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-

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

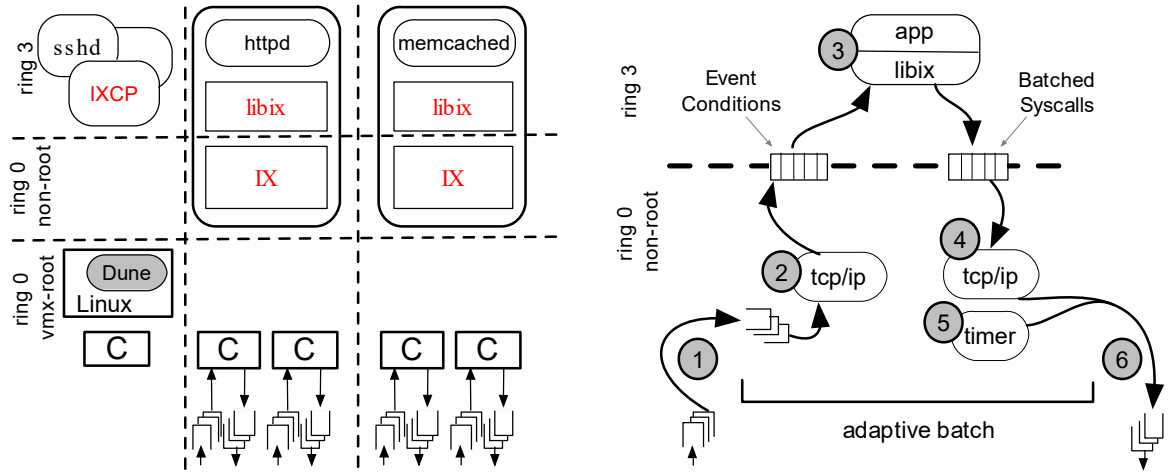
4.1 Overview

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-