The IX Operating System: Combining Low Latency, High Throughput, and Efficiency in a Protected Dataplane

ADAM BELAY, Stanford
GEORGE PREKAS and MIA PRIMORAC, EPFL
ANA KLIMOVIC, SAMUEL GROSSMAN, and CHRISTOS KOZYRAKIS, Stanford
EDOUARD BUGNION, EPFL

The conventional wisdom is that aggressive networking requirements, such as high packet rates for small messages and μs-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 and high resource efficiency while maintaining the protection and isolation benefits of 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 eliminating coherence traffic and multicore synchronization. The control plane dynamically adjusts core allocations and voltage/frequency settings to meet service-level objectives.

We demonstrate that IX outperforms Linux and a user-space network stack significantly in both throughput and end-to-end latency. Moreover, IX improves the throughput of a widely deployed, key-value store by up to 6.4× and reduces tail latency by more than 2×. With three varying load patterns, the control plane saves 46%–54% of processor energy, and it allows background jobs to run at 35%–47% of their standalone throughput.

Categories and Subject Descriptors: D.4.1 [**Operating Systems**]: Process Management; D.4.4 [**Operating Systems**]: Communications Management; D.4.7 [**Operating Systems**]: Organization and Design

General Terms: Design, Performance

Additional Key Words and Phrases: Virtualization, dataplane operating systems, latency-critical applications, microsecond-scale computing, energy-proportionality, workload consolidation

ACM Reference Format:
Adam Belay, George Prekas, Mia Primorac, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2016. The IX operating system: Combining low latency, high throughput, and efficiency in a protected dataplane. ACM Trans. Comput. Syst. 34, 4, Article 11 (December 2016), 39 pages.
DOI: http://dx.doi.org/10.1145/2997641

This work was funded by DARPA CRASH (under contract #N66001-10-2-4088), a Google research grant, the Stanford Experimental Datacenter Lab, the Microsoft-EPFL Joint Research Center, and NSF grant CNS-1422088. George Prekas was supported by a Google Graduate Research Fellowship and Adam Belay by a VMware Graduate Fellowship.
Authors' addresses: A. Belay, A. Klimovic, S. Grossman, and C. Kozyrakis, Computer Science Department, Stanford University, 353 Serra Mall, Stanford, CA, 94305; emails: {abelay, anakli, samuelgr, kozyraki}@stanford.edu; G. Prekas, M. Primorac, and E. Bugnion, School of Computer and Communication Sciences, EPFL, INN 237 (Bâtiment INN), Station 14, CH-1015 Lausanne, Switzerland; emails: {george.prekas, mia.primorac, edouard.bugnion}@epfl.ch.

DOI: http://dx.doi.org/10.1145/2997641

1. INTRODUCTION

Datacenter applications have evolved with the advent of web-scale services. User-facing, large-scale applications such as search, social networking, and e-commerce now rely extensively on high fan-out patterns between low-latency services. Such services exhibit low per-request service times (a handful of μs for a key-value store), have strict service-level objectives (SLOs, e.g. $<500\mu$s at the 99th percentile), and must sustain massive request rates for short messages with high client fan-in connection counts and churn [Atikoglu et al. 2012; Dean and Barroso 2013; Nishtala et al. 2013].

The conventional wisdom is that there is a basic mismatch between these requirements and existing networking stacks in commodity operating systems. To address the performance concern, some systems bypass the kernel and implement the networking stack in user-space [Jeong et al. 2014b; Kapoor et al. 2012; Marinos et al. 2014; Solarflare Communications 2011; Thekkath et al. 1993]. While kernel bypass eliminates privilege-level crossing overheads, on its own it does not eliminate the difficult tradeoffs between high packet rates and low latency (see Section 5.2.1). 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 [Dragojevic et al. 2014; Jose et al. 2011; Mitchell et al. 2013; Ousterhout et al. 2015]. However, such adapters must be present at both ends of the connection and can only be used within the datacenter.

Such latency-critical services are also challenging to run in a shared infrastructure environment. They are particularly sensitive to resource allocation and frequency settings, and they suffer frequent tail latency violations when common power management or consolidation approaches are used [Leverich and Kozyrakis 2014; Li et al. 2014]. As a result, operators typically deploy them on dedicated servers running in polling mode, forgoing opportunities for workload consolidation and reduced power consumption at below-peak utilization levels. Since these services are deployed on thousands of servers in large-scale datacenters, this deployment practice represents a huge waster in resource use.

Ideally, we want these services to achieve *energy proportionality*, so that their energy consumption scales with observed load [Barroso and Hölzle 2007; Lo et al. 2014]. Hardware enhancements, primarily in dynamic voltage/frequency scaling (DVFS) and idle modes in modern processors [Kim et al. 2008; Rotem et al. 2012], provide a foundation for energy proportionality. Moreover, we want these services to allow for *workload consolidation*, so that any spare resources during periods of low load can be used by workloads such as background analytics in order to raise server utilization [Vogels 2008; Verma et al. 2015]. The two goals map to distinct economic objectives: energy proportionality reduces operational expenses (*opex*), whereas workload consolidation reduces capital expenses (*capex*). Since capital costs often dominate the datacenter's total cost of ownership (*TCO*), consolidation is highly desirable. Nevertheless, it is not always possible, for example, when one application consumes the entirety of a given resource (e.g., memory). In such cases, energy proportionality is a necessity.

We propose IX, an operating system designed to break the four-way tradeoff between high throughput, low latency, strong protection, and resource efficiency. Its architecture builds upon lessons from high-performance middleboxes, such as firewalls, load balancers, and software routers [Dobrescu et al. 2009; Kohler et al. 2000]. IX separates the control plane, which is responsible for system configuration and coarse-grained 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 [Belay et al. 2012]. In our implementation, the

control plane leverages mechanisms of the full Linux kernel to dynamically reallocate resources, 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 associated 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 multicore 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 [Clements et al. 2013]. However, the `libix` user-level library includes an event-based API similar to the popular `libevent` library [Provos and Mathewson 2003], providing compatibility with a wide range of existing applications.

The core of the IX control plane is a dynamic controller that adjusts the number of cores allocated to a latency-sensitive application running on top of IX and the DVFS settings for these cores. The remaining cores can be placed in idle modes to reduce power consumption or can be safely used to run background tasks. The controller builds on two key mechanisms. The first mechanism detects backlog and increases in queuing delays that exceed the allowable upper bound for the specific latency-critical application. It monitors CPU utilization and signals required adjustments in resource allocation. The second mechanism, implemented in coordination with the dataplane, quickly migrates both network and application processing between cores transparently and without dropping or reordering packets.

To evaluate the dataplane, we compare IX with a TCP/IP dataplane against Linux 4.2 and mTCP, a state-of-the-art user-level TCP/IP stack [Jeong et al. 2014b]. On a 10GbE experiment using short messages, IX outperforms Linux and mTCP by up to 6.6× and 1.8×, respectively, for throughput. IX further scales to a 4x10GbE configuration using a single multicore socket. The unloaded uni-directional latency for two IX servers is 5.8μs, which is 3.3× better than 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 6.4× in terms of throughput at a given 99th percentile latency bound, as it can reduce kernel time, due essentially to network processing, from ∼80% with Linux to 60% with IX.

Before evaluating the control plane, we performed an exhaustive analysis of static configurations for a latency-critical service (`memcached` [memcached 2014]) running on a modern server to gain a principled understanding of the challenges for resource management in the presence of latency-critical services. We explored up to 224 possible settings for core allocation, use of hyperthreads, DVFS frequencies, and Turbo Boost. While our experiments use a single application, the implications have broad applicability because `memcached` has aggressive latency requirements, short service times, and a large number of independent clients that are common among many latency-critical applications. Our experiments reveal that there is an inherent tradeoff for any given static configuration between the maximum throughput and the overall efficiency when operating below peak load. Furthermore, the experiments reveal a Pareto-optimal frontier in the efficiency of static configurations at any given load level, which allows for close to linear improvements in energy proportionality and workload consolidation factors.

We then evaluated our control plane with two control policies that optimize for energy proportionality and workload consolidation, respectively. A policy determines how resources (cores, hyperthreads, and DVFS settings) are adjusted to reduce underutilization or to restore violated SLOs. The two policies are derived from the exhaustive analysis of the 224 static configurations. For the platform studied (a Xeon E5-2665), we conclude that for *best energy proportionality*, (1) we start with the lowest clock rate and allocate additional cores to the latency-critical task as its load grows, using at first only one hyperthread per core; (2) we enable the second hyperthread only when all cores are in use; and finally (3) we increase the clock rate for the cores running the latency-critical task. For *best consolidation*, we (1) start at the nominal clock rate and add cores with both hyperthreads enabled as load increases and (2) finally enable Turbo Boost as a last resort.

IX demonstrates that, by revisiting networking APIs and taking advantage of modern NICs and multicore chips, we can design systems that achieve high throughput, low latency, 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. Finally, we also demonstrate that latency-sensitive applications can be deployed efficiently through dynamic resource allocation policies that target a specific tail latency.

This article contains the research contributions of two conference papers that focus on the dataplane [Belay et al. 2014] and the control plane [Prekas et al. 2015], respectively. The evaluation results presented in this article have been reproduced with IX v.1.0, which is available in open source [IX on GitHub 2016]. A corresponding technical report provides detailed instructions to reproduce all the results of this article [Prekas et al. 2016].

The rest of the article is organized as follows. Section 2 motivates the need for a new OS architecture. Sections 3 and 4 present the design principles and implementation of IX. Section 5 presents the quantitative evaluation. Sections 6 and 7 discuss open issues and related work. We conclude in Section 8.

2. BACKGROUND AND MOTIVATION

Our work focuses on improving operating systems for applications with aggressive networking requirements running on multicore 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 [Barroso 2014; Rumble et al. 2011]. 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 [Dean and Barroso 2013]. Although tail tolerance is actually an end-to-end challenge, the system software stack plays a significant role in exacerbating the problem [Leverich and Kozyrakis 2014]. Overall, each service node must ideally provide tight bounds on the 99th percentile request latency.

High packet rates. The requests and, oftentimes, the replies between the various services that compose a datacenter application are quite small. In Facebook's `memcached` service, for example, the vast majority of requests use keys shorter than 50 bytes and

involve values shorter than 500 bytes [Atikoglu et al. 2012], and each node can scale to serve millions of requests per second [Nishtala et al. 2013].

The high packet rate must also be sustainable under a large number of concurrent connections and high connection churn [Graham 2013]. 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 [Nishtala et al. 2013].

Protections. Since multiple services commonly share servers in both public and private datacenters [Dean and Barroso 2013; Hindman et al. 2011; Schwarzkopf et al. 2013], there is a 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. Unfortunately, classic operating system schedulers are ill-matched to ensure tail control [Leverich and Kozyrakis 2014; Li et al. 2014]. Novel dynamic resource management mechanisms and policies are required to improve energy proportionality and workload consolidation in the presence of latency-sensitive applications [Lo et al. 2014, 2015; Li et al. 2016].

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, 10GbE NICs and switches are widely deployed in datacenters, with 40GbE and 100GbE technologies right around the corner. The combination of tens of hardware threads and 10GbE NICs should allow for rates of 15M packets/sec with minimum-sized packets. We should also achieve 10 to 20μs round-trip latencies given 3μs latency across a pair of 10GbE 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 are based on an assumption of multiple applications sharing a single processing core and packet interarrival 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-grained 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-grained scheduling of applications to cores increase 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 [Solarflare Communications 2011], mTCP [Jeong et al. 2014b], and Sandstorm [Marinos et al. 2014] 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 Section 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 [Bellovin 2004]. 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 [Dragojevic et al. 2014; Jose et al. 2011; Mitchell et al. 2013; Ousterhout et al. 2015]. 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 [Nishtala et al. 2013]. 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 [Han et al. 2012]. 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 multithreaded applications to accept incoming connections in parallel. Affinity-accept reduces overheads by ensuring that all processing for a network flow is affinitized to the same core [Pesterev et al. 2012]. Recent Linux kernels support a busy polling driver mode that trades increased CPU utilization for reduced latency [Intel Corp. 2013], but it is not yet compatible with `epoll`. When microsecond latencies are irrelevant, properly tuned stacks can maintain millions of open connections [WhatsApp, Inc. 2012].

3. IX DESIGN APPROACH

The first two requirements in Section 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 [Dobrescu et al. 2009; Kohler et al. 2000] by integrating the networking 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 OSs. 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 OSs 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 Section 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 dataplane 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 [Engler et al. 1995], 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 [Belay et al. 2012]. Dataplanes have capabilities similar to guest OSs 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 [Mogul and Ramakrishnan 1997], 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 [Han et al. 2012; Soares and Stumm 2010] and at the network API and hardware queue level [Jeong et al. 2014b]. 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: (1) we never wait to batch requests and batching only occurs in the presence of congestion, and (2) 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 [Ramakrishnan et al. 2001]), and make policy decisions for congestion management (e.g., via RED [Floyd and Jacobson 1993]).

