows for networking stacks that optimize for both bandwidth and latency. It is designed around a native, zero-copy API that supports processing of bounded batches of packets to completion. Each dataplane executes all network processing stages for a batch of packets in the dataplane kernel, followed by the associ-

Appears in Proc. 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2014

ated application processing in user mode. This approach amortizes API overheads and improves both instruction and data locality. We set the batch size adaptively based on load. The IX dataplane also optimizes for multi-core scalability. The network adapters (NICs) perform flow-consistent hashing of incoming traffic to distinct queues. Each dataplane instance exclusively controls a set of these queues and runs the networking stack and a single application without the need for synchronization or coherence traffic during common case operation. The IX API departs from the POSIX API, and its design is guided by the commutativity rule [12]. However, the `libix` user-level library includes an event-based API similar to the popular `libevent` library [51], providing compatibility with a wide range of existing applications.

We compare IX with a TCP/IP dataplane against Linux 3.16.1 and mTCP, a state-of-the-art user-level TCP/IP stack [29]. On a 10GbE experiment using short messages, IX outperforms Linux and mTCP by up to 10× and 1.9× respectively for throughput. IX further scales to a 4x10GbE configuration using a single multi-core socket. The unloaded uni-directional latency for two IX servers is 5.7µs, which is 4× better than between standard Linux kernels and an order of magnitude better than mTCP, as both trade-off latency for throughput. Our evaluation with memcached, a widely deployed key-value store, shows that IX improves upon Linux by up to 3.6× in terms of throughput at a given 99th percentile latency bound, as it can reduce kernel time, due essentially to network processing, from $\sim 75\%$ with Linux to $< 10\%$ with IX.

IX demonstrates that, by revisiting networking APIs and taking advantage of modern NICs and multi-core chips, we can design systems that achieve high throughput and low latency and robust protection and resource efficiency. It also shows that, by separating the small subset of performance-critical I/O functions from the rest of the kernel, we can architect radically different I/O systems and achieve large performance gains, while retaining compatibility with the huge set of APIs and services provided by a modern OS like Linux.

The rest of the paper is organized as follows. §2 motivates the need for a new OS architecture. §3 and §4 present the design principles and implementation of IX. §5 presents the quantitative evaluation. §6 and §7 discuss open issues and related work.

2 Background and Motivation

Our work focuses on improving operating systems for applications with aggressive networking requirements running on multi-core servers.

2.1 Challenges for Datacenter Applications

Large-scale, datacenter applications pose unique challenges to system software and their networking stacks:

Microsecond tail latency: To enable rich interactions between a large number of services without impacting the overall latency experienced by the user, it is essential to reduce the latency for some service requests to a few tens of µs [3, 54]. Because each user request often involves hundreds of servers, we must also consider the long tail of the latency distributions of RPC requests across the datacenter [14]. Although tail-tolerance is actually an end-to-end challenge, the system software stack plays a significant role in exacerbating the problem [36]. Overall, each service node must ideally provide tight bounds on the 99th percentile request latency.

High packet rates: The requests and, often times, the replies between the various services that comprise a datacenter application are quite small. In Facebook's `memcached` service, for example, the vast majority of requests uses keys shorter than 50 bytes and involves values shorter than 500 bytes [2], and each node can scale to serve millions of requests per second [46].

The high packet rate must also be sustainable under a large number of concurrent connections and high connection churn [23]. If the system software cannot handle large connection counts, there can be significant implications for applications. The large connection count between application and `memcached` servers at Facebook made it impractical to use TCP sockets between these two tiers, resulting in deployments that use UDP datagrams for `get` operations and an aggregation proxy for `put` operations [46].

Protection: Since multiple services commonly share servers in both public and private datacenters [14, 25, 56], there is need for isolation between applications. The use of kernel-based or hypervisor-based networking stacks largely addresses the problem. A trusted network stack can firewall applications, enforce access control lists (ACLs), and implement limiters and other policies based on bandwidth metering.

Resource efficiency: The load of datacenter applications varies significantly due to diurnal patterns and spikes in user traffic. Ideally, each service node will use the fewest resources (cores, memory, or IOPS) needed to satisfy packet rate and tail latency requirements at any point. The remaining server resources can be allocated to other applications [15, 25] or placed into low power mode for energy efficiency [4]. Existing operating systems can support such resource usage policies [36, 38].

2.2 The Hardware – OS Mismatch

The wealth of hardware resources in modern servers should allow for low latency and high packet rates for datacenter applications. A typical server includes one or two processor sockets, each with eight or more multithreaded cores and multiple, high-speed channels to DRAM and PCIe devices. Solid-state drives and PCIe-based Flash storage are also increasingly popular. For networking, 10 GbE NICs and switches are widely deployed in datacenters, with 40 GbE and 100 GbE technologies right around the corner. The combination of tens of hardware threads and 10 GbE NICs should allow for rates of 15M packets/sec with minimum sized packets. We should also achieve 10–20µs round-trip latencies given 3µs latency across a pair of 10 GbE NICs, one to five switch crossings with cut-through latencies of a few hundred ns each, and propagation delays of 500ns for 100 meters of distance within a datacenter.

Unfortunately, commodity operating systems have been designed under very different hardware assumptions. Kernel schedulers, networking APIs, and network stacks have been designed under the assumptions of multiple applications sharing a single processing core and packet inter-arrival times being many times higher than the latency of interrupts and system calls. As a result, such operating systems trade off both latency and throughput in favor of fine-grain resource scheduling. Interrupt coalescing (used to reduce processing overheads), queuing latency due to device driver processing intervals, the use of intermediate buffering, and CPU scheduling delays frequently add up to several hundred µs of latency to remote requests. The overheads of buffering and synchronization needed to support flexible, fine-grain scheduling of applications to cores increases CPU and memory system overheads, which limits throughput. As requests between service tiers of datacenter applications often consist of small packets, common NIC hardware optimizations, such as TCP segmentation and receive side coalescing, have a marginal impact on packet rate.

2.3 Alternative Approaches

Since the network stacks within commodity kernels cannot take advantage of the abundance of hardware resources, a number of alternative approaches have been suggested. Each alternative addresses a subset, but not all, of the requirements for datacenter applications.

User-space networking stacks: Systems such as OpenOnload [59], mTCP [29], and Sandstorm [40] run the entire networking stack in user-space in order to eliminate kernel crossing overheads and optimize packet processing without incurring the complexity of kernel modifications. However, there are still tradeoffs between packet rate and latency. For instance, mTCP uses dedicated threads for the TCP stack, which communicate at relatively coarse granularity with application threads. This aggressive batching amortizes switching overheads at the expense of higher latency (see §5). It also complicates resource sharing as the network stack must use a large number of hardware threads regardless of the actual load. More importantly, security tradeoffs emerge when networking is lifted into the user-space and application bugs can corrupt the networking stack. For example, an attacker may be able to transmit raw packets (a capability that normally requires root privileges) to exploit weaknesses in network protocols and impact other services [8]. It is difficult to enforce any security or metering policies beyond what is directly supported by the NIC hardware.

Alternatives to TCP: In addition to kernel bypass, some low-latency object stores rely on RDMA to offload protocol processing on dedicated Infiniband host channel adapters [17, 31, 44, 47]. RDMA can reduce latency, but requires that specialized adapters be present at both ends of the connection. Using commodity Ethernet networking, Facebook's memcached deployment uses UDP to avoid connection scalability limitations [46]. Even though UDP is running in the kernel, reliable communication and congestion management are entrusted to applications.

Alternatives to POSIX API: MegaPipe replaces the POSIX API with lightweight sockets implemented with in-memory command rings [24]. This reduces some software overheads and increases packet rates, but retains all other challenges of using an existing, kernel-based networking stack.

OS enhancements: Tuning kernel-based stacks provides incremental benefits with superior ease of deployment. Linux `SO_REUSEPORT` allows multi-threaded applications to accept incoming connections in parallel. Affinity-accept reduces overheads by ensuring all processing for a network flow is affinitized to the same core [49]. Recent Linux Kernels support a busy polling driver mode that trades increased CPU utilization for reduced latency [27], but it is not yet compatible with `epoll`. When microsecond latencies are irrelevant, properly tuned stacks can maintain millions of open connections [66].

