The Demikernel Datapath OS Architecture for Microsecond-scale Datacenter Systems

Irene Zhang[♥], Amanda Raybuck[♣], Pratyush Patel[*], Kirk Olynyk[♥], Jacob Nelson[♥], Omar S. Navarro Leija[★], Ashlie Martinez[*], Jing Liu[♠], Anna Kornfeld Simpson[*], Sujay Jayakar[∞], Pedro Henrique Penna[♥], Max Demoulin[★], Piali Choudhury[♥], Anirudh Badam[♥]

[♥]Microsoft Research, [♣]University of Texas at Austin, [*]University of Washington, [♠]University of Wisconsin Madison, [★]University of Pennsylvania, [∞]Zerowatt, Inc.

Abstract

Datacenter systems and I/O devices now run at single-digit microsecond latencies, requiring ns-scale operating systems. Traditional kernel-based operating systems impose an unaffordable overhead, so recent kernel-bypass OSes [73] and libraries [23] eliminate the OS kernel from the I/O datapath. However, none of these systems offer a general-purpose datapath OS replacement that meet the needs of µs-scale systems.

This paper proposes Demikernel, a flexible datapath OS and architecture designed for heterogenous kernel-bypass devices and µs-scale datacenter systems. We build two prototype Demikernel OSes and show that minimal effort is needed to port existing µs-scale systems. Once ported, Demikernel lets applications run across heterogenous kernel-bypass devices with ns-scale overheads and no code changes.

CCS Concepts • Software and its engineering → Operating systems.

Keywords operating system, kernel bypass, datacenters

ACM Reference Format:
Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S. Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, Pedro Henrique Penna, Max Demoulin, Piali Choudhury, Anirudh Badam. 2021. The Demikernel Datapath OS Architecture for Microsecond-scale Datacenter Systems. In *ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21), October 26–29, 2021, Virtual Event, Germany.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3477132.3483569

1 Overview

Datacenter I/O devices and systems are increasingly µs-*scale*: network round-trips, disk accesses and in-memory systems, like Redis [80], can achieve *single-digit microsecond latencies*. To avoid becoming a bottleneck, datapath systems software must operate at *sub-microsecond – or nanosecond – latencies*. To minimize latency, widely deployed kernel-bypass devices [78, 16] move legacy OS kernels to the control path and let µs-scale applications directly perform datapath I/O.

Kernel-bypass devices fundamentally change the traditional OS architecture: they eliminate the OS kernel from the I/O datapath without a clear replacement. Kernel-bypass devices offload OS protection (e.g., isolation, address translation) to safely offer user-level I/O and more capable devices implement some OS management (e.g., networking) to further reduce CPU usage. Existing kernel-bypass libraries [57, 23, 44] supply some missing OS components; however, none are a *general-purpose, portable datapath OS.*

Without a standard datapath architecture and general-purpose datapath OS, kernel-bypass is difficult for µs-scale applications to leverage. Programmers do not want to re-architect applications for different devices because they may not know in advance what will be available. New device features seemingly develop every year, and programmers cannot continuously re-design their applications to keep pace with these changes. Further, since datacenter servers are constantly upgraded, cloud providers (e.g., Microsoft Azure) deploy many generations of hardware concurrently.

Thus, µs-scale applications require a datapath architecture with a portable OS that implements common OS management: storage and networking stacks, memory management and CPU scheduling. Beyond supporting heterogenous devices with ns-scale latencies, a datapath OS must meet new needs of µs-scale applications. For example, zero-copy I/O is important for reducing latency, so µs-scale systems require an API with clear zero-copy I/O semantics and memory management that coordinates shared memory access between the application and OS. Likewise, µs-scale applications perform I/O every few microseconds, so fine-grained CPU multiplexing between application work and OS tasks is also critical.