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 multiqueue NICs with receive-side scaling (RSS [Microsoft Corp. 2014]) 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. 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 multithreaded applications [Clements et al. 2013]. 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.

TCP-friendly flow group migration. The IX control plane establishes dynamically the mapping of RSS flow groups to queues to balance the traffic among the hardware threads. The IX dataplane implements the actual flow group migration and programs the NIC's RSS Redirection Table [Intel Corp. 2014a] to change the mappings. The implementation does not impact the steady-state performance of the dataplane and

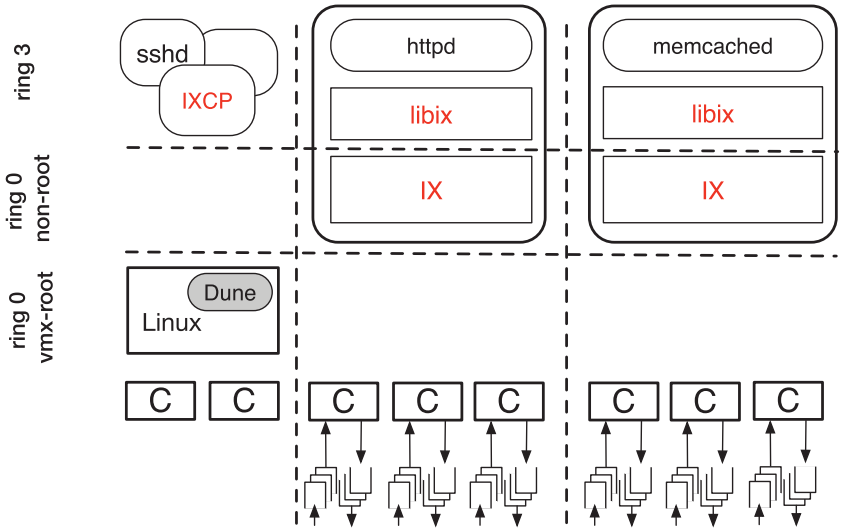


Fig. 1. Protection and separation of control and dataplane in IX.

its coherence-free design. The migration algorithm contains distinct phases that ensure that migration does not create network anomalies such as dropping packets or processing them out of order in the networking stack.

Dynamic control loop with user-defined policies. At its core, the control plane has a control loop that monitors the queuing delay to detect likely SLO violations and reacts by adding system resources within milliseconds. It monitors the utilization of the IX dataplane to similarly remove unnecessary system resources. The IX control plane relies on the host Linux kernel mechanisms to adjust system resources such as changing the processor frequency or the number of cores allocated to the IX dataplane. It relies on the IX dataplane's TCP-friendly flow group migration mechanism to balance the load among the cores. Although the control loop specifies *when* resources must be adjusted, it does not specify *which* resource must be added or removed, as this policy decision is a function of the platform's characteristics, the application's ability to scale horizontally, and the overall objective (energy proportionality or workload consolidation).

4. IX IMPLEMENTATION

4.1. Overview

Figure 1 presents the IX architecture, focusing on the separation between the control plane and the multiple dataplanes. The hardware environment is a multicore server with one or more multiqueue 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.

We run the Linux kernel in VMX root ring 0, the mode typically used to run hypervisors in virtualized systems [Uhlig et al. 2005]. We use the Dune module within Linux to enable dataplanes to run as application-specific OSs in VMX nonroot ring 0, the mode typically used to run guest kernels in virtualized systems [Belay et al. 2012].

Applications run in VMX nonroot 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, Figure 1 shows one dataplane for a multithreaded `memcached` server and another dataplane for a multithreaded `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, and 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 [Leverich and Kozyrakis 2014].

Each IX dataplane operates as a single address-space OS and supports two thread types within a shared, user-level address space: (1) *elastic threads,* which interact with the IX dataplane to initiate and consume network I/O, and (2) *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 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 [Engler et al. 1995], 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 OSs, it uses exclusively large pages (2MB). We favor large pages due to their reduced address translation overhead [Basu et al. 2013; Belay et al. 2012] 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 [Varghese and Lauck 1987]. It is optimized for the common case where most timers are canceled before they expire. We support

Table I. Lines of Code (in Thousands)

KSLOC	IX	lwIP	Dune	total
Control plane	0.4			0.4
Data plane	9.7	9.4	4.9	24.0
Linux kernel		2.5		2.5
User-level library	1.0			1.0

Table II. The IX Dataplane System Call and Event Condition API

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

extremely high-resolution timeouts, as low as 16μs, which has been shown to improve performance during TCP in-cast congestion [Vasudevan et al. 2009].

Our current IX dataplane implementation is based on Dune and requires the VT-x virtualization features available on Intel x86-64 systems [Uhlig et al. 2005]. 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.

Table I lists the code size (in thousands of SLOC [Wheeler 2001]). The rows correspond to the different protection domains of the system, while the columns correspond to the different open-source projects involved. The TCP/IP stack uses a highly modified version of lwIP [Dunkels 2001]. 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-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 multicore scalability and fine-grained timer management. However, we did not yet optimize the lwIP code for performance. Hence, the results of Section 5 have room for improvement. In addition, the IX dataplane links with an unmodified DPDK library, which is used to initially configure the NIC. DPDK code is not used during datapath operations; instead, IX accesses NIC descriptor rings directly.

4.3. Dataplane API and Operation

The elastic threads of an application interact with the IX dataplane through three asynchronous, nonblocking mechanisms summarized in Table II: they issue *batched system calls* to the dataplane; they consume *event conditions* generated by the dataplane; and they have direct, but safe, access to mbufs 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.

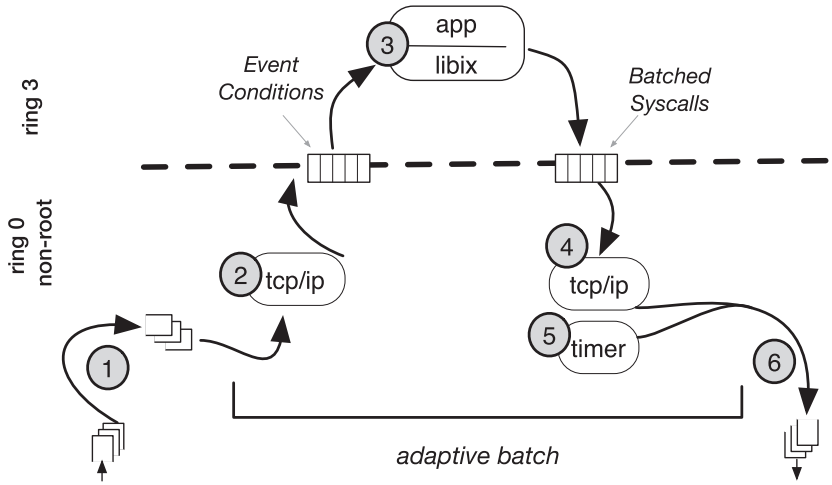


Fig. 2. Interleaving of protocol processing and application execution in the IX dataplane.

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 II 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 [Han et al. 2012].

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 OSs 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 nonblocking 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 [Fisk and Feng 2000].

Figure 2 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 informs the protocol stack of any buffers that have finished transmitting, based on the transmit ring's head position. 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. Multicore Scalability

The IX dataplane is optimized for multicore 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 [David et al. 2013]. 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 [Clements et al. 2013]. 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 [Microsoft Corp. 2014], 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 [McKenney and Slingwine 1998]. 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.

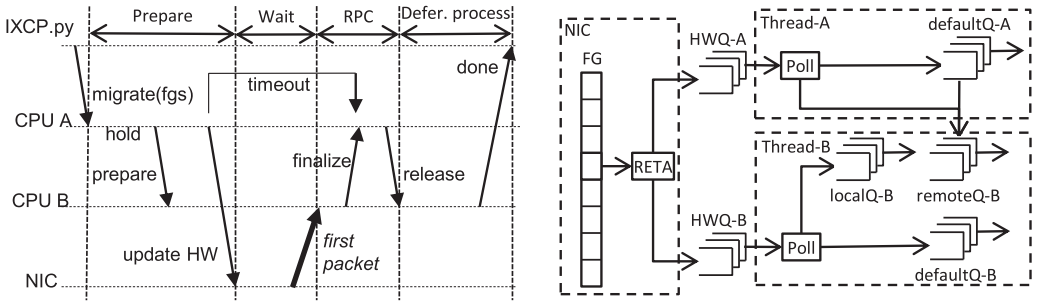


Fig. 3. Flow-group migration algorithm.

Finally, the application code may include interthread 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. Flow Group Migration

When adding or removing a thread, `IXCP` generates a set of migration requests. Each individual request is for a set of flow groups (`fgs`) currently handled by one elastic thread `A` to be handled by elastic thread `B`. To simplify the implementation, the controller serializes the migration requests and the dataplane assumes that at most one such request is in progress at any point in time. Each thread has three queues that can hold incoming network packets and ensure that packets are delivered in order to the network layer.

Figure 3 illustrates the migration steps in a thread-centric view (Figure 3(a)) and in a packet-centric view (Figure 3(b)). The controller and the dataplane threads communicate via lock-free structures in shared memory. First, the controller signals `A` to migrate `fgs` to `B`. `A` first marks each flow group of the set `fgs` with a special tag to hold off normal processing on all threads, moves packets that belong to the flow group set `fgs` from `defaultQ-A` to `remoteQ-B`, and stops all timers belonging to the flow group set. `A` then reprograms the NIC's RSS Relocation Table for index `fgs`. Packets still received by `A` will be appended to `remoteQ-B`; packets received by `B` will go to `localQ-B`.

Upon reception of the first packet whose flow group belongs to `fgs` by `B`, `B` signals `A` to initiate the final stage of migration. Then, `B` finalizes the migration by re-enabling `fgs`'s timers, removing all migration tags, and prepending to its `defaultQ-B` the packets from `remoteQ-B` and the packets from `localQ-B`. Finally, `B` notifies the control plane that the operation is complete. A migration timer ensures completion of the operation when the NIC does not receive further packets.

4.6. The IXCP Control Loop

The IXCP daemon largely relies on Linux host- and IX dataplane-provided mechanisms. It is implemented in ∼500 lines of Python. At its core, the controller adjusts processor resources by suspending and resuming IX elastic threads, specifying the mapping between flow groups and threads, and controlling the processor frequency. For server consolidation scenarios, it may additionally control the resources allocated to background tasks.

The control loop implements a user-specified policy that determines the upper bound on the acceptable queuing delay and the sequence of resource allocation adjustments. For this, it relies on a key side effect of IX's use of adaptive batching: unprocessed

packets that form the backlog are queued in a central location, namely, in step (1) in the pipeline of Figure 2. Packets are then processed in order in bounded batches to completion through both the networking stack and the application logic. In other words, each IX core operates like a simple FCFS queuing server, onto which classic queuing and control theory principles can be easily applied. In contrast, conventional operating systems distribute buffers throughout the system: in the NIC (because of coalesced interrupts), in the driver before networking processing, and in the socket layer before being consumed by the application. Furthermore, these conventional systems provide no ordering guarantees across flows, which makes it difficult to pinpoint congestion.

To estimate queuing delays, the controller monitors the iteration time τ and the queue depth Q. With B the maximal batch size, the tail latency is $\sim max(delay) = \lceil Q/B \rceil * \tau$. The dataplane computes each instantaneous metric every 10*ms* for the previous 10*ms* interval. As these metrics are subject to jitter, the dataplane computes the exponential weighted moving averages using multiple smoothing factors (α) in parallel. For example, we track the queue depth as $Q(t, \alpha) = \alpha * Q_{now} + (1-\alpha) * Q(t-1, \alpha)$. The control loop executes at a frequency of 10Hz, which is sufficient to adapt to load changes.

The control loop is responsible to determine *when* to adjust resources, but not the sequence of resource adjustment steps. For example, adding a core, enabling hyperthread, or increasing processor frequency can each increase throughput. In principle, the selection of the resource allocation (and deallocation) sequence can be derived from a Pareto analysis among all possible static configuration. For energy proportionality, the optimization metric is the energy consumption; for workload consolidation, it is the throughput of the background job. We show in Section 5.3 how such a methodology can be applied in practice for a given workload and compute platform.

Deciding when to remove resources is trickier than deciding when to add them, as shallow and near-empty queues do not provide reliable metrics. Instead, the control loop measures idle time and relies on the observation that each change in the configuration adds or removes a predictable level of throughput. The control loop makes resource deallocation decisions when idle time exceeds the throughput ratio.

4.7. Security Model

The IX API and implementation have 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 application, 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 nonresponsive 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 OF THE DATAPLANE

We compared IX to a baseline running Linux kernel version 4.2 and to mTCP [Jeong et al. 2014b]. 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 @ 3.5Ghz and Xeon E5-2650 @ 2.6Ghz. The server is a Xeon E5-2665 @ 2.4Ghz with 256GB of DRAM. Each client and server socket has eight 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 4.2 Linux kernel, the most recent at the time of writing. We enable hyperthreading when it improves performance. Except for Section 5.2.1, client machines always run Linux. All power management features are disabled for all systems in all experiments. Jumbo frames are never enabled. All 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 [Jeong et al. 2014a] 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 Section 6).