3 IX Design Approach

The first two requirements in §2.1 — microsecond latency and high packet rates — are not unique to datacenter applications. These requirements have been addressed in the design of middleboxes such as firewalls, load-balancers, and software routers [16, 34] by integrating the network-

ing stack and the application into a single *dataplane*. The two remaining requirements — protection and resource efficiency — are not addressed in middleboxes because they are single-purpose systems, not exposed directly to users.

Many middlebox dataplanes adopt design principles that differ from traditional OSes. First, they *run each packet to completion*. All network protocol and application processing for a packet is done before moving on to the next packet, and application logic is typically intermingled with the networking stack without any isolation. By contrast, a commodity OS decouples protocol processing from the application itself in order to provide scheduling and flow control flexibility. For example, the kernel relies on device and soft interrupts to context switch from applications to protocol processing. Similarly, the kernel's network stack will generate TCP `ACKs` and slide its receive window even when the application is not consuming data, up to an extent. Second, middlebox dataplanes optimize for *synchronization-free operation* in order to scale well on many cores. Network flows are distributed into distinct queues via flow-consistent hashing and common case packet processing requires no synchronization or coherence traffic between cores. By contrast, commodity OSes tend to rely heavily on coherence traffic and are structured to make frequent use of locks and other forms of synchronization.

IX extends the dataplane architecture to support untrusted, general-purpose applications and satisfy all requirements in §2.1. Its design is based on the following key principles:

Separation and protection of control and data plane: IX separates the control function of the kernel, responsible for resource configuration, provisioning, scheduling, and monitoring, from the dataplane, which runs the networking stack and application logic. Like a conventional OS, the control plane multiplexes and schedules resources among dataplanes, but in a coarse-grained manner in space and time. Entire cores are dedicated to dataplanes, memory is allocated at large page granularity, and NIC queues are assigned to dataplane cores. The control plane is also responsible for elastically adjusting the allocation of resources between dataplanes.

The separation of control and data plane also allows us to consider radically different I/O APIs, while permitting other OS functionality, such as file system support, to be passed through to the control plane for compatibility. Similar to the Exokernel [19], each dataplane runs a single application in a single address space. However, we use modern virtualization hardware to provide three-way isolation between the control plane, the dataplane, and untrusted user code [7]. Dataplanes have capabilities similar to guest OSes in virtualized systems. They manage their own address translations, on top of the address space provided by the control plane, and can protect the networking stack from untrusted application logic through the use of privilege rings. Moreover, dataplanes are given direct pass-through access to NIC queues through memory mapped I/O.

Run to completion with adaptive batching: IX dataplanes run to completion all stages needed to receive and transmit a packet, interleaving protocol processing (kernel mode) and application logic (user mode) at well-defined transition points. Hence, there is no need for intermediate buffering between protocol stages or between application logic and the networking stack. Unlike previous work that applied a similar approach to eliminate receive livelocks during congestion periods [45], IX uses run to completion during all load conditions. Thus, we are able to use polling and avoid interrupt overhead in the common case by dedicating cores to the dataplane. We still rely on interrupts as a mechanism to regain control, for example, if application logic is slow to respond. Run to completion improves both message throughput and latency because successive stages tend to access many of the same data, leading to better data cache locality.

The IX dataplane also makes extensive use of batching. Previous systems applied batching at the system call boundary [24, 58] and at the network API and hardware queue level [29]. We apply batching in every stage of the network stack, including but not limited to system calls and queues. Moreover, we use batching *adaptively* as follows: (i) we never wait to batch requests and batching only occurs in the presence of congestion; (ii) we set an upper bound on the number of batched packets. Using batching only on congestion allows us to minimize the impact on latency, while bounding the batch size prevents the live set from exceeding cache capacities and avoids transmit queue starvation. Batching improves packet rate because it amortizes system call transition overheads and improves instruction cache locality, prefetching effectiveness, and branch prediction accuracy. When applied adaptively, batching also decreases latency because these same efficiencies reduce head-of-line blocking.

The combination of bounded, adaptive batching and run to completion means that queues for incoming packets can build up only at the NIC edge, before packet processing starts in the dataplane. The networking stack sends acknowledgments to peers only as fast as the application can process them. Any slowdown in the application-processing rate quickly leads to shrinking windows in peers. The dataplane can also monitor queue depths at

the NIC edge and signal the control plane to allocate additional resources for the dataplane (more hardware threads, increased clock frequency), notify peers explicitly about congestion (e.g., via ECN [52]), and make policy decisions for congestion management (e.g., via RED [22]).

Native, zero-copy API with explicit flow control: We do not expose or emulate the POSIX API for networking. Instead, the dataplane kernel and the application communicate at coordinated transition points via messages stored in memory. Our API is designed for true zero-copy operation in both directions, improving both latency and packet rate. The dataplane and application cooperatively manage the message buffer pool. Incoming packets are mapped read-only into the application, which may hold onto message buffers and return them to the dataplane at a later point. The application sends to the dataplane scatter/gather lists of memory locations for transmission but, since contents are not copied, the application must keep the content immutable until the peer acknowledges reception. The dataplane enforces flow control correctness and may trim transmission requests that exceed the available size of the sliding window, but the application controls transmit buffering.

Flow consistent, synchronization-free processing: We use multi-queue NICs with receive-side scaling (RSS [43]) to provide flow-consistent hashing of incoming traffic to distinct hardware queues. Each hardware thread (hyperthread) serves a single receive and transmit queue per NIC, eliminating the need for synchronization and coherence traffic between cores in the networking stack. The control plane establishes the mapping of RSS flow groups to queues to balance the traffic among the hardware threads. Similarly, memory management is organized in distinct pools for each hardware thread. The absence of a POSIX socket API eliminates the issue of the shared file descriptor namespace in multi-threaded applications [12]. Overall, the IX dataplane design scales well with the increasing number of cores in modern servers, which improves both packet rate and latency. This approach does not restrict the memory model for applications, which can take advantage of coherent, shared memory to exchange information and synchronize between cores.

4 IX Implementation

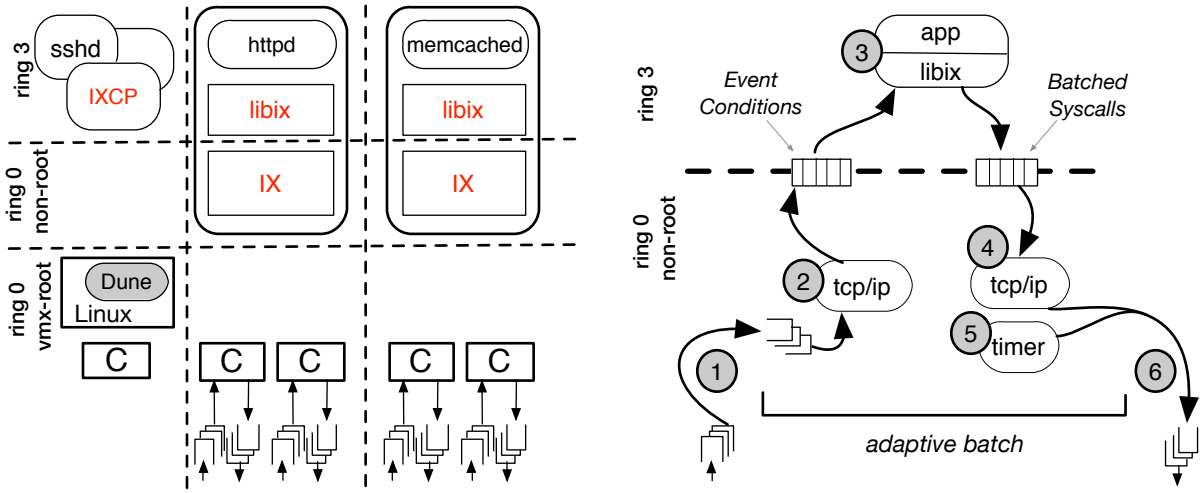
4.1 Overview

Fig. 1a presents the IX architecture, focusing on the separation between the control plane and the multiple dataplanes. The hardware environment is a multi-core server with one or more multi-queue NICs with RSS support. The IX control plane consists of the full Linux kernel and `IXCP`, a user-level program. The Linux kernel initializes PCIe devices, such as the NICs, and provides the basic mechanisms for resource allocation to the dataplanes, including cores, memory, and network queues. Equally important, Linux provides system calls and services that are necessary for compatibility with a wide range of applications, such as file system and signal support. `IXCP` monitors resource usage and dataplane performance and implements resource allocation policies. The development of efficient allocation policies involves understanding difficult tradeoffs between dataplane performance, energy proportionality, and resource sharing between co-located applications as their load varies over time. We leave the design of such policies to future work and focus primarily on the IX dataplane architecture.