This paper presents Demikernel, a flexible datapath OS and architecture designed for heterogenous kernel-bypass devices and µs-scale kernel-bypass datacenter systems. Demikernel defines: (1) new datapath OS management features for µs applications, (2) a new portable datapath API (PDPIX) and (3) a flexible datapath architecture for minimizing latency across

SOSP '21, October 26–29, 2021, Virtual Event, Germany

ACM ISBN 978-1-4503-8709-5/21/10.
https://doi.org/10.1145/3477132.3483569

heterogenous devices. Demikernel datapath OSes run with a legacy controlplane kernel (e.g., Linux or Windows) and consist of interchangeable library OSes with the same API, OS management features and architecture. Each library OS is device-specific: it offloads to the kernel-bypass device when possible and implements remaining OS management in a user-space library. These libOSes aim to simplify the development of µs-scale datacenter systems across heterogenous kernel-bypass devices with while minimizing OS overheads.

Demikernel follows a trend away from kernel-oriented OSes to library-oriented datapath OSes, motivated by the CPU bottleneck caused by increasingly efficient I/O devices. It is not designed for systems that benefit from directly accessing kernel-bypass hardware (e.g., HPC [45], software middleboxes [90, 72], RDMA storage systems [17, 98, 7, 101]) because it imposes a common API that hides more complex device features (e.g., one-sided RDMA).

This paper describes two prototype Demikernel datapath OSes for Linux and Windows. Our implementation is largely in Rust, leveraging its memory safety benefits within the datapath OS stack. We also describe the design of a new zero-copy, ns-scale TCP stack and kernel-bypass-aware memory allocator. Our evaluation found it easy to build and port Demikernel applications with I/O processing latencies of ≈50ns per I/O, and a 17-26% peak throughput overhead, compared to directly using kernel-bypass APIs.

2 Demikernel Datapath OS Requirements

Modern kernel-bypass devices, OSes and libraries eliminate the OS kernel from the I/O datapath but do not replace all of its functionality, leaving a gap in the kernel-bypass OS architecture. This gap exposes a key question: what is the right datapath OS replacement for µs-scale systems? This section details the requirements of µs-scale systems and heterogenous kernel-bypass devices that motivate Demikernel's design.

2.1 Support Heterogenous OS Offloads

As shown in Figure 1, today's datapath architectures are *ad hoc*: existing kernel-bypass libraries [34, 73] offer different OS features atop specific kernel-bypass devices. Portability is challenging because different device offload different OS features. For example, DPDK provides a low-level, raw NIC interface, while RDMA implements a network protocol with congestion control and ordered, reliable transmission. Thus, systems that work with DPDK implement a full networking stack, which is unnecessary for systems using RDMA.

This heterogeneity stems from a fundamental trade-off that hardware designers have long struggled with [50, 66, 10] – offloading more features improves performance but increases device complexity – which only becomes worse as recent research proposes increasingly complex offloads [52, 41, 86].For example, DPDK is more general and widely usable, but RDMA achieves the lowest CPU and latency overheads for µs-scale systems within the datacenter, so any datapath

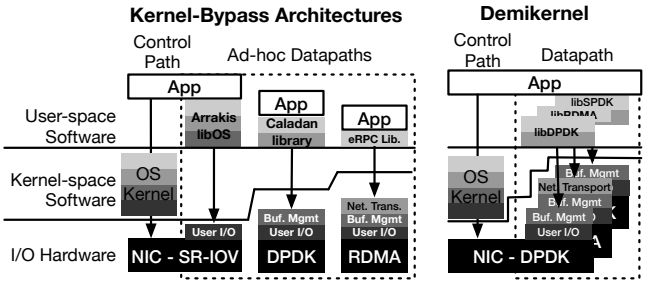


Figure 1. *Example kernel-bypass architectures.* Unlike the Demikernel architecture (right), Arrakis [73], Caladan [23] and eRPC [8]'s architectures do not flexibly support heterogenous devices.

OS must portably support both. Future NICs may introduce other trade-offs, so Demikernel's design must *flexibly accommodate heterogenous kernel-bypass devices with varied hardware capabilities and OS offloads.*

2.2 Coordinate Zero-Copy Memory Access

Zero-copy I/O is critical for minimizing latency; however, it requires coordinating memory access carefully across the I/O device, stack and application. Kernel-bypass zero-copy I/O requires two types of memory access coordination, which are both difficult for programmers to manage and not explicitly managed by existing kernel-bypass OSes.