5.2. Dataplane Performance

5.2.1. 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 [Snell et al. 1996]. In all cases, we run the same system on both ends (Linux, mTCP, or IX).

Figure 4 shows the goodput achieved for different message sizes. Two IX servers have a one-way latency of 5.8μs for 64B messages and achieve goodput of 5Gbps, half of the maximum, with messages as small as 20000 bytes. In contrast, two Linux servers have a one-way latency of 19.0μs and require 192KB 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

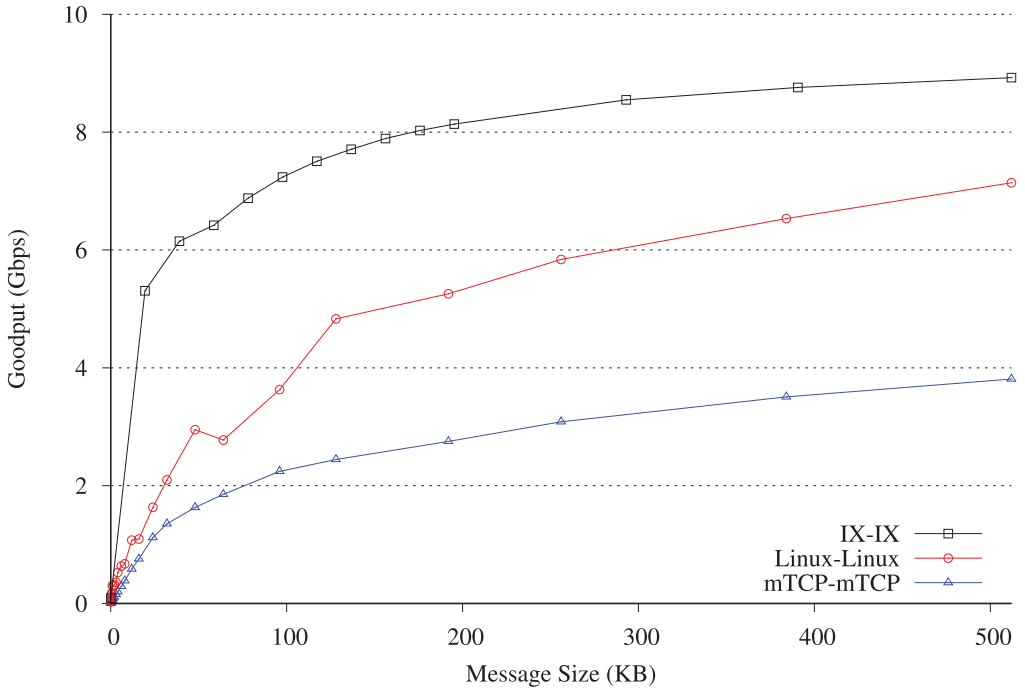


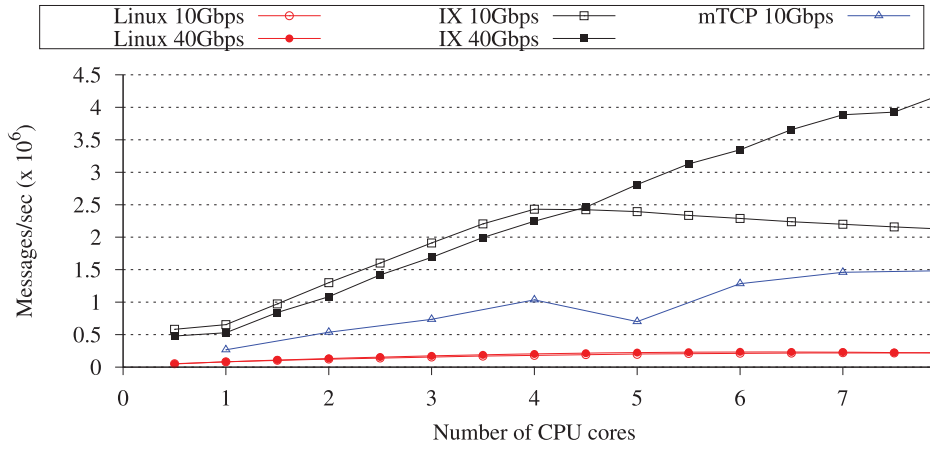
Fig. 4. NetPIPE performance for varying message sizes and system software configurations.

the cost of context switching [Jeong et al. 2014b], which comes at the expense of higher latency than both IX and Linux in this particular test.

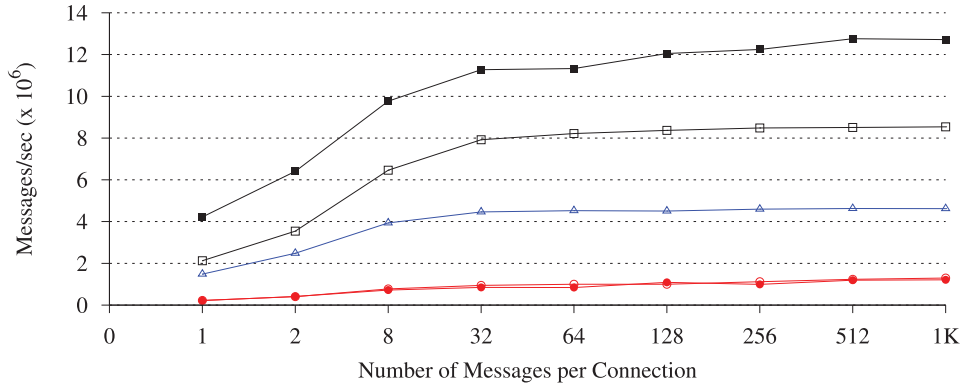
5.2.2. Throughput and Scalability. We evaluate IX's throughput and multicore scalability with the same benchmark used to evaluate MegaPipe [Han et al. 2012] and mTCP [Jeong et al. 2014b]. Eighteen clients connect to a single server listening on a single port, send a remote request of size s bytes, and 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 Jeong et al. [2014b], clients close the connection using a reset (`TCP RST`) to avoid exhausting ephemeral ports.

Figure 5 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 [Jeong et al. 2014b]. For all three tests (core scaling, message count scaling, message size scaling), IX scales more aggressively than mTCP and Linux. Figure 5(a) shows that IX needs only four cores to saturate the 10GbE link, whereas mTCP requires all eight cores. On Figure 5(b), for 1,024 round trips per connection, IX delivers 8.5 million messages per second, which is 1.8× the throughput of mTCP and 6.6× that of Linux. With this packet rate, IX achieves line rate and is limited only by 10GbE bandwidth.

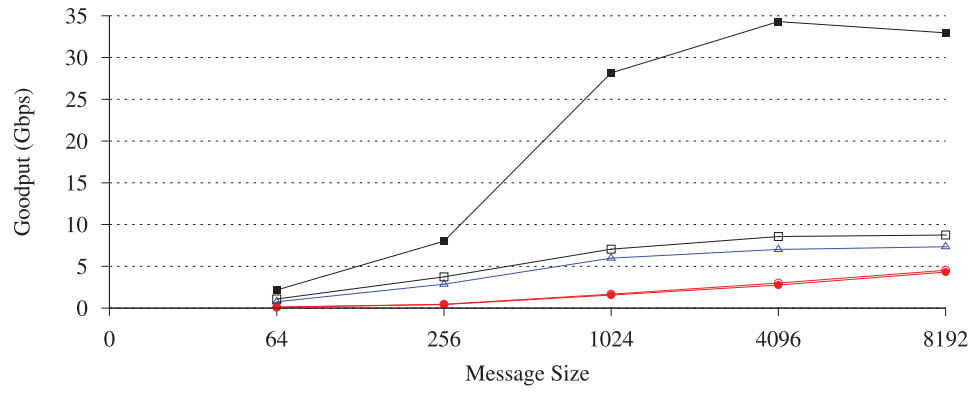
Figure 5 also shows that IX scales well beyond 10GbE to a 4x10GbE configuration. Figure 5(a) shows that IX linearly scales to deliver 4.2 million TCP connections per second on 4x10GbE. Figure 5(b) shows a speedup of 2.0× with $n = 1$ and of 1.5× with $n = 1,024$ over 10GbE IX. Finally, Figure 5(c) shows IX can deliver 8KB messages with



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



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



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

Fig. 5. Multicore scalability and high connection churn for 10GbE and 4x10GbE setups. In (a), half steps indicate hyperthreads.

a goodput of 33.0 Gbps, for a wire throughput of 36.3 Gbps, out of a possible 39.7 Gbps. Overall, IX makes it practical to scale protected TCP/IP processing beyond 10GbE, even with a single-socket multicore server.

5.2.3. Connection Scalability. We also evaluate IX's scalability when handling a large number of concurrent connections on the 4x10GbE setup. Eighteen 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.

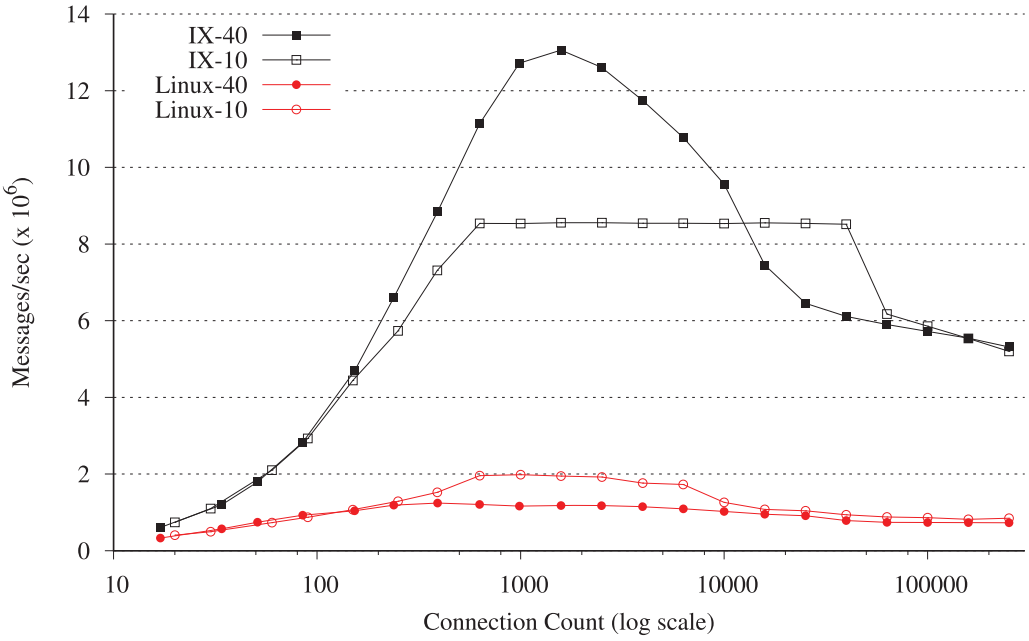
Figure 6 shows up to 250,000 connections, which is the upper bound we can reach with the available client machines. As expected, Figure 6(a) shows that throughput increases with the degree of connection concurrency but then decreases for very large connection counts due to the increasingly high cost of multiplexing among open connections. At the peak, IX performs 11× better than Linux, consistent with the results from Figure 5(b). With 250,000 connections and 4x10GbE, IX is able to deliver 41% of its own peak throughput.

Figure 6(b) shows 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 [Huggahalli et al. 2005], eliminates nearly all cache misses associated with DMA transfers when given enough time between polling intervals, resulting in as little as 1.6 L3 cache misses per message for up to 2,500 concurrent connections, a scale where all of IX's data structures fit easily in the L3 cache. In contrast, the workload averages 29 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.

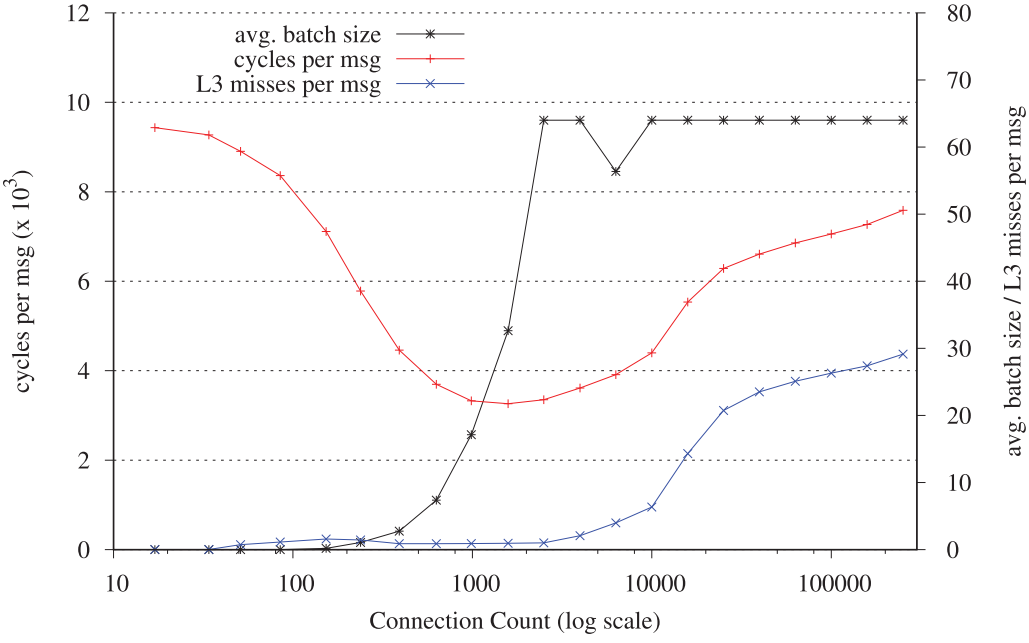
Figure 6(b) additionally gives insights about the positive impact of the adaptive batching. As the load increases, the average batch size increases from 0 to the maximum configured value, which is 64 in our benchmark setup. At the same time, the average number of cycles per message decreases from 9,000 to less than 4,000, before it starts increasing again due to the negative impact of L3 cache misses.