We run the Linux kernel in VMX root ring 0, the mode typically used to run hypervisors in virtualized systems [62]. We use the Dune module within Linux to enable dataplanes to run as application-specific OSes in VMX non-root ring 0, the mode typically used to run guest kernels in virtualized systems [7]. Applications run in VMX non-root ring 3, as usual. This approach provides dataplanes with direct access to hardware features, such as page tables and exceptions, and pass-through access to NICs. Moreover, it provides full, three-way protection between the control plane, dataplanes, and untrusted application code.

Each IX dataplane supports a single, multithreaded application. For instance, Fig. 1a shows one dataplane for a multi-threaded `memcached` server and another dataplane for a multi-threaded `httpd` server. The control plane allocates resources to each dataplane in a coarse-grained manner. Core allocation is controlled through real-time priorities and `cpusets`; memory is allocated in large pages; each NIC hardware queue is assigned to a single dataplane. This approach avoids the overheads and unpredictability of fine-grained time multiplexing of resources between demanding applications [36].

Each IX dataplane operates as a single address-space OS and supports two thread types within a shared, user-level address space: (i) *elastic threads* which interact with the IX dataplane to initiate and consume network I/O and (ii) *background threads*. Both elastic and background threads can issue arbitrary POSIX system calls that are intermediated and validated for security by the dataplane before being forwarded to the Linux kernel. Elastic threads are expected to *not* issue blocking calls because of the adverse impact on network behavior resulting from delayed packet processing. Each elastic thread makes exclusive use of a core or hardware thread allocated



(a) Protection and separation of control and data plane.

(b) Interleaving of protocol processing and application execution.

Figure 1: The IX dataplane operating system.

to the dataplane in order to achieve high performance with predictable latency. In contrast, multiple background threads may timeshare an allocated hardware thread. For example, if an application were allocated four hardware threads, it could use all of them as elastic threads to serve external requests or it could temporarily transition to three elastic threads and use one background thread to execute tasks such as garbage collection. When the control plane revokes or allocates an additional hardware thread using a protocol similar to the one in Exokernel [19], the dataplane adjusts its number of elastic threads.

4.2 The IX Dataplane

We now discuss the IX dataplane in more detail. It differs from a typical kernel in that it is specialized for high performance network I/O and runs only a single application, similar to a library OS but with memory isolation. However, our dataplane still provides many familiar kernel-level services.

For memory management, we accept some internal memory fragmentation in order to reduce complexity and improve efficiency. All hot-path data objects are allocated from per hardware thread memory pools. Each memory pool is structured as arrays of identically sized objects, provisioned in page-sized blocks. Free objects are tracked with a simple free list, and allocation routines are inlined directly into calling functions. *Mbufs*, the storage object for network packets, are stored as contiguous chunks of bookkeeping data and MTU-sized buffers, and are used for both receiving and transmitting packets.

The dataplane also manages its own virtual address translations, supported through nested paging. In contrast to contemporary OSes, it uses exclusively large pages (2MB). We favor large pages due to their reduced address translation overhead [5, 7] and the relative abundance of physical memory resources in modern servers. The dataplane maintains only a single address space; kernel pages are protected with supervisor bits. We deliberately chose not to support swappable memory in order to avoid adding performance variability.

We provide a hierarchical timing wheel implementation for managing network timeouts, such as TCP retransmissions [63]. It is optimized for the common case where most timers are canceled before they expire. We support extremely high-resolution timeouts, as low as 16 µs, which has been shown to improve performance during TCP incast congestion [64].

Our current IX dataplane implementation is based on Dune and requires the VT-x virtualization features available on Intel x86-64 systems [62]. However, it could be ported to any architecture with virtualization support, such as ARM, SPARC, and Power. It also requires one or more Intel 82599 chipset NICs, but it is designed to easily support additional drivers. The IX dataplane currently consists of 39K SLOC [67] and leverages some existing codebases: 41% is derived from the DPDK variant of the Intel NIC device driver [28], 26% from the lwIP TCP/IP stack [18], and 15% from the Dune library. We did not use the remainder of the DPDK framework, and all three code bases are highly modified for IX. The rest is approximately 7K SLOC of new code. We chose lwIP as a starting point for TCP/IP processing because of its modularity and its maturity as a RFC-compliant, feature-rich networking stack. We implemented our own RFC-

System Calls (batched)		
Type	**Parameters**	**Description**
connect	cookie, dst_IP, dst_port	Opens a connection
accept	handle, cookie	Accepts a connection
sendv	handle, scatter_gather_array	Transmits a scatter-gather array of data
recv_done	handle, bytes_acked	Advances the receive window and frees memory buffers
close	handle	Closes or rejects a connection

Event Conditions		
Type	**Parameters**	**Description**
knock	handle, src_IP, src_port	A remotely initiated connection was opened
connected	cookie, outcome	A locally initiated connection finished opening
recv	cookie, mbuf_ptr, mbuf_len	A message buffer was received
sent	cookie, bytes_sent, window_size	A send completed and/or the window size changed
dead	cookie, reason	A connection was terminated

Table 1: The IX dataplane system call and event condition API.

compliant support for UDP, ARP, and ICMP. Since lwIP was optimized for memory efficiency in embedded environments, we had to radically change its internal data structures for multi-core scalability and fine-grained timer management. However, we did not yet optimize the lwIP code for performance. Hence, the results of §5 have room for improvement.

4.3 Dataplane API and Operation

The elastic threads of an application interact with the IX dataplane through three asynchronous, non-blocking mechanisms summarized in Table 1: they issue *batched systems calls* to the dataplane; they consume *event conditions* generated by the dataplane; and they have direct, but safe, access to (*mbuf*s) containing incoming payloads. The latter allows for zero-copy access to incoming network traffic. The application can hold on to mbufs until it asks the dataplane to release them via the `recv_done` batched system call.

Both batched system calls and event conditions are passed through arrays of shared memory, managed by the user and the kernel respectively. IX provides an unbatched system call (`run_io`) that yields control to the kernel and initiates a new run to completion cycle. As part of the cycle, the kernel overwrites the array of batched system call requests with corresponding return codes and populates the array of event conditions. The handles defined in Table 1 are kernel-level flow identifiers. Each handle is associated with a cookie, an opaque value provided by the user at connection establishment to enable efficient user-level state lookup [24].

IX differs from POSIX sockets in that it directly exposes flow control conditions to the application. The `sendv` system call does not return the number of bytes buffered. Instead, it returns the number of bytes that were accepted and sent by the TCP stack, as constrained by correct TCP sliding window operation. When the receiver acknowledges the bytes, a `sent` event condition informs the application that it is possible to send more data. Thus, send window-sizing policy is determined entirely by the application. By contrast, conventional OSes buffer send data beyond raw TCP constraints and apply flow control policy inside the kernel.

We built a user-level library, called `libix`, which abstracts away the complexity of our low-level API. It provides a compatible programming model for legacy applications and significantly simplifies the development of new applications. `libix` currently includes a very similar interface to `libevent` and non-blocking POSIX socket operations. It also includes new interfaces for zero-copy read and write operations that are more efficient, at the expense of requiring changes to existing applications.

`libix` automatically coalesces multiple write requests into single `sendv` system calls during each batching round. This improves locality, simplifies error handling, and ensures correct behavior, as it preserves the data stream order even if a transmit fails. Coalescing also facilitates transmit flow control because we can use the transmit vector (the argument to `sendv`) to keep track of outgoing data buffers and, if necessary, reissue writes when the transmit window has more available space, as notified by the `sent` event condition. Our buffer sizing policy is currently very basic; we enforce a maximum pending send byte limit, but we plan to make this more dynamic in the future [21].

