Research Statement

Adam Belay

I am fascinated by problems of scale and efficiency, and I like to undertake large software projects. While pursuing my Ph.D., I had the privilege of leading three major research projects, each involving multiple collaborators and students. The first was Dune, a platform that safely exposes privileged CPU features directly to applications.

A recurring theme in my research is to enable better use of hardware by making hardware features available at novel levels of abstraction. Using Dune, I built IX, a system that dramatically improves network I/O performance. I am also currently building Shinjuku, a system that fundamentally rethinks scheduling and significantly reduces context switching overheads. In general, my interests lie in operating systems and networking. I also like to focus on low-level software issues, often drawing inspiration from computer architecture.

The vision behind my research is to tackle the growing cost of software along two fronts: hardware usage and developer effort. Today, we waste a tremendous amount of both because of a disconnect between hardware capabilities and software implementations. The increasing number of abstractions in today’s software are making it difficult to exploit the power and flexibility of hardware. At the same time, many existing software interfaces are fundamentally limiting hardware scalability [7].

This gap between hardware and software is widening for two reasons. First, as Moore’s law draws to an end, hardware is still scaling in many respects but software no longer automatically gets faster as hardware improves. Increasingly, better performance requires deliberately modifying software. Yet scalability is already a difficult challenge; how will developers keep up with hundreds of cores and 100+ Gbps network interfaces? Second, emerging data center applications are also pushing software scalability to its limits. One consequence is that user-visible response times are frequently dominated by the slowest of a large number of internal responses [8]. Such architectures are making tail latency a critical constraint, and so far the most common way to control tail latency is to keep hardware utilization low.

Of course, if it were straightforward to modify software, then we could make progress through better algorithms and interface design alone. Unfortunately, large efficiency gains require revisiting many of the design assumptions baked into present-day systems, and to do so, we must contend with software bloat. For example, the Linux Kernel has exceeded 13 million lines of code, making it impractical for a small research team to re-architect Linux’s existing implementation around new principles. At the same time Linux exports over 300 system calls, supports hundreds of hardware devices, and provides many other specialized interfaces (such as the /proc and /sys file systems). Reimplementing these interfaces faithfully enough to support production workloads in a clean-slate kernel is equally intractable, making it hard to even evaluate the true benefits of a new research kernel, let alone realize them.

A unique advantage of my research is the ability to expose hardware across abstraction barriers without sacrificing the compatibility or convenience provided by an existing OS environment. This has been a powerful technique for bypassing software bloat, making it possible to tailor hardware use to application requirements. Of course, it can be difficult for developers to manage low-level hardware details directly, and developer productivity depends on having the right software abstractions. I have been able to strike a balance for two reasons. First, because my research maintains compatibility, it is possible to fall back on existing software on the slow path. Second, high-level APIs and abstractions (e.g. system calls, runtimes, standard libraries) can be improved by modifying their underlying implementations to use hardware directly.

To illustrate my approach, consider the case of sandboxing untrusted code. Google uses a technology called Native Client to safely run third-party native code directly in the Chrome browser [15]. Native Client employs software fault isolation (SFI), a powerful but complex sandboxing technique [14]. For this reason, Native Client requires a large amount of software infrastructure, including a modified compiler toolchain. For decades, however, CPUs have provided a native mechanism for confining code, namely ring protection [12]. Dune exposes ring protection directly to applications, making it possible to implement an analogous yet better performing sandbox in only a few thousand lines of code. I was able to use Dune to yield similar benefits across all of my research projects.

**Exposing CPU features to applications.** Dune [3] is a Linux kernel module that exposes kernel-level CPU features directly to applications. It enables developers to build custom library OSes [9] that run inside an ordinary Linux environment. It also includes an untrusted library, called libdune, that provides many useful abstractions for
application-specific hardware optimizations.