5.2.4. 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 [memcached 2014]. 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 80% of execution time in kernel mode for network processing [Leverich and Kozyrakis 2014]. It is a difficult application to scale because the common deployments involve high connection counts for `memcached` servers and small-sized requests and replies [Atikoglu et al. 2012; Nishtala et al. 2013]. Furthermore, `memcached` has well-known scalability limitations [Lim et al. 2014]. To alleviate some of the limitations, we configure `memcached` with a larger hash table size (`-o hashpower=20`) and use a random replacement policy instead of the built-in LRU, which requires a global lock. We configure `memcached` similarly for Linux and IX.

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 [Leverich 2014]. `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 a time across 32 open connections, to eliminate statistical errors due to slight potential imbalances across network card queues and respective CPU cores



(a) Throughput for 10GbE and 4x10GbE configurations.



(b) Hardware metrics for IX on the 4x10GbE configuration.

Fig. 6. Connection scalability of IX.

Table III. Unloaded Latency and Maximum RPS for a Given Service-Level Agreement for the Memcache Workloads ETC and USR

Configuration	Minimum Latency @99th pct	RPS for SLA: <500μs @99th pct
ETC-Linux	65μs	898K
ETC-IX	34μs	4186K
USR-Linux	66μs	902K
USR-IX	33μs	5817K

handling those queues. We configure `mutilate` to generate load representative of two workloads from Facebook [Atikoglu et al. 2012]: the ETC workload that represents that highest-capacity deployment in Facebook has 20B to 70B keys, 1B to 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 11 client machines to generate load for a total of 2,752 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 [Dean and Barroso 2013]. 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 Leverich and Kozyrakis [2014]: we pin memcached threads, configure interrupt distribution based on thread affinity, and tune interrupt moderation thresholds. Additionally, we increase the socket accept queue size and disable SYN cookies via `sysctl` and via the respective `memcached` command line argument to accommodate for the large connection accept rate at the beginning of the benchmark. Finally, to resolve observed unexpected 99th percentile latency spikes when running `memcached` under Linux, we disable transparent huge pages via `sysfs`, instruct `memcached` to use the `mlockall` system call, and utilize `numactl` to pin memory pages on the desired NUMA node of our server. 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: eight cores with hyperthreading enabled.

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` [Fan et al. 2013] or to support zero-copy I/O operations.

Figures 7(a) and 7(b) show the throughput latency curves for the two `memcached` workloads for Linux and IX, while Table III reports the unloaded, round-trip latencies and maximum request rate that meets a service-level agreement, both measured at the 99th percentile. IX cuts the unloaded latency of both workloads in 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 ~80% in the Linux kernel to 60% in the IX dataplane kernel. This allows IX to increase throughput by 4.7× and 6.4× for `ETC` and `USR`, respectively, at a 500μs tail latency SLA.

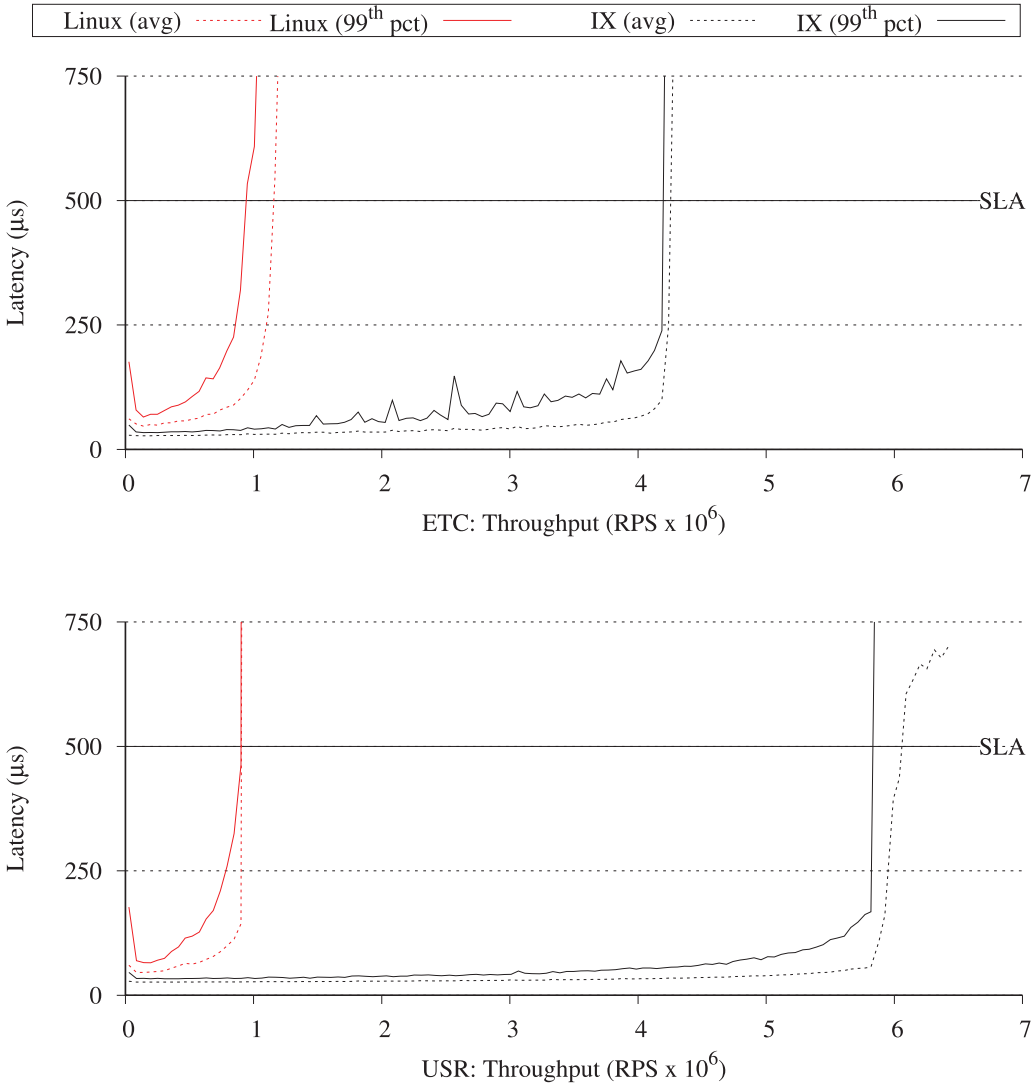
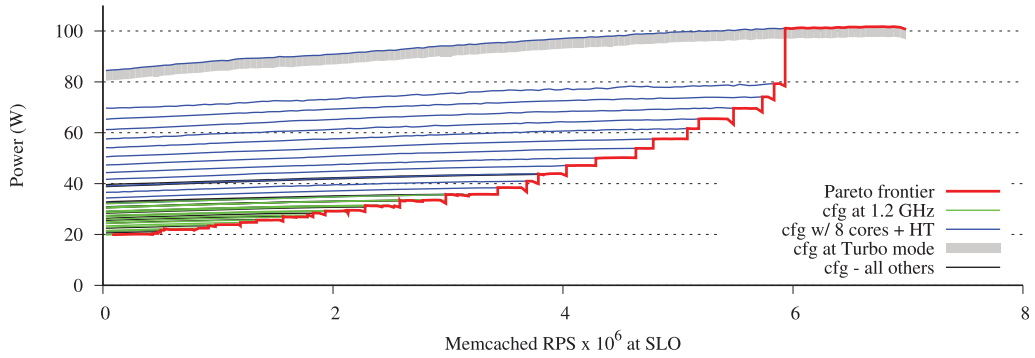


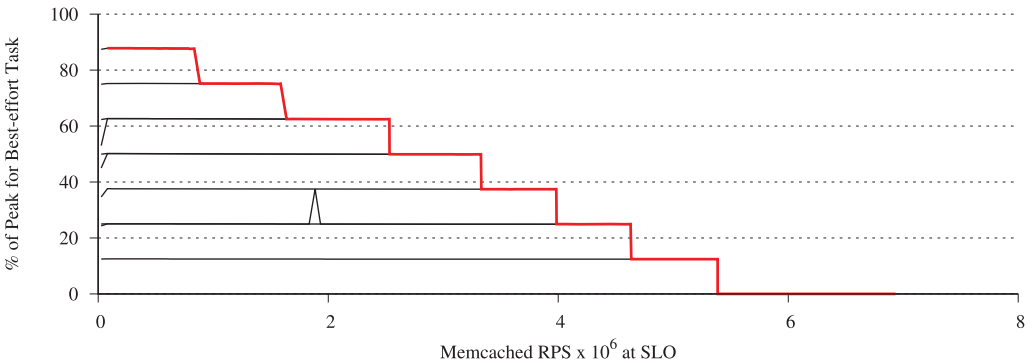
Fig. 7. Average and 99th percentile latency as a function of throughput for the ETC and USR memcached workloads.

5.3. Pareto-Optimal Static Configurations

Static resource configurations allow for controlled experiments to quantify the trade-off between an application's performance and the resources consumed. Our approach limits bias by considering many possible static configurations in the three-dimensional space of core, hyperthread, and frequency. For each static configuration, we characterize the maximum load that meets the SLO ($\leq 500\mu s$@99th percentile); we then measure the energy draw and throughput of the background job for all load levels up to the maximum load supported. From this large dataset, we derive the set of meaningful static configurations and build the Pareto efficiency frontier. The frontier specifies, for any possible load level, the optimal static configuration and the resulting minimal energy draw or maximum background throughput, depending on the scenario.



(a) Energy Proportionality



(b) Server Consolidation (IX)

Fig. 8. Pareto efficiency for energy proportionality and workload consolidation for IX. The Pareto efficiency is in red, while the various static configurations are color-coded according to their distinctive characteristics.

Figure 8 presents the frontier for the memcached USR workload for two different policies: energy proportionality, which aims to minimize the amount of energy consumed while maintaining SLO, and workload consolidation, which aims to maximize the throughput of some background process while also maintaining the SLO of the latency-sensitive application.

The graphs each plot the objective—which is either to minimize energy or maximize background throughput—as a function of the foreground throughput, provided that the SLO is met. Except for the red lines, each line corresponds to a distinct static configuration of the system: the green curves correspond to configuration at the minimal clock rate of 1.2GHz, the blue curves use all available cores and hyperthreads, and other configurations are in black. In Turbo Boost mode, the energy drawn is reported as a band since it depends on operating temperature.[1]

[1]For any given throughput level, we observe that the reported power utilization is stable for all CPU frequencies *except* for Turbo Boost. When running in Turbo Boost, the temperature of the CPU gradually rises over a few minutes from 58° to 78°, and with it the dissipated energy rises by 4 W for the same level of performance. The experiments in Section 5.3 run for a long time in Turbo Boost mode with a hot processor; we therefore report those results as an energy band of 4 W.

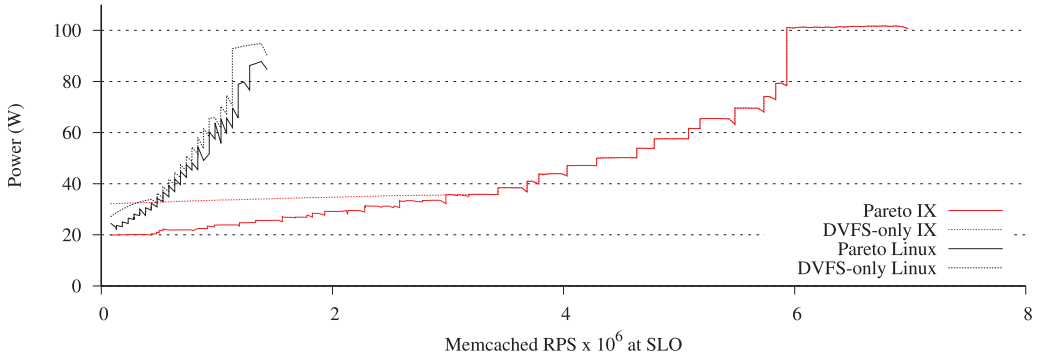


Fig. 9. Energy proportionality comparison between the Pareto-optimal frontier considering only DVFS adjustments and the full Pareto frontier considering core allocation, hyperthread allocations, and frequency.