Fig. 1b illustrates the run-to-completion operation for an elastic thread in the IX dataplane. NIC receive buffers are mapped in the server's main memory and the NIC's receive descriptor ring is filled with a set of buffer descriptors that allow it to transfer incoming packets using DMA. The elastic thread (1) polls the receive descriptor ring and

potentially posts fresh buffer descriptors to the NIC for use with future incoming packets. The elastic thread then (2) processes a bounded number of packets through the TCP/IP networking stack, thereby generating event conditions. Next, the thread (3) switches to the user-space application, which consumes all event conditions. Assuming that the incoming packets include remote requests, the application processes these requests and responds with a batch of system calls. Upon return of control from user-space, the thread (4) processes all batched system calls, and in particular the ones that direct outgoing TCP/IP traffic. The thread also (5) runs all kernel timers in order to ensure compliant TCP behavior. Finally (6), it places outgoing Ethernet frames in the NIC's transmit descriptor ring for transmission, and it notifies the NIC to initiate a DMA transfer for these frames by updating the transmit ring's tail register. In a separate pass, it also frees any buffers that have finished transmitting, based on the transmit ring's head position, potentially generating `sent` event conditions. The process repeats in a loop until there is no network activity. In this case, the thread enters a quiescent state which involves either hyperthread-friendly polling or optionally entering a power efficient C-state, at the cost of some additional latency.

4.4 Multi-core Scalability

The IX dataplane is optimized for multi-core scalability, as elastic threads operate in a synchronization and coherence free manner in the common case. This is a stronger requirement than lock-free synchronization, which requires expensive atomic instructions even when a single thread is the primary consumer of a particular data structure [13]. This is made possible through a set of conscious design and implementation tradeoffs.

First, system call implementations can only be synchronization-free if the API itself is commutative [12]. The IX API is commutative between elastic threads. Each elastic thread has its own flow identifier namespace, and an elastic thread cannot directly perform operations on flows that it does not own.

Second, the API implementation is carefully optimized. Each elastic thread manages its own memory pools, hardware queues, event condition array, and batched system call array. The implementation of event conditions and batched system calls benefits directly from the explicit, cooperative control transfers between IX and the application. Since there is no concurrent execution by producer and consumer, event conditions and batched system calls are implemented without synchronization primitives based on atomics.

Third, the use of flow-consistent hashing at the NICs ensures that each elastic thread operates on a disjoint subset of TCP flows. Hence, no synchronization or coherence occurs during the processing of incoming requests for a server application. For client applications with outbound connections, we need to ensure that the reply is assigned to the same elastic thread that made the request. Since we cannot reverse the Toeplitz hash used by RSS [43], we simply probe the ephemeral port range to find a port number that would lead to the desired behavior. Note that this implies that two elastic threads in a client cannot share a flow to a server.

IX does have a small number of shared structures, including some that require synchronization on updates. For example, the ARP table is shared by all elastic threads and is protected by RCU locks [41]. Hence, the common case reads are coherence-free but the rare updates are not. RCU objects are garbage collected after a quiescent period that spans the time it takes each elastic thread to finish a run to completion cycle.

IX requires synchronization when the control plane reallocates resources between dataplanes. For instance, when a core is revoked from a dataplane, the corresponding network flows must be assigned to another elastic thread. Such events are rare because resource allocation happens in a coarse-grained manner. Finally, the application code may include inter-thread communication and synchronization. While using IX does not eliminate the need to develop scalable application code, it ensures that there are no scaling bottlenecks in the system and protocol processing code.

4.5 Security Model

The IX API and implementation has a cooperative flow control model between application code and the network-processing stack. Unlike user-level stacks, where the application is trusted for correct networking behavior, the IX protection model makes few assumptions about the application. A malicious or misbehaving application can only hurt itself. It cannot corrupt the networking stack or affect other applications. All application code in IX runs in user-mode, while dataplane code runs in protected ring 0. Applications cannot access dataplane memory, except for read-only message buffers. No sequence of batched system calls or other user-level actions can be used to violate correct adherence to TCP and other network specifications. Furthermore, the dataplane can be used to enforce network security policies, such as firewalling and access control lists. The IX security model is as strong as conventional kernel-based networking stacks, a feature that is missing from all recently proposed user-level stacks.

The IX dataplane and the application collaboratively manage memory. To enable zero-copy operation, a buffer used for an incoming packet is passed read-only to the ap-

plication, using virtual memory protection. Applications are encouraged (but not required) to limit the time they hold message buffers, both to improve locality and to reduce fragmentation because of the fixed size of message buffers. In the transmit direction, zero-copy operation requires that the application must not modify outgoing data until reception is acknowledged by the peer, but if the application violates this requirement, it will only result in incorrect data payload.

Since elastic threads in IX execute both the network stack and application code, a long running application can block further network processing for a set of flows. This behavior in no way affects other applications or dataplanes. We use a timeout interrupt to detect elastic threads that spend excessive time in user mode (e.g., in excess of 10ms). We mark such applications as non-responsive and notify the control plane.

The current IX prototype does not yet use an IOMMU. As a result, the IX dataplane is trusted code that has access to descriptor rings with host-physical addresses. This limitation does not affect the security model provided to applications.

5 Evaluation

We compared IX to a baseline running the most recent Linux kernel and to mTCP [29]. Our evaluation uses both networking microbenchmarks and a widely deployed, event-based application. In all cases, we use TCP as the networking protocol.

5.1 Experimental Methodology

Our experimental setup consists of a cluster of 24 clients and one server connected by a Quanta/Cumulus 48x10GbE switch with a Broadcom Trident+ ASIC. The client machines are a mix of Xeon E5-2637 @ 3.5 Ghz and Xeon E5-2650 @ 2.6 Ghz. The server is a Xeon E5-2665 @ 2.4 Ghz with 256 GB of DRAM. Each client and server socket has 8 cores and 16 hyperthreads. All machines are configured with Intel x520 10GbE NICs (82599EB chipset). We connect clients to the switch through a single NIC port, while for the server it depends on the experiment. For *10GbE* experiments, we use a single NIC port, and for *4x10GbE* experiments, we use four NIC ports bonded by the switch with a L3+L4 hash.

Our baseline configuration in each machine is an Ubuntu LTS 14.0.4 distribution, updated to the 3.16.1 Linux kernel, the most recent at time of writing. We enable hyperthreading when it improves performance. Except for §5.2, client machines always run Linux. All power management features are disabled for all systems in all experiments. Jumbo frames are never enabled. All

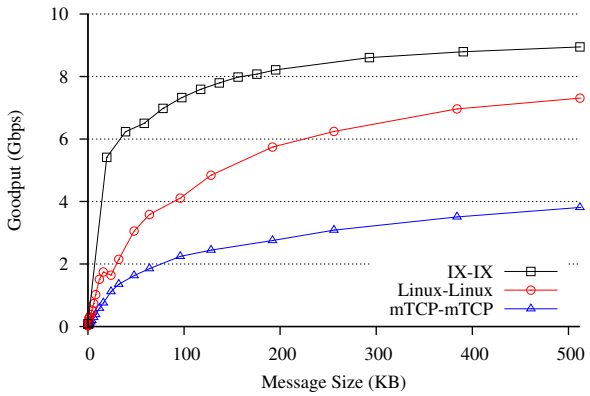


Figure 2: NetPIPE performance for varying message sizes and system software configurations.

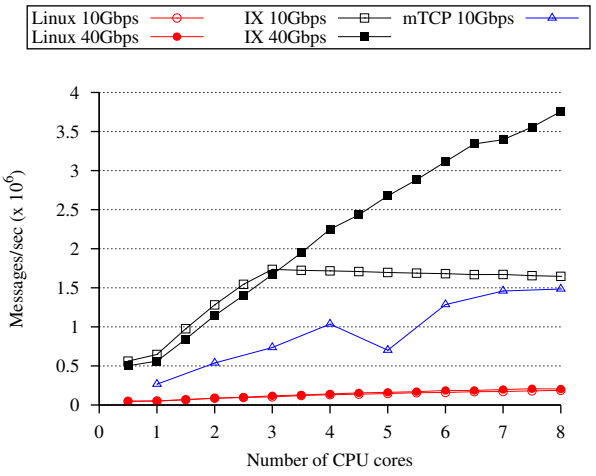
Linux workloads are pinned to hardware threads to avoid scheduling jitter, and background tasks are disabled.