First, kernel-bypass requires the I/O device's IOMMU (or a user-level device driver) to perform address translation, which requires coordination with the CPU's IOMMU and TLB. To avoid page faults and ensure address mappings stay fixed during ongoing I/O, kernel-bypass devices require designated *DMA-capable* memory, which is pinned in the OS kernel. This designation works differently across devices; for example, RDMA uses explicit registration of DMA-capable memory while DPDK and SPDK use a separate pool-based memory allocator backed with huge pages for DMA-capable memory. Thus, a programmer must know in advance what memory to use for I/O, which is not always possible and can be complex to deduce, and use the appropriate designation for the available device.

The second form of coordination concerns actual access to the I/O memory buffers. Since the I/O stack and kernel-bypass device work directly with application memory, the programmer must not free or modify that memory while in use for I/O. This coordination goes beyond simply waiting until the I/O has completed. For example, the TCP stack might send the memory to the NIC, but then network loses the packet. If the application modified or freed the memory buffer in the meantime, the TCP stack has no way to retransmit it. As a result, the application must also coordinate with the TCP stack when freeing memory. This coordination becomes increasingly complex, especially in asynchronous or multithreaded distributed systems. Given this complexity, Demikernel's design must *portably manage complex zero-copy I/O coordination between applications and OS components*.

2.3 Multiplex and Schedule the CPU at µs-scale

µs-scale datacenter systems commonly perform I/O every few microseconds; thus, a datapath OS must be able to multiplex and schedule I/O processing and application work at similar speeds. Existing kernel-level abstractions, like processes and threads, are too coarse-grained for µs-scale scheduling because they consume entire cores for hundreds of microseconds. As a result, kernel-bypass systems lack a general-purpose scheduling abstraction.

Recent user-level schedulers [9, 75] allocate application workers on a µs-scale per-I/O basis; however, they still use coarse-grained abstractions for OS work (e.g., whole threads [23] or cores [30]). Some go a step further and take a microkernel approach, separating OS services into another process [57] or ring [3] for better security.

µs-scale RDMA systems commonly interleave I/O and application request processing. This design makes scheduling implicit instead of explicit: a datapath OS has no way to control the balance of CPU cycles allocated to the application versus datapath I/O processing. For example, both FaRM [17] and Pilaf [64] always immediately perform I/O processing when messages arrive, even given higher priority tasks (e.g., allocating new buffer space for incoming packets that might otherwise be dropped).

None of these systems are ideal because their scheduling decisions remain distributed, either between the kernel and user-level scheduler (for DPDK systems) or across the code (for RDMA systems). Recent work, like eRPC [34], has shown that multiplexing application work and datapath OS tasks on a single thread is required to achieve ns-scale overheads. Thus, Demikernel's design requires *a* µs-*scale scheduling abstraction and scheduler.*

3 Demikernel Overview and Approach

Demikernel is the first datapath OS that meets the requirements of µs-scale applications and kernel-bypass devices. It has new OS features and a new portable datapath API, which are programmer-visible, and a new OS architecture and design, which is not visible to programmers. This section gives an overview of Demikernel's approach, the next describes the programmer-visible features and API, while Section 5 details the Demikernel (lib)OS architecture and design.

3.1 Design Goals

While meeting the requirements detailed in the previous section, Demikernel has three high-level design goals:

- *Simplify* µs-*scale kernel-bypass system development.* Demikernel must offer OS management that meets common needs for µs-scale applications and kernel-bypass devices.
- *Offer portability across heterogenous devices.* Demikernel should let applications run across multiple types of kernel-bypass devices (e.g., RDMA and DPDK) and virtualized environments with no code changes.
- *Achieve nanosecond-scale latency overheads.* Demikernel datapath OSes have a per-I/O budget of less than 1µs for I/O processing and other OS services.

3.2 System Model and Assumptions

Demikernel relies on popular kernel-bypass devices, including RDMA [61] and DPDK [16] NICs and SPDK disks [88], but also accommodates future programmable devices [60, 58, 69]. We assume Demikernel datapath OSes run in the same process and thread as the application, so they mutually trust each other and any isolation and protection are offered by the control path kernel or kernel-bypass device. These assumptions are safe in the datacenter where applications typically bring their own libraries and OS, and the datacenter operator enforces isolation using hardware.