Finally, the red line is the Pareto frontier, which corresponds, for any load level, to the optimal result using any of the static configurations available. Each graph only shows the static configurations that participate in the frontier.

Energy proportionality. We evaluate 224 distinct combinations: from one to eight cores, using consistently either one or two threads per core, for 14 different DVFS levels from 1.2GHz to 2.4GHz as well as Turbo Boost. Figure 8(a) shows the 45 static configurations (out of 224) that build the Pareto frontier for energy proportionality. The figures confirm the intuition that (1) various static configurations have very different dynamic ranges, beyond which they are no longer able to meet the SLO; (2) each static configuration draws substantially different levels of energy for the same amount of work; (3) at the low end of the curve, many distinct configurations operate at the minimal frequency of 1.2GHz, obviously with a different number of cores and threads, and contribute to the frontier—these are shown in green in the figure; and (4) at the high end of the range, many configurations operate with the maximum of eight cores, with different frequencies including Turbo Boost.

Consolidation. The methodology here is a little different. We first characterize the background job and observe that it delivers energy-proportional throughput up to 2.4GHz, but that Turbo Boost came at an energy/throughput premium. Consequently, we restrict the Pareto configuration space at 2.4GHz; the objective function is the throughput of the background job, expressed as a fraction of the throughput of that same job without any foreground application. Background jobs run on all cores that are not used by the foreground application. Figure 8(b) shows the background throughput, expressed as a fraction of the standalone throughput, as a function of the foreground throughput, provided that the foreground application meets the SLO: as the foreground application requires additional cores to meet the SLO, the background throughput decreases proportionally.

DVFS-only alternative. Figure 9 further analyzes the data and compares the Pareto frontiers of Linux 4.2 and IX for the energy-proportional scenario with an alternate frontier that only considers changes in DVFS frequency. We observe that the impact of DVFS-only controls differs noticeably between Linux and IX: with Linux, the DVFS-only alternate frontier is very close to the Pareto frontier, meaning that a DVFS-only approach such as Pegasus [Lo et al. 2014] or Adrenaline [Hsu et al. 2015] would be adequate. This is due to Linux's idling behavior, which saves resources. In the case of IX, however—and likely for any polling-based dataplane—a DVFS-only scheduler would provide worse energy proportionality at low-moderate loads than a corresponding

Linux-based solution. As many datacenter servers operate in the 10% to 30% range [Barroso et al. 2013], we conclude that a dynamic resource allocation scheme involving both DVFS and core allocation is necessary for dataplane architectures.

5.4. Control Plane Effectiveness

We use the results from Section 5.3 to derive a resource configuration policy framework, whose purpose is to determine the sequence of configurations to be applied, as a function of the load on the foreground application, to both the foreground (latency-sensitive) and background (batch) applications. Specifically, given an ever-increasing (or -decreasing) load on the foreground applications, the goal is to determine the sequence of resource configurations minimizing energy consumption or maximizing background throughput, respectively.

We observe that (1) the latency-sensitive application (`memcached`) can scale nearly linearly, up to the eight cores of the processor; (2) it benefits from running a second thread on each core, with a consistent speedup of 1.3×; (3) it is most energy efficient to first utilize the various cores, and only then to enable the second hyperthread on each core, rather than the other way around; and (4) it is least energy efficient to increase the frequency.

We observe that the background application (1) also scales linearly but (2) does not benefit from the second hyperthread, and (3) is nearly energy proportional across the frequency spectrum, with the exception of Turbo Boost. From a total-cost-of-ownership perspective, the most efficient operating point for the workload consolidation of the background task is therefore to run the system at the processor's nominal 2.4GHz frequency whenever possible.

We combine these observations with the data from the Pareto analysis and derive the following policies:

Energy-proportional policy. As a base state, run with only one core and hyperthread with the socket set at the minimal clock rate (1.2GHz). To add resources, first enable additional cores, then enable hyperthreads on all cores (as a single step), and only after that gradually increase the clock rate until reaching the nominal rate (2.4GHz); finally, enable Turbo Boost. To remove resources, do the opposite. This policy leads to a sequence of 22 different configurations.

Workload consolidation policy. As a base state, run the background jobs on all available cores with the processor at the nominal clock rate. To add resources to the foreground application, first shift cores from the background thread to the foreground application one at a time. This is done by first suspending the background threads; use both hyperthreads of the newly freed core for the foreground application. Next, stop the background job entirely and allocate all cores to the foreground applications. As a final step, enable Turbo Boost. This policy leads to a sequence of nine different configurations.

These policies closely track the corresponding Pareto frontier. For energy proportionality, (1) the 45 different static configurations of the frontier are a superset of the configurations enabled by the policy, and (2) the difference in overall impact in terms of energy spent is marginal. For consolidation, Pareto and policy nearly identically overlap.

We use three synthetic, time-accelerated load patterns to evaluate the effectiveness of the control loop under stressful conditions. All three vary between nearly idle and maximum throughput within a 4-minute period: the *slope* pattern gradually raises the target load from 0 and 6.2M RPS and then reduces its load; the *step* pattern increases load by 500 KRPS every 10 seconds; and finally the *sine+noise* pattern is a

Table IV. Energy Proportionality and Consolidation Gains

	Smooth	Step	Sine+Noise
Energy proportionality (W)			
Max. power	92	93	94
Measured	43 (−54%)	46 (−51%)	51 (−46%)
Pareto bound	38 (−59%)	38 (−59%)	43 (−54%)
Server consolidation opportunity (% of peak)			
Pareto bound	53%	51%	44%
Measured	47%	43%	35%

basic sinusoidal pattern modified by randomly adding sharp noise that is uniformly distributed over [−250,+250] KRPS and recomputed every 5 seconds. The slope pattern provides a baseline to study smooth changes, the step pattern models abrupt and massive changes, and the sine+noise pattern is representative of daily web patterns [Urdaneta et al. 2009].

Figure 10, Figure 11, and Figure 12 show the results of these three dynamic load patterns for the energy proportionality and workload consolidation scenarios. In each case, the top figure measures the observed throughput. They are annotated with the control loop events that add resources (green) or remove them (red). Empty triangles correspond to core allocations and full triangles to DVFS changes. The middle figure evaluates the soundness of the algorithm and reports the 99th percentile latency, as observed by a client machine and reported every second. Finally, the bottom figures compare the overall efficiency of our solution based on dynamic resource controls with (1) the maximal static configuration, using all cores and Turbo Boost, and (2) the ideal, synthetic efficiency computed using the Pareto frontier of Figure 8.

Energy proportionality. The left column of Figures 10, 11, and 12 shows the dynamic behavior for the energy proportionality scenario. The top-left graph shows that the workload tracks the desired throughput of the pattern and exercises the entire sequence of configurations, gradually adding cores, enabling hyperthreading, increasing the frequency, and finally enabling Turbo Boost, before doing it in reverse. The step pattern of Figure 11 is particularly challenging, as the instant change in load level requires multiple, back-to-back configuration changes. With a few exceptions, the middle-left graph shows that the latencies remain well below the 500μs SLO. We further discuss the violations later. For these three figures, the bottom-left graph compares the power dissipated by the workload with the corresponding power levels as determined by the Pareto frontier (lower bound) or the maximum static configuration (upper bound). This graph measures the effectiveness of the control loop to maximize energy proportionality. We observe that the dynamic (actually measured) power curve tracks the Pareto (synthetic) curve well, which defines a bound on energy savings. When the dynamic resource controls enter Turbo Boost mode, the measured power in all three cases starts at the lower end of the 4 W range and then gradually rises, as expected. Table IV shows that the three patterns have Pareto savings bounds of 54%, 59% and 59%. IX's dynamic resource controls results in energy savings of 46%, 51% and 54%, which is 85%, 86% and 92% of the theoretical bound.

Consolidation. The right column of Figures 10, 11, and 12 shows the dynamic behavior for the workload consolidation scenario. Here also, the top-right graphs show that the throughput tracks well the desired load. Recall that the consolidation policy always operates at the processor's nominal rate (or Turbo), which limits the number of configuration changes. The middle-right graph similarly confirms that the system meets the SLO, with few exceptions. The bottom-right graphs plot the throughput of

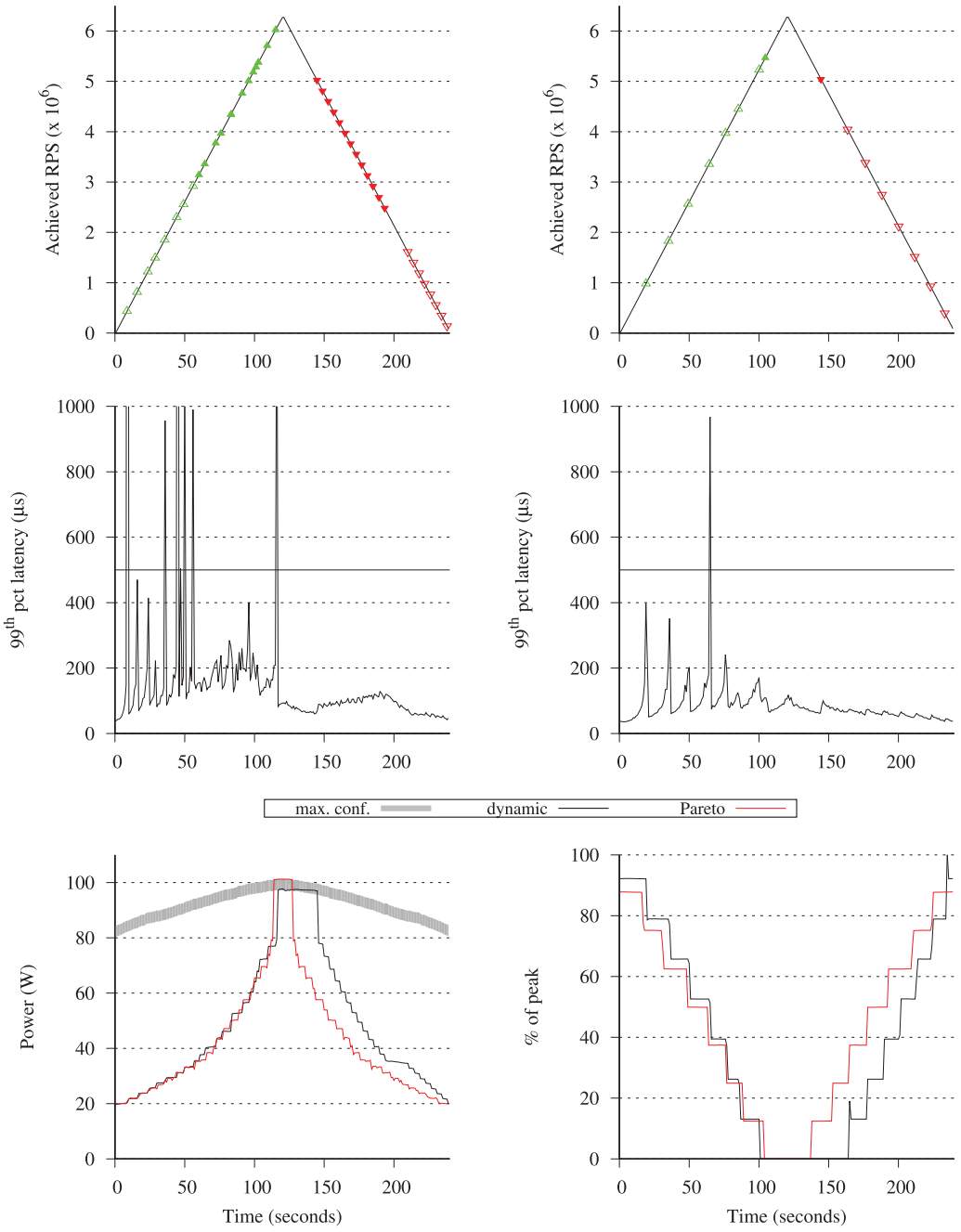


Fig. 10. Energy proportionality (left) and workload consolidation (right) for the *slope* pattern.