The Linux client and server implementations of our benchmarks use the `libevent` framework with the `epoll` system call. We downloaded and installed mTCP from the public-domain release [30], but had to write the benchmarks ourselves using the mTCP API. We run mTCP with the 2.6.36 Linux kernel, as this is the most recent supported kernel version. We report only 10GbE results for mTCP, as it does not support NIC bonding. For IX, we bound the maximum batch size to $B = 64$ packets per iteration, which maximizes throughput on microbenchmarks (see §6).

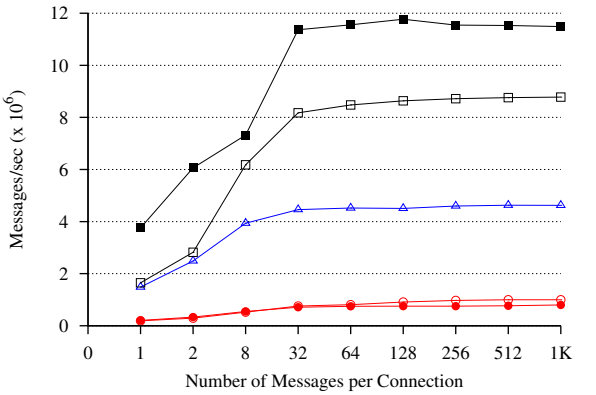
5.2 Latency and Single-flow Bandwidth

We first evaluated the latency of IX using NetPIPE, a popular ping-pong benchmark, using our 10GbE setup. NetPIPE simply exchanges a fixed-size message between two servers and helps calibrate the latency and bandwidth of a single flow [57]. In all cases, we run the same system on both ends (Linux, mTCP, or IX).

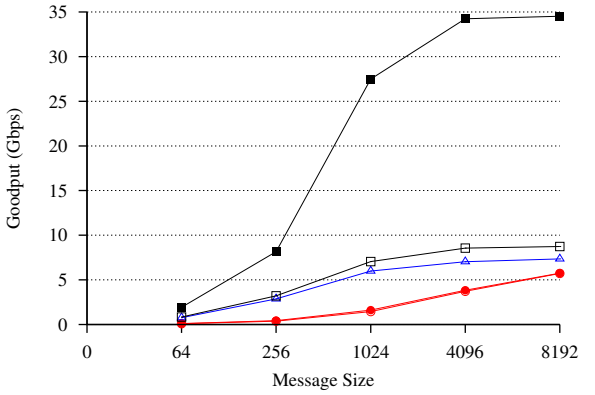
Fig. 2 shows the goodput achieved for different message sizes. Two IX servers have a one-way latency of 5.7µs for 64B messages and achieve goodput of 5Gbps, half of the maximum, with messages as small as 20KB. In contrast, two Linux servers have a one-way latency of 24µs and require 385KB messages to achieve 5Gbps. The differences in system architecture explain the disparity: IX has a dataplane model that polls queues and processes packets to completion whereas Linux has an interrupt model, which wakes up the blocked process. mTCP uses aggressive batching to offset the cost of context switching [29], which comes at the expense of higher latency than both IX and Linux in this particular test.



(a) Multi-core scalability (n=1, s=64B)



(b) n round-trips per connection. (s=64B)



(c) Different message sizes s (n=1)

Figure 3: Multi-core scalability and high connection churn for 10GbE and 4x10GbE setups. In (a), half steps indicate hyperthreads.

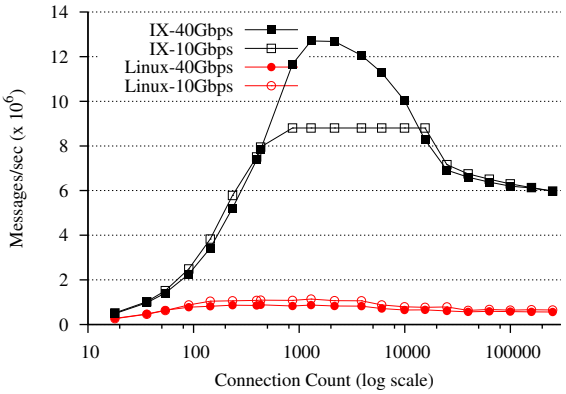


Figure 4: Connection scalability for the 10GbE and 4x10GbE configurations.

5.3 Throughput and Scalability

We evaluate IX's throughput and multi-core scalability with the same benchmark used to evaluate MegaPipe [24] and mTCP [29]. 18 clients connect to a single server listening on a single port, send a remote request of size s bytes, and wait for an echo of a message of the same size. Similar to the NetPIPE benchmark, while receiving the message, the server holds off its echo response until the message has been entirely received. Each client performs this synchronous remote procedure call n times before closing the connection. As in [29], clients close the connection using a reset (`TCP RST`) to avoid exhausting ephemeral ports.

Fig. 3 shows the message rate or goodput for both the 10GbE and the 40GbE configurations as we vary the number of cores used, the number of round-trip messages per connection, and the message size respectively. For the 10GbE configuration, the results for Linux and mTCP are consistent with those published in the mTCP paper [29]. For all three tests (core scaling, message count scaling, message size scaling), IX scales more aggressively than mTCP and Linux. Fig. 3a shows that IX needs only 3 cores to saturate the 10GbE link whereas mTCP requires all 8 cores. On Fig. 3b for 1024 round-trips per connection, IX delivers 8.8 million messages per second, which is 1.9× the throughput of mTCP and of and 8.8× that of Linux. With this packet rate, IX achieves line rate and is limited only by 10GbE bandwidth.

Fig. 3 also shows that IX scales well beyond 10GbE to a 4x10GbE configuration. Fig. 3a shows that IX linearly scales to deliver 3.8 million TCP connections per second on 4x10GbE. Fig. 3b shows a speedup of 2.3× with $n = 1$ and of 1.3× with $n = 1024$ over 10GbE IX. Finally, Fig. 3c shows IX can deliver 8KB messages with a goodput of 34.5 Gbps, for a wire throughput of 37.9 Gbps,

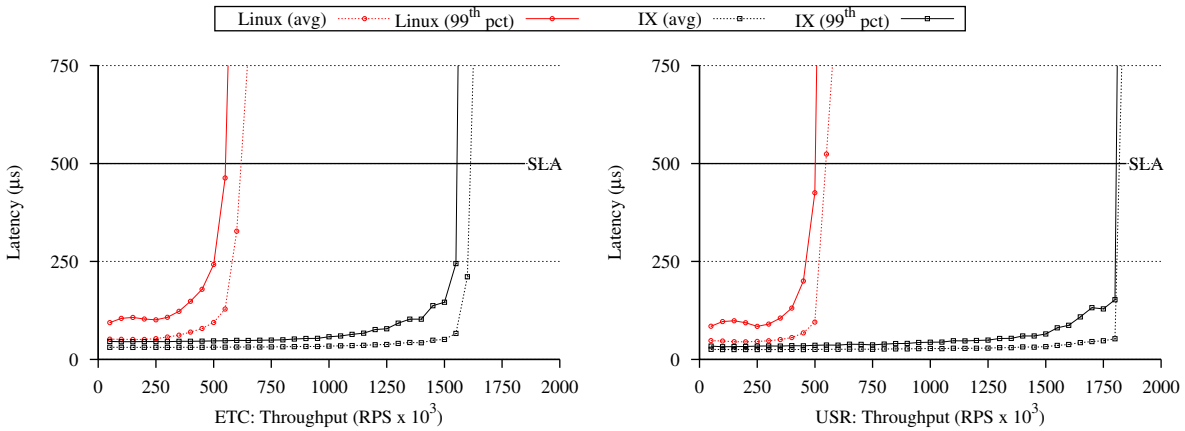


Figure 5: Average and 99th percentile latency as a function of throughput for two memcached workloads.

out of a possible 39.7Gbps. Overall, IX makes it practical to scale protected TCP/IP processing beyond 10GbE, even with a single socket multi-core server.

5.4 Connection Scalability

We also evaluate IX's scalability when handling a large number of concurrent connections on the 4x10GbE setup. 18 client machines runs n threads, with each thread repeatedly performing a 64B remote procedure call to the server with a variable number of active connections. We experimentally set $n = 24$ to maximize throughput. We report the maximal throughput in messages per second for a range of total established connections.