Demikernel makes it possible to send application memory directly over the network, so applications must carefully consider the location of sensitive data. If necessary, this capability can be turned off. Other techniques, like information flow control or verification could also be leveraged to ensure the safety and security of application memory.

To minimize datapath latency, Demikernel uses cooperative scheduling, so applications must run in a tight I/O processing loop (i.e., enter the datapath libOS to perform I/O at least once every millisecond). Failing to allocate cycles to the datapath OS can cause I/O failures (e.g., packets to be dropped and acks/retries not sent). Other control path work (e.g., background logging) can run on a separate thread or core and go through the legacy OS kernel. Our prototypes currently focus on independently scheduling single CPU cores, relying on hardware support for multi-core scheduling [27]; however, the architecture can fit more complex scheduling algorithms [23, 71]. A companion paper [15] proposes a new kernel-bypass request scheduler that leverages Demikernel's understanding of application semantics to provide better tail latency for µs-scale workloads with widely varying request execution times.

3.3 Demikernel Approach

This section summarizes the new features of Demikernel's design – both internal and external – to meet the needs detailed in the previous section.

A portable datapath API and flexible OS architecture. Demikernel tackles heterogenous kernel-bypass offloads with a new portable datapath API and flexible OS architecture. Demikernel takes a library OS approach by treating the kernel-bypass hardware the datapath kernel and accommodating heterogenous kernel-bypass devices with *interchangeable* library OSes. As Figure 1 shows, the Demikernel kernel-bypass architecture extends the kernel-bypass architecture with a flexible datapath architecture. Each Demikernel datapath OS works with a legacy control path OS kernel and consists of several interchangeable datapath *library OSes* (libOSes) that implement a new high-level datapath API, called PDPIX.

PDPIX extends the standard POSIX API to better accommodate µs-scale kernel-bypass I/O. Microsecond kernel-bypass systems are I/O-oriented: they spend most time and memory processing I/O. Thus, PDPIX centers around an *I/O queue* abstraction that makes I/O explicit: it lets µs-scale applications submit entire I/O requests, eliminating latency issues with POSIX's pipe-based I/O abstraction.

To minimize latency, Demikernel libOSes offload OS features to the device when possible and implement the remaining features in software. For example, our RDMA libOS relies on the RDMA NIC for ordered, reliable delivery, while the DPDK libOS implements it in software. Demikernel libOSes have different implementations, and can even be written in different languages; however, they share the same OS features, architecture and core design.

A DMA-capable heap, use-after-free protection. Demikernel provides three new external OS features to simplify zero-copy memory coordination: a portable API with clear semantics for I/O memory buffer ownership, (2) a zero-copy, DMA-capable heap and (3) use-after-free (UAF) protection for zero-copy I/O buffers. Unlike POSIX, PDPIX defines clear zero-copy I/O semantics: applications pass ownership to the Demikernel datapath OS when invoking I/O and do not receive ownership back until the I/O completes.

The DMA-capable heap eliminates the need for programmers to designate I/O memory. Demikernel libOSes replace the application's memory allocator to back the heap with DMA-capable memory in a device-specific way. For example, the RDMA libOS's memory allocator registers heap memory transparently for RDMA on the first I/O access, while the DPDK libOS's allocator backs the application's heap with memory from the DPDK memory allocator.

Demikernel libOS allocators also provide UAF protection. It guarantees that shared, in-use zero-copy I/O buffers are not freed until both the application and datapath OS explicitly free them. It simplifies zero-copy coordination by preventing applications from accidentally freeing memory buffers in use for I/O processing (e.g., TCP retries). However, UAF protection does not keep applications from modifying in-use buffers because there is no affordable way to Demikernel datapath OSes to offer write-protection.

Leveraging the memory allocator lets the datapath OS control memory used to back the heap and when objects are freed. However, it is a trade-off: though the allocator has insight into the application (e.g., object sizes), the design requires all applications to use the Demikernel allocator.