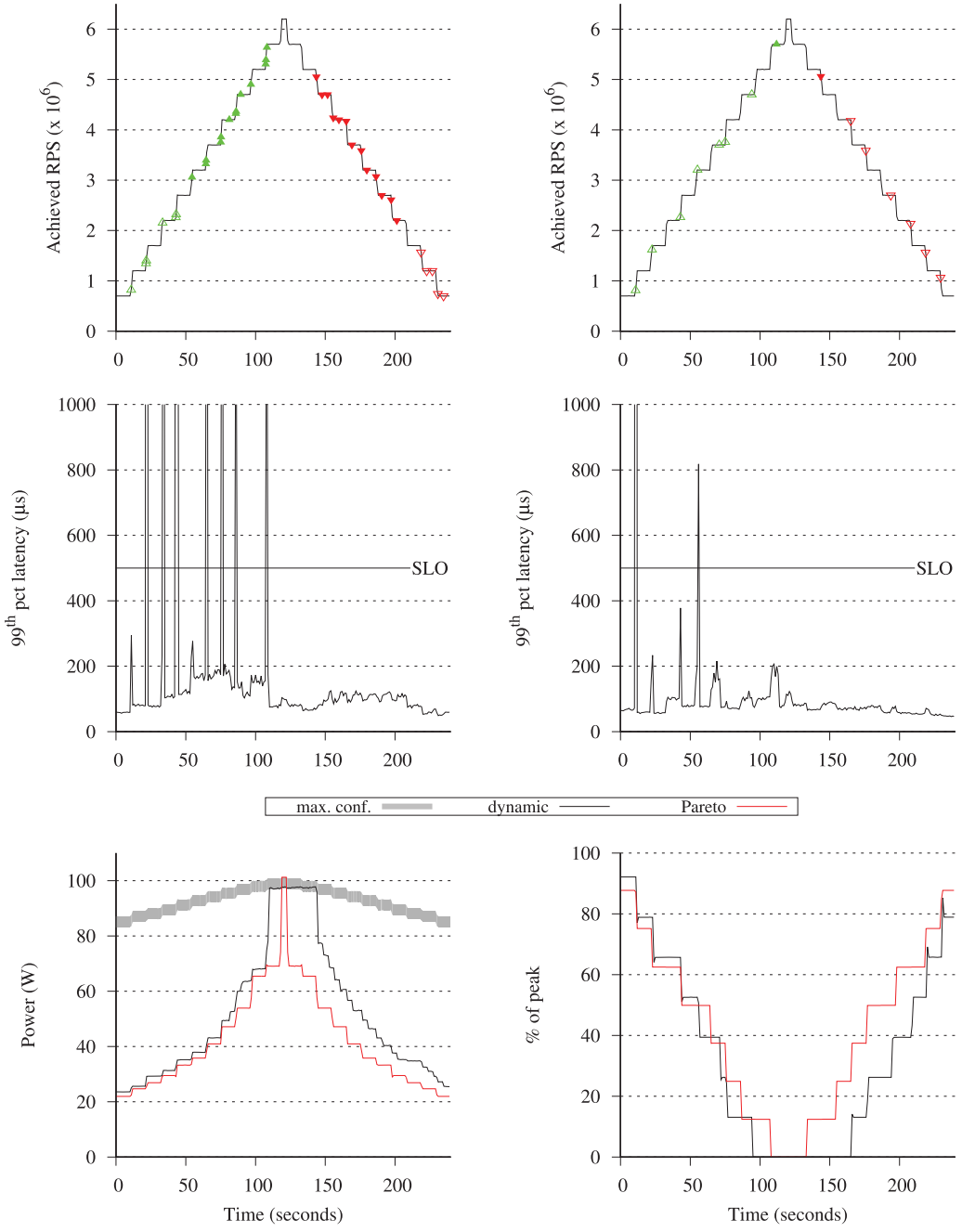


Fig. 11. Energy proportionality (left) and workload consolidation (right) for the *step* pattern.

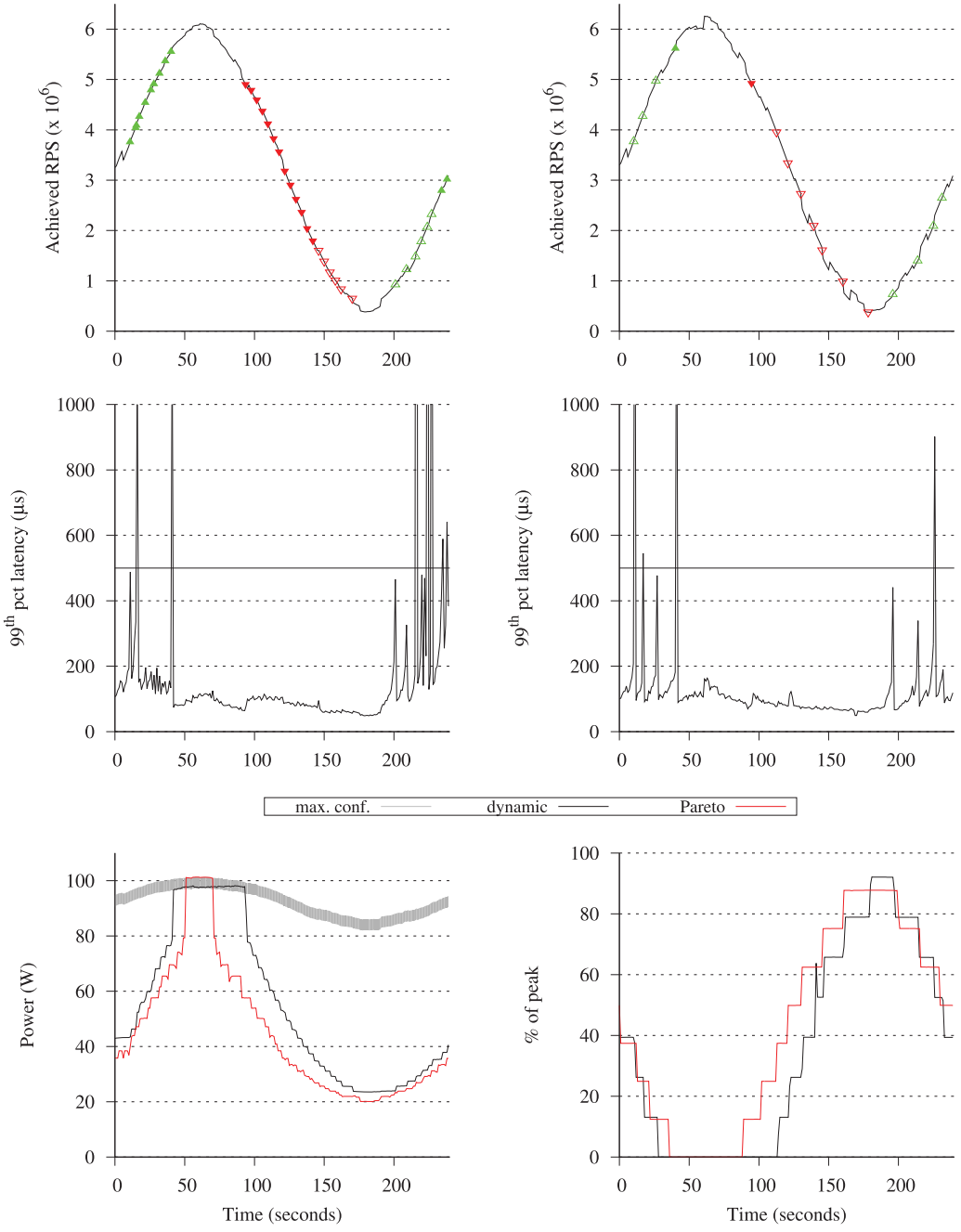


Fig. 12. Energy proportionality (left) and workload consolidation (right) for the *sin+noise* pattern.

Table V. Breakdown of Flow Group Migration Measured During the Six Benchmarks

		avg	95th pct.	max.	stddev
add core	prepare (μs)	119	212	10,082	653
	wait (μs)	113	475	1,018	158
	rpc (μs)	102	238	378	75
	deferred (μs)	125	460	2,534	283
	total (μs)	462	1,227	12,804	927
	# packets	83	285	2,753	280
remove core	prepare (μs)	23	49	312	25
	wait (μs)	33	106	176	31
	rpc (μs)	12	27	48	7
	deferred (μs)	16	43	82	11
	total (μs)	86	154	370	40
	# packets	3	9	25	3

the background batch application, expressed as a percentage of its throughput on a dedicated processor at 2.4GHz. We compare it only to the Pareto-optimal upper bound as a maximum configuration would monopolize cores and deliver zero background throughput. Table IV shows that, for these three patterns, our consolidation policy delivers 35%–47% of the standalone throughput of the background job, which corresponds to 80%–89% of the Pareto bound.

SLO violations. A careful study of the SLO violations of the six runs shows that they fall into two categories. First, there are 16 violations caused by delays in packet processing due to flow group migrations resulting from the addition of a core. Second, there are nine violations caused by an abrupt increase of throughput, mostly in the step pattern, which occur before any flow migrations. The control plane then reacts quickly (in ∼100ms) and accommodates to the new throughput by adjusting resources. To further confirm the abrupt nature of throughput increase specific to the step pattern, we note that the system performed up to three consecutive increases in resources in order to resolve a single violation. Twenty-three of the 25 total violations last a single second, with the remaining two violations lasting 2 seconds. We believe that the compliance with the SLO achieved by our system is more than adequate for any practical production deployment.

Flow group migration analysis. Table V measures the latency of the 565 flow group migrations that occur during the six benchmarks, as described in Section 4.5. It also reports the total number of packets whose processing is deferred during the migration (rather than dropped or reordered). We first observe that migrations present distinct behaviors when scaling up and when scaling down the number of cores. The difference can be intuitively explained since the migrations during the scale-up are performed in a heavily loaded system, while the system during the scale-down is partially idle. In absolute terms, migrations that occur when adding a core take 462 on average and less than 1.5ms 95% of the time. The outliers can be explained by rare occurrences of longer preparation times or when processing up to 2753 deferred packets.

6. DISCUSSION

What makes IX *fast*. The results in Section 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

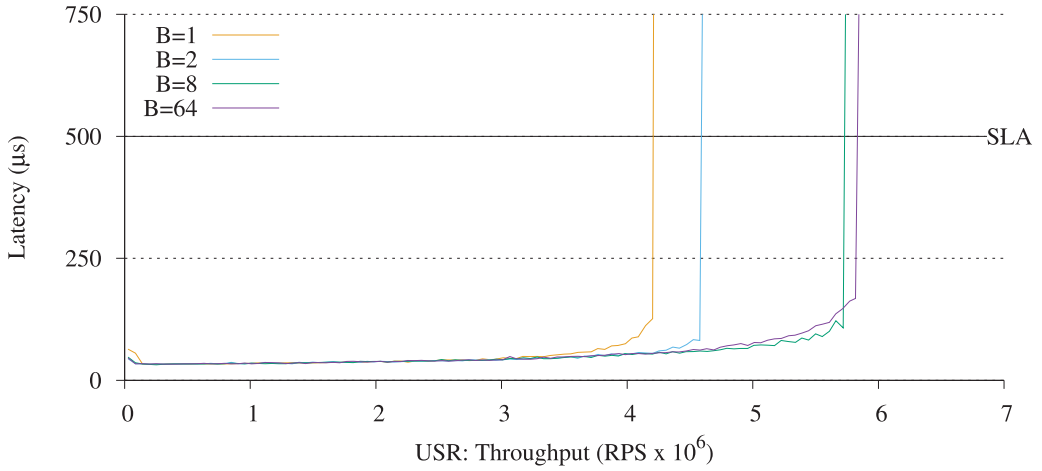


Fig. 13. 99th percentile latency as a function of throughput for USR workload from Figure 7(b), for different values of the batch bound B.

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 as the `libix` event library. The zero-copy approach helps even when the user-level libraries add a level of copying, as 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 multicore 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 the VMX nonroot mode add only a small amount of extra overhead, on the order of a single L3 cache miss [Belay et al. 2012]. 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. Figure 13 compares the latency versus throughput on the `USR`memcached workload of Figure 7(b) 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 $B = 1$ and $B = 16$. 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 coalesce PCIe writes on the transmit path, as that would have impacted latency.

Using Pareto as a guide. Even though the Pareto results are not used by the dynamic resource controller, the Pareto frontier proved to be a valuable guide, first to motivate and quantify the problem, then to derive the configuration policy sequence, and finally to evaluate the effectiveness of the dynamic resource control by setting an upper bound on the gains resulting from dynamic resource allocation. Many factors such as software scalability, hardware resource contention, and network and disk I/O bottlenecks will influence the Pareto frontier of any given application, and therefore the derived control loop policies. Without violating the SLO, the methodology explicitly trades off worst average and tail latency for better overall efficiency. More complex SLOs, taking into account multiple aspects of latency distribution, would define a different Pareto frontier and likely require adjustments to the control loop.

Adaptive, flow-centric scheduling. The new flow-group migration algorithm of Section 4.5 leads to a *flow-centric* approach to resource scheduling, where the network stack and application logic always follow the steering decision. POSIX applications can balance flows by migrating file descriptors between threads or processes, but this tends to be inefficient because it is difficult for the kernel to match the flow's destination receive queue to changes in CPU affinity. Flow director can be used by Linux to adjust the affinity of individual network flows as a reactive measure to application-level and kernel thread migration rebalancing, but the limited size of the redirection table prevents this mechanism from scaling to large connection counts. By contrast, our approach allows flows to be migrated in entire groups, improving efficiency, and is compatible with more scalable hardware flow steering mechanisms based on RSS.

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. 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, nonblocking 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 Figure 2). Occasional timer interrupts are also required to ensure full TCP compliance in the event an elastic thread blocks for an extended period.