Fig. 4 shows up to 250,000 connections, which is the upper bound we can reach with the available client machines. As expected, throughput increases with the degree of connection concurrency, but then decreases for very large connections counts due to the increasingly high cost of multiplexing among open connections. At the peak, IX performs 10× better than Linux, consistent with the results from Fig. 3b. With 250,000 connections and 4x10GbE, IX is able to deliver 47% of its own peak throughput. We verified that the drop in throughput is not due to an increase in the instruction count, but instead can be attributed to the performance of the memory subsystem. Intel's Data Direct I/O technology, an evolution of DCA [26], eliminates nearly all cache misses associated with DMA transfers when given enough time between polling intervals, resulting in as little as 1.4 L3 cache misses per message for up to 10,000 concurrent connections, a scale where all of IX's data structures fit easily in the L3 cache. In contrast, the workload averages 25 L3 cache misses per message when handling 250,000 concurrent connections. At high connection counts, the working set of this workload is dominated by the TCP connection state and does not fit into the processor's L3 cache. Nevertheless, we believe that further optimizations in the size and access pattern of lwIP's TCP/IP protocol control block structures can substantially reduce this handicap.

5.5 Memcached Performance

Finally, we evaluated the performance benefits of IX with `memcached`, a widely deployed, in-memory, key-value store built on top of the `libevent` framework [42]. It is frequently used as a high-throughput, low-latency caching tier in front of persistent database servers. `memcached` is a network-bound application, with threads spending over 75% of execution time in kernel mode for network processing [36]. It is a difficult application to scale because the common deployments involve high connection counts for `memcached` servers and small-sized requests and replies [2, 46].

We use the `mutilate` load-generator to place a selected load on the server in terms of requests per second (RPS) and measure response latency [35]. `mutilate` coordinates a large number of client threads across multiple machines to generate the desired RPS load, while a separate unloaded client measures latency by issuing one request at the time. We configure `mutilate` to generate load representative of two workloads from Facebook [2]: the ETC workload that represents that highest capacity deployment in Facebook, has 20B–70B keys, 1B–1KB values, and 75% GET requests; and the USR workload that represents deployment with most GET requests in Facebook, has short keys (<20B), 2B values, and 99% GET requests. In USR, almost all traffic involves minimum-sized TCP packets. Each request is issued separately (no `multiget` operations). However, clients are permitted to pipeline up to four requests per connection if needed to keep up with their target request rate. We use 23 client machines to generate load for a total of 1,476 connections to the `memcached` server.

To provide insights into the full range of system behaviors, we report average and 99th percentile latency as a function of the achieved throughput. The 99th percentile latency captures tail latency issues and is the most relevant metric for datacenter applications [14]. Most commercial `memcached` deployments provision each server so that the 99th percentile latency does not exceed 200µs to 500µs. We carefully tune the Linux baseline setup according to the guidelines in [36]: we pin memcached threads, configure interrupt-distribution based on thread-affinity, and tune interrupt moderation thresholds. We believe that our baseline Linux numbers are as tuned as possible for this hardware using the open-source version of `memcached-1.4.18`. We report the results for the server configuration that provides the best performance: 8 cores with Linux, but only 6 with IX.

Porting `memcached` to IX primarily consisted of adapting it to use our event library. In most cases, the port was straightforward, replacing Linux and `libevent` function calls with their equivalent versions in our API. We did yet not attempt to tune the internal scalability of `memcached` [20] or to support zero-copy I/O operations.

Fig. 5 shows the throughput-latency curves for the two `memcached` workloads for Linux and IX, while Table 2 reports the unloaded, round-trip latencies and maximum request rate that meets a service-level agreement, both measured at the 99th percentile. IX noticeably reduces the unloaded latencies to roughly half. Note that we use Linux clients for these experiments; running IX on clients should further reduce latency.

At high request rates, the distribution of CPU time shifts from being $\sim$ 75% in the Linux kernel to $<$ 10% in the IX dataplane kernel. This allows IX to increase throughput by 2.8× and 3.6× for `ETC` and `USR` respectively at a 500µs tail latency SLA. The improvement for `ETC` is lower due to the increased lock contention within the application itself, in particular because it has a higher write frequency. Lock contention within application code is also the reason that IX cannot provide throughput improvements with more than 6 cores.

Configuration	Minimum latency @99th pct	RPS for SLA: $<$ 500µs @99th pct
ETC-Linux	94µs	550K
ETC-IX	45µs	1550K
USR-Linux	85µs	500K
USR-IX	32µs	1800K

Table 2: Unloaded latency and maximum RPS for a given service-level agreement for the memcache workloads `ETC` and `USR`.

6 Discussion

What makes IX fast: The results in §5 show that a networking stack can be implemented in a protected OS kernel and still deliver wire-rate performance for most benchmarks. The tight coupling of the dataplane architecture, using only a minimal amount of batching to amortize transition costs, causes application logic to be scheduled at the right time, which is essential for latency-sensitive workloads. Therefore, the benefits of IX go beyond just minimizing kernel overheads. The lack of intermediate buffers allows for efficient, application-specific implementations of I/O abstractions such the `libix` event library. The zero-copy approach helps even when the user-level libraries add a level of copying, as it is the case for the `libevent` compatible interfaces in `libix`. The extra copy occurs much closer to the actual use, thereby increasing cache locality. Finally, we carefully tuned IX for multi-core scalability, eliminating constructs that introduce synchronization or coherence traffic.

The IX dataplane optimizations — run to completion, adaptive batching, and a zero-copy API — can also be implemented in a user-level networking stack in order to get similar benefits in terms of throughput and latency. While a user-level implementation would eliminate protection domain crossings, it would not lead to significant performance improvements over IX. Protection domain crossings inside VMX non-root mode add only a small amount of extra overhead, on the order of a single L3 cache miss [7]. Moreover, these overheads are quickly amortized at higher packet rates.

Subtleties of adaptive batching: Batching is commonly understood to trade off higher latency at low loads for better throughput at high loads. IX uses adaptive, bounded batching to actually improve on both metrics. Fig. 6 compares the latency vs. throughput on the `USR` `memcached` workload of Fig. 5 for different upper bounds B to the batch size. At low load, B does not impact tail latency, as adaptive batching does not delay processing of pending packets. At higher load, larger values of B improve throughput, by 29% between $B = 1$ to $B = 16$. For this workload, $B \geq 16$ maximizes throughput.

While tuning IX performance, we ran into an unexpected hardware limitation that was triggered at high packet rates with small average batch sizes (i.e. before the dataplane was saturated): the high rate of PCIe writes required to post fresh descriptors at every iteration led to performance degradation as we scaled the number of cores. To avoid this bottleneck, we simply coalesced PCIe writes on the receive path so that we replenished at least 32 descriptor entries at a time. Luckily, we did not have to

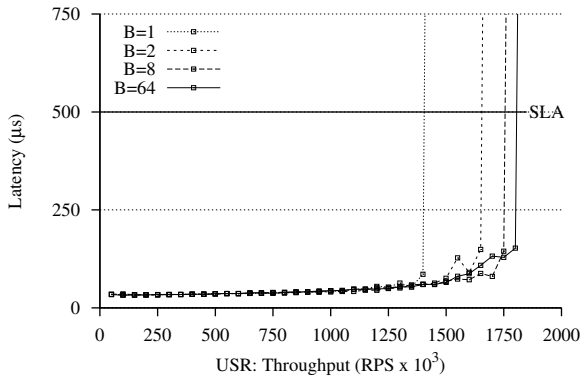


Figure 6: 99th percentile latency as a function of throughput for `USR` workload from Fig. 5, for different values of the batch bound B.

coalesce PCIe writes on the transmit path, as that would have impacted latency.

Limitations of current prototype: The current IX implementation does not yet exploit IOMMUs or VT-d. Instead, it maps descriptor rings directly into IX memory, using the Linux pagemap interface to determine physical addresses. Although this choice puts some level of trust into the IX dataplane, application code remains securely isolated. In the future, we plan on using IOMMU support to further isolate IX dataplanes. We anticipate overhead will be low because of our use of large pages. Also, the IX prototype currently does not take advantage of the NIC's SR-IOV capabilities, but instead allocates entire physical devices to dataplanes.