Dune is able to safely expose CPU features because instruction set architectures have been serendipitously modified to make virtual machines more efficient. In particular, running guest kernel code has been a major source of overhead and complexity. To reduce this cost, CPU vendors now provide extensions that enable direct execution of the majority of guest kernel instructions. In contrast to a normal hypervisor, Dune configures these extensions to provide a process abstraction instead of a machine abstraction. For example, instead of emulating the physical memory layout of a hardware platform, Dune exposes the address space of a Linux process.

There are several opportunities to improve application performance through better hardware access. For example, the Boehm garbage collector [6] relies on signals and mprotect to detect which memory has been modified since the last collection cycle. Hardware records such information in page table structures, but Linux does not provide a mechanism to expose it. Using Dune, we modified the Boehm garbage collector to use the page table directly, reducing garbage collection times by up to 40%. In general, Dune can be used to accelerate a wide range of virtual memory use cases, speeding up the Appel and Li microbenchmarks by 7× [2]. Dune also exposes several other hardware mechanisms, including ring protection, exceptions, interrupts, and CPU power management.

Low Latency and High Throughput Networking. IX is an OS, built on top of Dune, that is designed to make network I/O extremely efficient [4]. Fully utilizing 10 gigabit Ethernet is a considerable challenge, especially with complex network protocols like TCP. Previous efforts have bypassed the kernel, making network controller queues directly available to applications [10]. Through IX, we demonstrated that protection domain crossings are not a barrier to network performance and that it is possible to retain the protection advantages of a traditional OS security model. IX dramatically improves throughput and reduces latency. For example, IX increases memcached TCP request rates by 5× and reduces 99th percentile tail latency to levels below Linux’s median latency. Moreover, IX can saturate up to four 10 gigabit Ethernet links with a single CPU socket.

Network processing overheads are high for two reasons. First, packet processing has become convoluted, relying on a combination of interrupts and polling as well as several internal layers of scheduling. Second, network system call interfaces have remained basically unchanged since the original Berkeley Sockets, and are suboptimal for modern caches and multicore systems. IX revisits the design assumption that kernel-level packet buffers and network events should be decoupled from the application. Instead, IX introduces a clean-slate network system call API (compatibility is maintained in other system calls) that enables tight coordination between the application and the kernel. Specifically, it allows the application to access packet buffers directly, but in a way that does not sacrifice memory isolation. Moreover, the application is required to consume network events immediately as they are posted (they can still be delayed internally inside the application, if required), optimizing for data cache locality. IX also applies batching to improve instruction cache hit rates and prefetching effectiveness, but it does so adaptively to avoid impacting latency. Finally, IX avoids the need for memory synchronization, which can also lead to cache misses, by using flow-consistent hashing to ensure each flow is affinity-sized to a particular core.

Inspired by network middleboxes, IX embraces separation of control plane and data plane. Specifically, each application runs in a separate, isolated runtime that we refer to as a data-plane kernel. The control plane is an ordinary Linux process that is responsible for coarse-grained resource management, including assignment of CPU cores and load balancing of packets. Structuring the system in this fashion reduced complexity and made it easier to focus the data-plane kernel’s implementation on efficient networking. We experimented with a variety of control plane scheduling policies, demonstrating energy proportionality and low performance interference with background batch processing tasks [11].

Scheduling for High Utilization. Even if we could make NIC interactions completely free, the distribution of request sizes itself contains inherent variance. Moreover, it is often necessary to consolidate several workloads on a server to achieve high utilization, and each workload could have very different latency requirements. Given these challenges, we are building a second research OS, called Shinjuku, that is designed to provide low scheduling latency and predictable response times without sacrificing utilization. Shinjuku is currently in preparation for submission to OSDI.

In contemporary OSes, scheduling decisions are distributed among cores and driven by timer interrupts, making it difficult to reason about overall performance. In Shinjuku, we revisit the design assumption that scheduling should run locally on each core. Instead, Shinjuku uses a centralized scheduler that runs on a dedicated core per socket, handling all network requests and running with full knowledge of all jobs on the local CPU socket. While this approach may sound counter-intuitive, it has yielded some important advantages. Shinjuku’s complete view of the system makes it possible to apply a range of
different algorithms with provably good performance for their target workloads. Moreover, by having a core available to constantly poll the network controller, Shinjuku can respond to a high priority packet immediately without any interrupt mitigation delays. Also, Shinjuku can use centralized knowledge about task history to optimize job placement for cache locality.