Hardware trends. Our experimental setup uses one *Sandy Bridge* processor and the Intel 82599 NIC [Intel Corp. 2014a]. Hash filters for flow group steering could benefit from recent trends in NIC hardware. For example, Intel's new XL710 chipset [Intel Corp. 2014b] has a 512 entry hash LUT (as well as independent 64 entry LUTs for each VF) in contrast to the 128 entries available in the 82599 chipset. This has the potential to reduce connection imbalances between cores, especially with high core counts. The newly released *Haswell* processors provide per-core DVFS controls, which further increases the Pareto space.

Future work. We also plan to 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 [Alizadeh et al. 2010] and ECN [Ramakrishnan et al. 2001]. 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 nonvolatile memory access. Finally, we will investigate library support for alternative APIs on top of our low-level interface, such as MegaPipe [Han et al. 2012], cooperative threading [von Behren et al. 2003], and rule-based models [Stutsman and Ousterhout 2013]. 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 Section 2.3.

Hardware virtualization. Hardware support for virtualization naturally separates control and execution functions, for example, to build type-2 hypervisors [Bugnion et al. 2012; Kivity 2007], run virtual appliances [Sapuntzakis et al. 2003], or provide processes with access to privileged instructions [Belay et al. 2012]. Similar to IX, Arrakis uses hardware virtualization to separate the I/O dataplane from the control plane [Peter et al. 2016]. 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 [Baumann et al. 2009] 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 [Engler et al. 1995]. Library operating systems often run as virtual machines [Bugnion et al. 1997] used, for instance, to deploy cloud services [Madhavapeddy et al. 2013]. IX limits itself to the implementation of the networking stack, allowing applications to implement their own resource management policies, for example, 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 [Han et al. 2012; Jeong et al. 2014b; Rizzo 2012; Soares and Stumm 2010]. 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 [Pai et al. 2000]. POSIX OSs have been modified to support zero copy through page remapping and copy-on-write [Chu 1996]. 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 [Levis et al. 2004]. However, IX is optimized for datacenter workloads, while TinyOS focuses on memory-constrained, sensor environments.

Scheduling. Scheduler activations [Anderson et al. 1992] give applications greater control over hardware threads and provide a mechanism for custom application-level scheduling. Callisto [Harris et al. 2014] uses a similar strategy to improve the performance of colocated parallel runtime systems. Our approach differs in that an independent control plane manages the scheduling of hardware threads based on receive queuing latency indicators while the dataplane exposes a simple kernel threading abstraction. SEDA [Welsh et al. 2001] also monitors queuing behavior to make scheduling decisions such as thread pool sizing. Chronos [Kapoor et al. 2012] makes use of software-based flow steering, but with a focus on balancing load to reduce latency. Affinity Accept

[Pesterev et al. 2012] embraces a mixture of software- and hardware-based flow steering in order to improve TCP connection affinity and increase throughput. We focus instead on energy proportionality and workload consolidation.

Energy proportionality. The energy proportionality problem [Barroso and Hölzle 2007] has been well explored in previous work. Some systems have focused on solutions tailored to throughput-oriented workloads [Meisner et al. 2011a] or read-only workloads [Krioukov et al. 2011]. Meisner et al. [2011b] highlight unique challenges for low-latency workloads and advocate full system active low-power modes. Similar to our system, Pegasus [Lo et al. 2014] achieves CPU energy proportionality for low-latency workloads. Our work expands on Pegasus by exploring the elastic allocation of hardware threads in combination with processor power management states and by basing scheduling decisions on internal latency metrics within a host endpoint instead of an external controller. [Niccolini et al. 2012] show that a software router, running on a dedicated machine, can be made energy-proportional. Similar to our approach, queue length is used as a control signal to manage core allocation and DVFS settings. However, we focus on latency-sensitive applications, rather than middlebox traffic, and consider the additional case of workload consolidation.

Colocation. Because host endpoints contain some components that are not energy proportional and thus are most efficient when operating at 100% utilization, colocation of workloads is also an important tool for improving energy efficiency. At the cluster scheduler level, BubbleUp [Mars et al. 2012] and Paragon [Delimitrou and Kozyrakis 2014] make scheduling decisions that are interference aware through efficient classification of the behavior of workload colocation. Leverich and Kozyrakis [2014] demonstrate that colocation of batch and low-latency jobs is possible on commodity operating systems. Our approach explores this issue at higher throughputs and with tighter-latency SLOs. Bubble-Flux [Yang et al. 2013] additionally controls background threads; we control background and latency-sensitive threads. CPI^2 [Zhang et al. 2013] detects performance interference by observing changes in CPI and throttles offending jobs. This work is orthogonal to ours and could be a useful additional signal for our control plane. Heracles manages multiple hardware and software isolation mechanisms, including packet scheduling and cache partitioning, to colocate latency-sensitive applications with batch tasks while maintaining millisecond SLOs [Lo et al. 2015]. We limit our focus to DVFS and core assignment but target more aggressive SLOs.

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 multicore servers.

The dynamic resource controller allocates cores and sets processor frequency to adapt to changes in the load of latency-sensitive applications. The novel rebalancing mechanisms do not impact the steady-state performance of the dataplane and can migrate a set of flow groups in milliseconds without dropping or reordering packets. We develop two resource control policies focused on optimizing energy proportionality and workload consolidation.

On microbenchmarks, IX noticeably outperforms both Linux and mTCP in terms of 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 6.4×, while reducing tail latency by more than 2×.

We use three varying load patterns to evaluate the effectiveness of our approach to resource control. Our results show that resource controls can save 46%–54% of the processor's energy, or enable a background job to deliver 35%–47% of its standalone throughput. We synthesize the Pareto frontier by combining the behavior of all possible static configurations. Our policies deliver 85%–92% of the Pareto-optimal bound in terms of energy proportionality and 80%–89% in terms of consolidation.

ACKNOWLEDGMENTS

The authors would like to thank David Mazières for his many insights into the system and his detailed feedback on the article. We also thank Katerina Argyraki, James Larus, Jacob Leverich, Philip Levis, Andrew Warfield, Willy Zwaenepoel, and the anonymous reviewers for their comments.

REFERENCES

Mohammad Alizadeh, Albert G. Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM 2010 Conference*. 63–74. DOI:http://dx.doi.org/10.1145/1851182.1851192

Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. 1992. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Trans. Comput. Syst.* 10, 1 (1992), 53–79. DOI:http://dx.doi.org/10.1145/146941.146944

Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. 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*. 53–64. DOI:http://dx.doi.org/10.1145/2254756.2254766

Luiz Andre Barroso. 2014. 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/.

Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. 2013. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines* (2nd ed.). Morgan & Claypool Publishers.

Luiz André Barroso and Urs Hölzle. 2007. The case for energy-proportional computing. *IEEE Comput.* 40, 12 (2007), 33–37. DOI:http://dx.doi.org/10.1109/MC.2007.443

Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. 2013. Efficient virtual memory for big memory servers. In *Proceedings of the 40th International Symposium on Computer Architecture (ISCA'13)*. 237–248. DOI:http://dx.doi.org/10.1145/2485922.2485943

Andrew Baumann, Paul Barham, Pierre-Évariste Dagand, Timothy L. Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. 2009. The multikernel: A new OS architecture for scalable multicore systems. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP'09)*. 29–44. DOI:http://dx.doi.org/10.1145/1629575.1629579

Adam Belay, Andrea Bittau, Ali José Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. 2012. Dune: Safe user-level access to privileged CPU features. In *Proceedings of the 10th Symposium on Operating System Design and Implementation (OSDI'12)*. 335–348.

Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2014. IX: A protected dataplane operating system for high throughput and low latency. In *Proceedings of the 11th Symposium on Operating System Design and Implementation (OSDI'14)*. 49–65.

Steven M. Bellovin. 2004. A look back at "security problems in the TCP/IP protocol suite." In *Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC'04)*. 229–249. DOI:http://dx.doi.org/10.1109/CSAC.2004.3

Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. 1997. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Trans. Comput. Syst.* 15, 4 (1997), 412–447. DOI:http://dx.doi.org/10.1145/265924.265930

Edouard Bugnion, Scott Devine, Mendel Rosenblum, Jeremy Sugerman, and Edward Y. Wang. 2012. Bringing virtualization to the x86 architecture with the original VMware workstation. *ACM Trans. Comput. Syst.* 30, 4 (2012), 12. DOI:http://dx.doi.org/10.1145/2382553.2382554

Hsiao-Keng Jerry Chu. 1996. Zero-copy TCP in solaris. In *Proceedings of the 1996 USENIX Annual Technical Conference (ATC'96)*. 253–264.

Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert Tappan Morris, and Eddie Kohler. 2013. The scalable commutativity rule: Designing scalable software for multicore processors. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13)*. 1–17. DOI: http://dx.doi.org/10.1145/2517349.2522712

Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. 2013. 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)*. 33–48. DOI: http://dx.doi.org/10.1145/2517349.2522714

Jeffrey Dean and Luiz André Barroso. 2013. The tail at scale. *Commun. ACM* 56, 2 (2013), 74–80. DOI: http://dx.doi.org/10.1145/2408776.2408794

Christina Delimitrou and Christos Kozyrakis. 2014. Quality-of-service-aware scheduling in heterogeneous data centers with paragon. *IEEE Micro* 34, 3 (2014), 17–30. DOI: http://dx.doi.org/10.1109/MM.2014.7

Mihai Dobrescu, Norbert Egi, Katerina J. Argyraki, Byung-Gon Chun, Kevin R. Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. 2009. RouteBricks: Exploiting parallelism to scale software routers. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP'09)*. 15–28. DOI: http://dx.doi.org/10.1145/1629575.1629578

Aleksandar Dragojevic, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast remote memory. In *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI'14)*. 401–414.

Adam Dunkels. 2001. Design and implementation of the lwIP TCP/IP stack. *Swedish Inst. Comput. Sci.* 2 (2001), 77.

Dawson R. Engler, M. Frans Kaashoek, and James O'Toole. 1995. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP'95)*. 251–266. DOI: http://dx.doi.org/10.1145/224056.224076

Bin Fan, David G. Andersen, and Michael Kaminsky. 2013. 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)*. 371–384.

Mike Fisk and W. Feng. 2000. *Dynamic Adjustment of TCP Window Sizes*. Technical Report. Los Alamos Unclassified Report (LAUR) 00-3221, Los Alamos National Laboratory.

Sally Floyd and Van Jacobson. 1993. Random early detection gateways for congestion avoidance. *IEEE/ACM Trans. Netw.* 1, 4 (1993), 397–413. DOI: http://dx.doi.org/10.1109/90.251892

Robert Graham. 2013. The C10M Problem. Retrieved from http://c10m.robertgraham.com.

Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. 2012. MegaPipe: A new programming interface for scalable network I/O. In *Proceedings of the 10th Symposium on Operating System Design and Implementation (OSDI'12)*. 135–148.

Tim Harris, Martin Maas, and Virendra J. Marathe. 2014. Callisto: Co-scheduling parallel runtime systems. In *Proceedings of the 2014 EuroSys Conference*. 24:1–24:14. DOI: http://dx.doi.org/10.1145/2592798.2592807

Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy H. Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th Symposium on Networked Systems Design and Implementation (NSDI'11)*.

Chang-Hong Hsu, Yunqi Zhang, Michael A. Laurenzano, David Meisner, Thomas F. Wenisch, Jason Mars, Lingjia Tang, and Ronald G. Dreslinski. 2015. Adrenaline: Pinpointing and reining in tail queries with quick voltage boosting. In *Proceedings of the 21st IEEE Symposium on High-Performance Computer Architecture (HPCA'15)*. 271–282. DOI: http://dx.doi.org/10.1109/HPCA.2015.7056039

Ram Huggahalli, Ravi R. Iyer, and Scott Tetrick. 2005. Direct cache access for high bandwidth network I/O. In *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA'05)*. 50–59. DOI: http://dx.doi.org/10.1109/ISCA.2005.23

Intel Corp. 2013. Open Source Kernel Enhancements for Low Latency Sockets Using Busy Poll. Retrieved from http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/open-source-kernel-enhancements-paper.pdf.

Intel Corp. 2014a. Intel 82599 10 GbE Controller Datasheet. Retrieved from http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/82599-10-gbe-controller-datasheet.pdf.

Intel Corp. 2014b. Intel Ethernet Controller XL710 Datasheet. Retrieved from http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/xl710-10-40-controller-datasheet.pdf.

IX on GitHub 2016. The IX Project. https://github.com/ix-project/.

EunYoung Jeong, Shinae Woo, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyõungSoo Park. 2014a. mTCP source code release, v. of 2014-02-26. Retrieved from https://github.com/eunyoung14/mtcp.

Eunyoung Jeong, Shinae Woo, Muhammad Asim Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. 2014b. 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)*. 489–502.