We also plan to add support for interrupts to the IX dataplanes. The IX execution model assumes some cooperation from application code running in elastic threads. Specifically, applications should handle events in a quick, non-blocking manner; operations with extended execution times are expected to be delegated to background threads rather than execute within the context of elastic threads. The IX dataplane is designed around polling, with the provision that interrupts can be configured as a fallback optimization to refresh receive descriptor rings when they are nearly full and to refill transmit descriptor rings when they are empty (steps (1) and (6) in Fig 1b). Occasional timer interrupts are also required to ensure full TCP compliance in the event an elastic thread blocks for an extended period.

Future work: This paper focused primarily on the IX dataplane architecture. IX is designed and implemented to support the dynamic addition and removal of elastic threads in order to achieve energy proportional and resource efficient computing. So far we have tested only static configurations. In future work, we will explore control plane issues, including a dynamic runtime that rebalances network flows between available elastic threads in a manner that maintains both throughput and latency constraints.

We will also explore the synergies between IX and networking protocols designed to support microsecond-level latencies and the reduced buffering characteristics of IX deployments, such as DCTCP [1] and ECN [52]. Note that the IX dataplane is not specific to TCP/IP. The same design principles can benefit alternative, potentially application specific, network protocols, as well as high-performance protocols for non-volatile memory access. Finally, we will investigate library support for alternative APIs on top of our low-level interface, such as MegaPipe [24], cooperative threading [65], and rule-based models [60]. Such APIs and programming models will make it easier for applications to benefit from the performance and scalability advantages of IX.

7 Related Work

We organize the discussion topically, while avoiding redundancy with the commentary in §2.3.

Hardware virtualization: Hardware support for virtualization naturally separates control and execution functions, e.g., to build type-2 hypervisors [10, 33], run virtual appliances [55], or provide processes with access to privileged instructions [7]. Similar to IX, Arrakis uses hardware virtualization to separate the I/O dataplane from the control plane [50]. IX differs in that it uses a full Linux kernel as the control plane; provides three-way isolation between the control plane, networking stack, and application; and proposes a dataplane architecture that optimizes for both high throughput and low latency. On the other hand, Arrakis uses Barrelfish as the control plane [6] and includes support for IOMMUs and SR-IOV.

Library operating systems: Exokernels extend the end-to-end principle to resource management by implementing system abstractions via library operating systems linked in with applications [19]. Library operating systems often run as virtual machines [9] used, for instance, to deploy cloud services [39]. IX limits itself to the implementation of the networking stack, allowing applications to implement their own resource management policies, e.g. via the `libevent` compatibility layer.

Asynchronous and zero-copy communication: Systems with asynchronous, batched, or exception-less system calls substantially reduce the overheads associated with frequent kernel transitions and context switches [24, 29, 53, 58]. IX's use of adaptive batching shares similar

benefits but is also suitable for low-latency communication. Zero-copy reduces data movement overheads and simplifies resource management [48]. POSIX OSes have been modified to support zero-copy through page remapping and copy-on-write [11]. By contrast, IX's cooperative memory management enables zero-copy without page remapping. Similar to IX, TinyOS passes pointers to packet buffers between the network stack and the application in a cooperative, zero-copy fashion [37]. However, IX is optimized for datacenter workloads, while TinyOS focuses on memory constrained, sensor environments.

8 Conclusion

We described IX, a dataplane operating system that leverages hardware virtualization to separate and isolate the Linux control plane, the IX dataplane instances that implement in-kernel network processing, and the network-bound applications running on top of them. The IX dataplane provides a native, zero-copy API that explicitly exposes flow control to applications. The dataplane architecture optimizes for both bandwidth and latency by processing bounded batches of packets to completion and by eliminating synchronization on multi-core servers. On microbenchmarks, IX noticeably outperforms both Linux and mTCP in terms of both latency and throughput, scales to hundreds of thousands of active concurrent connections, and can saturate 4x10GbE configurations using a single processor socket. Finally, we show that porting `memcached` to IX removes kernel bottlenecks and improves throughput by up to 3.6×, while reducing tail latency by more than 2×.

Acknowledgements

The authors would like to thank David Mazières for his many insights into the system and his detailed feedback on the paper. We also thank Katerina Argyraki, James Larus, Jacob Leverich, Philip Levis, Willy Zwaenepoel, the anonymous reviewers, and our shepherd Andrew Warfield for their comments. This work was supported by a Google research grant, the Stanford Experimental Datacenter Lab, and the Microsoft-EPFL Joint Research Center. Adam Belay is supported by a VMware Graduate Fellowship.

References

[1] M. Alizadeh, A. G. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data Center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM 2010 Conference*, pages 63–74, 2010.

[2] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload Analysis of a Large-Scale Key-Value Store. In *Proceedings of the 2012 ACM SIGMETRICS International conference on Measurement and modeling of computer systems*, pages 53–64, 2012.

[3] L. A. Barroso. Three things that must be done to save the data center of the future (ISSCC 2014 Keynote). `http://www.theregister.co.uk/Print/2014/02/11/google_research_three_things_that_must_be_done_to_save_the_data_center_of_the_future/`, 2014.

[4] L. A. Barroso and U. Hölzle. The Case for Energy-Proportional Computing. *IEEE Computer*, 40(12):33–37, 2007.

[5] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift. Efficient Virtual Memory for Big Memory Servers. In *Proceedings of the 40th International Symposium on Computer Architecture (ISCA '13)*, pages 237–248, 2013.

[6] A. Baumann, P. Barham, P.-É. Dagand, T. L. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The Multikernel: A new OS architecture for scalable multicore systems. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, pages 29–44, 2009.

[7] A. Belay, A. Bittau, A. J. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis. Dune: Safe User-level Access to Privileged CPU Features. In *Proceedings of the 10th Symposium on Operating System Design and Implementation (OSDI '12)*, pages 335–348, 2012.

[8] S. M. Bellovin. A Look Back at "Security Problems in the TCP/IP Protocol Suite". In *Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC '04)*, pages 229–249, 2004.

[9] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. *ACM Trans. Comput. Syst.*, 15(4):412–447, 1997.

[10] E. Bugnion, S. Devine, M. Rosenblum, J. Sugerman, and E. Y. Wang. Bringing Virtualization to the x86 Architecture with the Original VMware Workstation. *ACM Trans. Comput. Syst.*, 30(4):12, 2012.

[11] H.-K. J. Chu. Zero-Copy TCP in Solaris. In *Proceedings of the 1996 USENIX Annual Technical Conference*, pages 253–264, 1996.

[12] A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler. The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, pages 1–17, 2013.

[13] T. David, R. Guerraoui, and V. Trigonakis. Everything You Always Wanted to Know About Synchronization but Were Afraid to Ask. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, pages 33–48, 2013.

[14] J. Dean and L. A. Barroso. The Tail at Scale. *Commun. ACM*, 56(2):74–80, 2013.

[15] C. Delimitrou and C. Kozyrakis. Quasar: Resource-Efficient and QoS-Aware Cluster Management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*, pages 127–144, 2014.

[16] M. Dobrescu, N. Egi, K. J. Argyraki, B.-G. Chun, K. R. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, pages 15–28, 2009.

[17] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. FaRM: Fast Remote Memory. In *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI '14)*, 2014.

[18] A. Dunkels. Design and Implementation of the lwIP TCP/IP Stack. *Swedish Institute of Computer Science*, 2:77, 2001.

[19] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 251–266, 1995.

[20] B. Fan, D. G. Andersen, and M. Kaminsky. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *Proceedings of the 10th Symposium on Networked Systems Design and Implementation (NSDI '13)*, pages 371–384, 2013.

[21] M. Fisk and W. Feng. Dynamic Adjustment of TCP Window Sizes. Technical report, Tech. Rep. Los Alamos Unclassified Report (LAUR) 00-3221, Los Alamos National Laboratory, 2000.

[22] S. Floyd and V. Jacobson. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Trans. Netw.*, 1(4):397–413, 1993.

[23] R. Graham. The C10M Problem. `http://c10m.robertgraham.com`, 2013.