In addition to more intelligent scheduling, Shinjuku also dramatically reduces context switching overheads. Our motivation is to make the cost of preemption low enough to allow for frequent scheduling adjustments. In particular, we have been experimenting with address-space identifiers, a feature not yet supported by Linux that allows the CPU to switch address spaces without invalidating the TLB.

**Future directions.** My long term goal is to make it easier for programmers to build systems that consistently perform well and scale to large numbers of cores and machines. Frustratingly, systems built with expressive, high-level programming languages tend to have not only more overhead, but also unpredictable latency and poor scalability. I have witnessed first hand the complexity and engineering resources needed to understand and manage performance anomalies created by Java virtual machines during my internship at Twitter. Some of these problems are internal to the runtime. For example, garbage collection can have a substantial negative impact on tail latency. However, there are also many problems created by poor interactions between language runtimes and kernels, especially when their ideas about scheduling, concurrency, and synchronization are not well aligned. Moreover, enforcing priorities among concurrent requests would be a valuable capability, but these types of scheduling constraints are difficult to map from high-level language interfaces to ad-hoc kernel tuning parameters.

One concrete step would be to revisit the question of kernel extensibility. I would like to build a kernel that is structured like Shinjuku, but allows applications to load arbitrary code extensions in order to make inline packet scheduling and core allocation decisions. Ideally, these extensions would be permitted to safely access application-level data structures from within the kernel to allow for tighter coordination with language runtimes. In addition to the security and isolation properties pioneered in SPIN [5], novel performance constraints would also have to be enforced. For example, each packet processing invocation might have a strict budget of only 1,000 cycles. Building upon scheduler activations [1], an extension approach could avoid most inter-processor interrupts and reduce overheads enough to permit applications to make scheduling decisions on the basis of individual packets. Applications could also choose to drop packets early, making them more resilient to receive live-lock and denial-of-service attacks.

As I/O throughput continues to increase, software-only solutions may not be fast enough to keep up with packet rates. Thus, I am also interested in designing programmable network controller hardware. Motivated by my experiences with Dune, I believe an ideal network controller would expose its capabilities directly to applications. Because future network controllers are likely to be combined with the CPU, it ought to be possible to enable direct sharing of hardware data structures with software, perhaps even via cache coherence. I am also interested in understanding how network protocols will have to evolve to be more amenable to low-level packet classification and filtering. In particular, TCP might prove to be too heavy-weight to permit the full benefit of network controller programmability.

Finally, I would like to find ways of further improving data center utilization. One possibility might be to make computation more fungible. Today, the latency of spinning up virtual machines and even containers is extremely high. If developers could be given nearly instantaneous access to vast quantities of idle CPU cores, then it might enable an interesting new class of applications. Also, developers notoriously overestimate the resources they will use when asked to provision machines in advance [13]. More tightly coupling the allocation of resources to the network events that trigger their use could be an effective way of avoiding much of this waste and complexity.

In summary, we do not have to accept poor software performance. Now that CPUs are energy constrained, single-threaded performance has reached a fundamental limit, but hardware is still improving dramatically in other dimensions. The consequence is that software has been placed squarely on the critical path of future gains in performance, and, as a community, we are tasked with the challenge of designing software that is more efficient, predictable, and scalable. I have shown that significant improvements are possible by giving applications more control over hardware and by designing interfaces that are aligned with hardware performance characteristics. I believe that an academic setting is the best place to make progress on this important issue. Building and testing new solutions to problems in computer systems should not require the vast resources of commercial engineering teams. By restructuring hardware and software to make access to important capabilities egalitarian, we can eliminate artificial engineering barriers and encourage future innovation.
References