Jithin Jose, Hari Subramoni, Miao Luo, Minjia Zhang, Jian Huang, Md. Wasi ur Rahman, Nusrat S. Islam, Xiangyong Ouyang, Hao Wang, Sayantan Sur, and Dhabaleswar K. Panda. 2011. Memcached design on high performance RDMA capable interconnects. In *Proceedings of the 2011 International Conference on Parallel Processing (ICPP'11)*. 743–752. DOI:http://dx.doi.org/10.1109/ICPP.2011.37

Rishi Kapoor, George Porter, Malveeka Tewari, Geoffrey M. Voelker, and Amin Vahdat. 2012. Chronos: Predictable low latency for data center applications. In *Proceedings of the 2012 ACM Symposium on Cloud Computing (SOCC'12)*. 9. DOI:http://dx.doi.org/10.1145/2391229.2391238

Wonyoung Kim, Meeta Sharma Gupta, Gu-Yeon Wei, and David M. Brooks. 2008. System level analysis of fast, per-core DVFS using on-chip switching regulators. In *Proceedings of the 14th IEEE Symposium on High-Performance Computer Architecture (HPCA'08)*. 123–134. DOI:http://dx.doi.org/10.1109/HPCA.2008.4658633

Avi Kivity. 2007. KVM: The linux virtual machine monitor. In *Proceedings of the 2007 Ottawa Linux Symposium (OLS'07)*. 225–230.

Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. 2000. The click modular router. *ACM Trans. Comput. Syst.* 18, 3 (2000), 263–297. DOI:http://dx.doi.org/10.1145/354871.354874

Andrew Krioukov, Prashanth Mohan, Sara Alspaugh, Laura Keys, David E. Culler, and Randy H. Katz. 2011. NapSAC: Design and implementation of a power-proportional web cluster. *Comput. Commun. Rev.* 41, 1 (2011), 102–108. DOI:http://dx.doi.org/10.1145/1925861.1925878

Jacob Leverich. 2014. Mutilate: High-Performance Memcached Load Generator. Retrieved from https://github.com/leverich/mutilate.

Jacob Leverich and Christos Kozyrakis. 2014. Reconciling high server utilization and sub-millisecond quality-of-service. In *Proceedings of the 2014 EuroSys Conference*. 4:1–4:14. DOI:http://dx.doi.org/10.1145/2592798.2592821

Philip Levis, Samuel Madden, David Gay, Joseph Polastre, Robert Szewczyk, Alec Woo, Eric A. Brewer, and David E. Culler. 2004. The emergence of networking abstractions and techniques in TinyOS. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation (NSDI'04)*. 1–14.

Jing Li, Kunal Agrawal, Sameh Elnikety, Yuxiong He, I.-Ting Angelina Lee, Chenyang Lu, and Kathryn S. McKinley. 2016. Work stealing for interactive services to meet target latency. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'16)*. 14:1–14:13. DOI:http://dx.doi.org/10.1145/2851141.2851151

Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. 2014. Tales of the tail: Hardware, OS, and application-level sources of tail latency. In *Proceedings of the 2014 ACM Symposium on Cloud Computing (SOCC'14)*. 9:1–9:14. DOI:http://dx.doi.org/10.1145/2670979.2670988

Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. 2014. MICA: A holistic approach to fast in-memory key-value storage. In *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI'14)*. 429–444.

David Lo, Liqun Cheng, Rama Govindaraju, Luiz Andre Barroso, and Christos Kozyrakis. 2014. Towards energy proportionality for large-scale latency-critical workloads. In *Proceedings of the 41st International Symposium on Computer Architecture (ISCA'14)*. 301–312. DOI:http://dx.doi.org/10.1109/ISCA.2014.6853237

David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2015. Heracles: Improving resource efficiency at scale. In *Proceedings of the 42nd International Symposium on Computer Architecture (ISCA'15)*. 450–462. DOI:http://dx.doi.org/10.1145/2749469.2749475

Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David J. Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. 2013. Unikernels: Library operating systems for the cloud. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XVIII'13)*. 461–472. DOI:http://dx.doi.org/10.1145/2451116.2451167

Ilias Marinos, Robert N. M. Watson, and Mark Handley. 2014. Network stack specialization for performance. In *Proceedings of the ACM SIGCOMM 2014 Conference*. 175–186. DOI:http://dx.doi.org/10.1145/2619239.2626311

Jason Mars, Lingjia Tang, Kevin Skadron, Mary Lou Soffa, and Robert Hundt. 2012. Increasing utilization in modern warehouse-scale computers using bubble-up. *IEEE Micro* 32, 3 (2012), 88–99. DOI:http://dx.doi.org/10.1109/MM.2012.22

Paul E. McKenney and John D. Slingwine. 1998. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*. 509–518.

David Meisner, Brian T. Gold, and Thomas F. Wenisch. 2011a. The PowerNap server architecture. *ACM Trans. Comput. Syst.* 29, 1 (2011), 3. DOI:http://dx.doi.org/10.1145/1925109.1925112

David Meisner, Christopher M. Sadler, Luiz André Barroso, Wolf-Dietrich Weber, and Thomas F. Wenisch. 2011b. Power management of online data-intensive services. In *Proceedings of the 38th International Symposium on Computer Architecture (ISCA'11)*. 319–330. DOI:http://dx.doi.org/10.1145/2000064.2000103

memcached 2014. memcached – A distributed memory object caching system. Retrieved from http://memcached.org.

Microsoft Corp. 2014. Receive Side Scaling. http://msdn.microsoft.com/library/windows/hardware/ff556942.aspx.

Christopher Mitchell, Yifeng Geng, and Jinyang Li. 2013. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store. In *Proceedings of the 2013 USENIX Annual Technical Conference (ATC'13)*. 103–114.

Jeffrey C. Mogul and K. K. Ramakrishnan. 1997. Eliminating receive livelock in an interrupt-driven kernel. *ACM Trans. Comput. Syst.* 15, 3 (1997), 217–252.

Luca Niccolini, Gianluca Iannaccone, Sylvia Ratnasamy, Jaideep Chandrashekar, and Luigi Rizzo. 2012. Building a power-proportional software router. In *Proceedings of the 2012 USENIX Annual Technical Conference (ATC'12)*. 89–100.

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

John K. Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen M. Rumble, Ryan Stutsman, and Stephen Yang. 2015. The RAMCloud storage system. *ACM Trans. Comput. Syst.* 33, 3 (2015), 7. DOI:http://dx.doi.org/10.1145/2806887

Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. 2000. IO-lite: A unified I/O buffering and caching system. *ACM Trans. Comput. Syst.* 18, 1 (2000), 37–66. DOI:http://dx.doi.org/10.1145/332799.332895

Aleksey Pesterev, Jacob Strauss, Nickolai Zeldovich, and Robert Tappan Morris. 2012. Improving network connection locality on multicore systems. In *Proceedings of the 2012 EuroSys Conference*. 337–350. DOI:http://dx.doi.org/10.1145/2168836.2168870

Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas E. Anderson, and Timothy Roscoe. 2016. Arrakis: The operating system is the control plane. *ACM Trans. Comput. Syst.* 33, 4 (2016), 11. DOI:http://dx.doi.org/10.1145/2812806

George Prekas, Adam Belay, Mia Primorac, Ana Klimovic, Samuel Grossman, Marios Kogias, Bernard Gütermann, Christos Kozyrakis, and Edouard Bugnion. 2016. *IX Open-source Version 1.0 – Deployment and Evaluation Guide*. Technical Report. EPFL Technical Report 218568.

George Prekas, Mia Primorac, Adam Belay, Christos Kozyrakis, and Edouard Bugnion. 2015. Energy proportionality and workload consolidation for latency-critical applications. In *Proceedings of the 2015 ACM Symposium on Cloud Computing (SOCC'15)*. 342–355. DOI:http://dx.doi.org/10.1145/2806777.2806848

Niels Provos and Nick Mathewson. 2003. libevent: an event notification library. Retrieved from http://libevent.org.

K. Ramakrishnan, S. Floyd, and D. Black. 2001. The Addition of Explicit Congestion Notification (ECN) to IP. IETF Network Working Group, RFC3168, September 2001. https://tools.ietf.org/html/rfc3168.

Luigi Rizzo. 2012. Revisiting network I/O APIs: The netmap framework. *Commun. ACM* 55, 3 (2012), 45–51. DOI:http://dx.doi.org/10.1145/2093548.2093565

Efraim Rotem, Alon Naveh, Avinash Ananthakrishnan, Eliezer Weissmann, and Doron Rajwan. 2012. Power-management architecture of the intel microarchitecture code-named sandy bridge. *IEEE Micro* 32, 2 (2012), 20–27. DOI:http://dx.doi.org/10.1109/MM.2012.12

Stephen M. Rumble, Diego Ongaro, Ryan Stutsman, Mendel Rosenblum, and John K. Ousterhout. 2011. It's time for low latency. In *Proceedings of the 13th Workshop on Hot Topics in Operating Systems (HotOS-XIII'11)*.

Constantine P. Sapuntzakis, David Brumley, Ramesh Chandra, Nickolai Zeldovich, Jim Chow, Monica S. Lam, and Mendel Rosenblum. 2003. Virtual appliances for deploying and maintaining software. In *Proceedings of the 17th Large Installation System Administration Conference (LISA'03)*. 181–194.

Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. 2013. Omega: Flexible, scalable schedulers for large compute clusters. In *Proceedings of the 2013 EuroSys Conference*. 351–364. DOI:http://dx.doi.org/10.1145/2465351.2465386

Quinn O. Snell, Armin R. Mikler, and John L. Gustafson. 1996. Netpipe: A network protocol independent performance evaluator. In *Proceedings of the IASTED International Conference on Intelligent Information Management and Systems*, Vol. 6.

Livio Soares and Michael Stumm. 2010. FlexSC: Flexible system call scheduling with exception-less system calls. In *Proceedings of the 9th Symposium on Operating System Design and Implementation (OSDI'10)*. 33–46.

Solarflare Communications. 2011. Introduction to OpenOnload: Building Application Transparency and Protocol Conformance into Application Acceleration Middleware. Retrieved from http://www.solarflare.com/content/userfiles/documents/solarflare_openonload_intropaper.pdf.

Ryan Stutsman and John K. Ousterhout. 2013. Toward common patterns for distributed, concurrent, fault-tolerant code. In *Proceedings of the 14th Workshop on Hot Topics in Operating Systems (HotOS-XIV'13)*.

Chandramohan A. Thekkath, Thu D. Nguyen, Evelyn Moy, and Edward D. Lazowska. 1993. Implementing network protocols at user level. In *Proceedings of the ACM SIGCOMM 1993 Conference*. 64–73. DOI:http://dx.doi.org/10.1145/166237.166244

Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L. Santoni, Fernando C. M. Martins, Andrew V. Anderson, Steven M. Bennett, Alain Kägi, Felix H. Leung, and Larry Smith. 2005. Intel virtualization technology. *IEEE Comput.* 38, 5 (2005), 48–56. DOI:http://dx.doi.org/10.1109/MC.2005.163

Guido Urdaneta, Guillaume Pierre, and Maarten van Steen. 2009. Wikipedia workload analysis for decentralized hosting. *Comput. Netw.* 53, 11 (2009), 1830–1845. DOI:http://dx.doi.org/10.1016/j.comnet.2009.02.019

George Varghese and Anthony Lauck. 1987. 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)*. 25–38. DOI:http://dx.doi.org/10.1145/41457.37504

Vijay Vasudevan, Amar Phanishayee, Hiral Shah, Elie Krevat, David G. Andersen, Gregory R. Ganger, Garth A. Gibson, and Brian Mueller. 2009. Safe and effective fine-grained TCP retransmissions for datacenter communication. In *Proceedings of the ACM SIGCOMM 2009 Conference*. 303–314. DOI:http://dx.doi.org/10.1145/1592568.1592604

Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with borg. In *Proceedings of the 2015 EuroSys Conference*. 18:1–18:17. DOI:http://dx.doi.org/10.1145/2741948.2741964

Werner Vogels. 2008. Beyond server consolidation. *ACM Queue* 6, 1 (2008), 20–26. DOI:http://dx.doi.org/10.1145/1348583.1348590

J. Robert von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric A. Brewer. 2003. Capriccio: Scalable threads for internet services. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*. 268–281. DOI:http://dx.doi.org/10.1145/945445.945471

Matt Welsh, David E. Culler, and Eric A. Brewer. 2001. SEDA: An architecture for well-conditioned, scalable internet services. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01)*. 230–243. DOI:http://dx.doi.org/10.1145/502034.502057

WhatsApp Inc. 2012. 1 million is so 2011. Retrieved from https://blog.whatsapp.com/index.php/2012/01/1-million-is-so-2011.

David A. Wheeler. 2001. SLOCCount, v2.26. Retrieved from http://www.dwheeler.com/sloccount/.

Hailong Yang, Alex D. Breslow, Jason Mars, and Lingjia Tang. 2013. Bubble-flux: Precise online QoS management for increased utilization in warehouse scale computers. In *Proceedings of the 40th International Symposium on Computer Architecture (ISCA'13)*. 607–618. DOI:http://dx.doi.org/10.1145/2485922.2485974

Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. 2013. CPI^2: CPU performance isolation for shared compute clusters. In *Proceedings of the 2013 EuroSys Conference*. 379–391. DOI:http://dx.doi.org/10.1145/2465351.2465388

Received June 2016; accepted September 2016