[24] S. Han, S. Marshall, B.-G. Chun, and S. Ratnasamy. MegaPipe: A New Programming Interface for Scalable Network I/O. In *Proceedings of the 10th Symposium on Operating System Design and Implementation (OSDI '12)*, pages 135–148, 2012.

[25] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, pages 22–22, 2011.

[26] R. Huggahalli, R. R. Iyer, and S. Tetrick. Direct Cache Access for High Bandwidth Network I/O. In *Proceedings of the 32st International Symposium on Computer Architecture (ISCA '05)*, pages 50–59, 2005.

[27] Intel Corp. Open Source Kernel Enhancements for Low Latency Sockets using Busy Poll. `http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/open-source-kernel-enhancements-paper.pdf`, 2013.

[28] Intel Corp. Intel DPDK: Data Plane Development Kit. `http://dpdk.org/`, 2014.

[29] E. Jeong, S. Woo, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mTCP: A Highly Scalable User-level TCP Stack for Multicore Systems. In *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI '14)*, 2014.

[30] E. Jeong, S. Woo, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. o. Park. mTCP source code release, v. of 2014-02-26. `https://github.com/eunyoung14/mtcp`, 2014.

[31] J. Jose, H. Subramoni, M. Luo, M. Zhang, J. Huang, M. W. ur Rahman, N. S. Islam, X. Ouyang, H. Wang, S. Sur, and D. K. Panda. Memcached Design on High Performance RDMA Capable Interconnects. In *International Conference on Parallel Processing (ICPP '11)*, pages 743–752, 2011.

[32] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat. Chronos: Predictable Low Latency for Data Center Applications. In *ACM Symposium on Cloud Computing (SOCC '12)*, page 9, 2012.

[33] A. Kivity. KVM: The Linux Virtual Machine Monitor. In *Proceedings of the 2007 Ottawa Linux Symposium (OLS)*, pages 225–230, July 2007.

[34] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *ACM Trans. Comput. Syst.*, 18(3):263–297, 2000.

[35] J. Leverich. Mutilate: High-Performance Memcached Load Generator. `https://github.com/leverich/mutilate`, 2014.

[36] J. Leverich and C. Kozyrakis. Reconciling High Server Utilization and Sub-Millisecond Quality-of-Service. In *Proceedings of the 9th EuroSys Conference (Eurosys '14)*, page 4, 2014.

[37] P. Levis, S. Madden, D. Gay, J. Polastre, R. Szewczyk, A. Woo, E. A. Brewer, and D. E. Culler. The Emergence of Networking Abstractions and Techniques in TinyOS. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation (NSDI '04)*, pages 1–14, 2004.

[38] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso, and C. Kozyrakis. Towards Energy Proportionality for Large-Scale Latency-Critical Workloads. In *Proceedings of the 41st International Symposium on Computer Architecture (ISCA '14)*, pages 301–312, 2014.

[39] A. Madhavapeddy, R. Mortier, C. Rotsos, D. J. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. Unikernels: Library Operating Systems for the Cloud. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*, pages 461–472, 2013.

[40] I. Marinos, R. N. M. Watson, and M. Handley. Network Stack Specialization for Performance. In *Proceedings of the ACM SIGCOMM 2014 Conference*, pages 175–186, 2014.

[41] P. E. McKenney and J. D. Slingwine. Read-Copy Update: Using Execution History to Solve Concurrency Problems. In *Parallel and Distributed Computing and Systems*, pages 509–518, 1998.

[42] memcached – a distributed memory object caching system. `http://memcached.org`, 2014.

[43] Microsoft Corp. Receive Side Scaling. `http://msdn.microsoft.com/library/windows/hardware/ff556942.aspx`, 2014.

[44] C. Mitchell, Y. Geng, and J. Li. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *Proceedings of the 2013 USENIX Annual Technical Conference*, pages 103–114, 2013.

[45] J. C. Mogul and K. K. Ramakrishnan. Eliminating Receive Livelock in an Interrupt-Driven Kernel. *ACM Trans. Comput. Syst.*, 15(3):217–252, 1997.

[46] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling Memcache at Facebook. In *Proceedings of the 10th Symposium on Networked Systems Design and Implementation (NSDI '13)*, pages 385–398, 2013.

[47] D. Ongaro, S. M. Rumble, R. Stutsman, J. K. Ousterhout, and M. Rosenblum. Fast Crash Recovery in RAMCloud. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, pages 29–41, 2011.

[48] V. S. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: A Unified I/O Buffering and Caching System. *ACM Trans. Comput. Syst.*, 18(1):37–66, 2000.

[49] A. Pesterev, J. Strauss, N. Zeldovich, and R. T. Morris. Improving Network Connection Locality on Multicore Systems. In *Proceedings of the 7th EuroSys Conference (Eurosys '12)*, pages 337–350, 2012.

[50] S. Peter, J. Li, I. Zhang, D. R. K. Ports, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The Operating System is the Control Plane. In *Proceedings of the 11th Symposium on Operating System Design and Implementation (OSDI '14)*, 2014.

[51] N. Provos and N. Mathewson. libevent: an event notification library. `http://libevent.org`, 2003.

[52] K. Ramakrishnan, S. Floyd, D. Black, et al. The Addition of Explicit Congestion Notification (ECN) to IP. IETF RFC 3168, 2001.

[53] L. Rizzo. Revisiting Network I/O APIs: The netmap Framework. *Commun. ACM*, 55(3):45–51, 2012.

[54] S. M. Rumble, D. Ongaro, R. Stutsman, M. Rosenblum, and J. K. Ousterhout. It's Time for Low Latency. In *Proceedings of the 13th Workshop on Hot Topics in Operating Systems (HotOS-XIII)*, 2011.

[55] C. P. Sapuntzakis, D. Brumley, R. Chandra, N. Zeldovich, J. Chow, M. S. Lam, and M. Rosenblum. Virtual Appliances for Deploying and Maintaining Software. In *Proceedings of the 17th Conference on Systems Administration (LISA '03)*, pages 181–194, 2003.

[56] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: Flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th EuroSys Conference (EuroSys '13)*, pages 351–364, 2013.

[57] Q. O. Snell, A. R. Mikler, and J. L. Gustafson. Netpipe: A Network Protocol Independent Performance Evaluator. In *IASTED International Conference on Intelligent Information Management and Systems*, volume 6, 1996.

[58] L. Soares and M. Stumm. FlexSC: Flexible System Call Scheduling with Exception-Less System Calls. In *Proceedings of the 9th Symposium on Operating System Design and Implementation (OSDI '10)*, pages 33–46, 2010.

[59] Solarflare Communications. Introduction to OpenOnload: Building Application Transparency and Protocol Conformance into Application Acceleration Middleware. `http://www.solarflare.com/content/userfiles/documents/solarflare_openonload_intropaper.pdf`, 2011.

[60] R. Stutsman and J. K. Ousterhout. Toward Common Patterns for Distributed, Concurrent, Fault-Tolerant Code. In *Proceedings of the 14th Workshop on Hot Topics in Operating Systems (HotOS-XIV)*, 2013.

[61] C. A. Thekkath, T. D. Nguyen, E. Moy, and E. D. Lazowska. Implementing Network Protocols at User Level. In *Proceedings of the ACM SIGCOMM 1993 Conference*, pages 64–73, 1993.

[62] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. M. Martins, A. V. Anderson, S. M. Bennett, A. Kägi, F. H. Leung, and L. Smith. Intel Virtualization Technology. *IEEE Computer*, 38(5):48–56, 2005.

[63] G. Varghese and A. Lauck. Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP '87)*, pages 25–38, 1987.

[64] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D. G. Andersen, G. R. Ganger, G. A. Gibson, and B. Mueller. Safe and Effective Fine-Grained TCP Retransmissions for Datacenter Communication. In *Proceedings of the ACM SIGCOMM 2009 Conference*, pages 303–314, 2009.

[65] J. R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. A. Brewer. Capriccio: Scalable Threads for Internet Services. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 268–281, 2003.

[66] WhatsApp, Inc. 1 million is so 2011. `https://blog.whatsapp.com/index.php/2012/01/1-million-is-so-2011`, 2012.

[67] D. A. Wheeler. SLOCCount, v2.26. `http://www.dwheeler.com/sloccount/`, 2001.