Coroutines and µs-scale CPU scheduling. Kernel-bypass scheduling commonly happens on a per-I/O basis; however, the POSIX API is poorly suited to this use. `epoll` and `select` have a well-known "thundering herd" issue [56]: when the socket is shared, it is impossible to deliver events to precisely one worker. Thus, PDPIX introduces a new asynchronous I/O API, called `wait`, which lets applications workers wait on specific I/O requests and the datapath OSes explicitly assign I/O requests to workers.

Table 1. *Demikernel datapath OS services.* We compare Demikernel to kernel-based POSIX implementations and kernel-bypass programming APIs and libraries, including RDMA `ib_verbs` [78], DPDK [16] (SPDK [88]), recent networking [30, 20, 70, 42], storage [32, 44, 81] and scheduling [71, 33, 75, 23] libraries. ● = full support, ◑ = partial support, none = no support.

	Demikernel Datapath OS Services	POSIX	RDMA	DPDK	Net lib	Stor lib	Sched	Demik
I/O Stack	I1. Portable high-level API	●	◑	◑	◑	◑		●
	I2. Microsecond Net Stack		◑		●		◑	●
	I3. Microsecond Storage Stack					●		●
Schedule	C1. Alloc CPU to app and I/O	●		◑	◑	◑	●	●
	C2. Alloc I/O req to app workers	◑					●	●
	C3. App request scheduling API		◑				◑	●
Memory	M1. Mem ownership semantics		●		◑			●
	M2. DMA-capable heap	●		●	◑			●
	M3. Use-after-free protection							●

For µs-scale CPU multiplexing, Demikernel uses *coroutines* to encapsulate both OS and application computation. Coroutines are lightweight, have low-cost context switches and are well-suited for the state-machine-based asynchronous event handling that I/O stacks commonly require. We chose coroutines over user-level threads (e.g., Caladan's green threads [23]), which can perform equally well, because coroutines encapsulate state for each task, removing the need for global state management. For example, Demikernel's TCP stack uses one coroutine per TCP connection for retransmissions, which keeps the relevant TCP state.

Every libOS has a centralized coroutine scheduler, optimized for the kernel-bypass device. Since interrupts are unaffordable at ns-scale [33, 15], Demikernel coroutines are *cooperative*: they typically yield after a few microseconds or less. Traditional coroutines typically work by polling: the scheduler runs every coroutine to check for progress. However, we found polling to be unaffordable at ns-scale since large numbers of coroutines are blocked on infrequent I/O events (e.g., a packet for the TCP connection arrives). Thus, Demikernel coroutines are also *blockable*. The scheduler separates *runnable* and *blocked* coroutines and moves blocked ones to the runnable queue only after the event occurs.

4 Demikernel Datapath OS Features and API

Demikernel offers new OS features to meet µs-scale application requirements. This section describes Demikernel from a programmer's perspective, including PDPIX.

4.1 Demikernel Datapath OS Feature Overview

Table 1 summarizes and compares Demikernel's OS feature support to existing kernel-bypass APIs [62, 16] and libraries [30, 23, 44]. Unlike existing kernel-bypass systems,

the Demikernel datapath OS offers a portable I/O API for kernel-bypass devices with high-level abstractions, like sockets and files (Table 1:I1). For each device type, it also implements (I2) a µs-scale networking stack with features like ordered, reliable messaging, congestion control and flow control and (I3) a µs-scale storage stack with disk block allocation and data organization.

Demikernel provides two types of CPU scheduling: (C1) allocating CPU cycles between libOS I/O processing and application workers, and (C2) allocating I/O requests among application workers. This paper focuses on C1, as C2 is a well-studied topic [23]. Our companion paper, Perséphone [15], explores how to leverage Demikernel for better I/O request scheduling at microsecond timescales. To better support kernel-bypass schedulers, we replace `epoll` with a new API (C3) that explicitly supports I/O request scheduling.

kernel-bypass applications require zero-copy I/O to make the best use of limited CPU cycles. To better this task, Demikernel offers (M1) a zero-copy I/O API with clear memory ownership semantics between the application, the libOS and the I/O device, and (M2) makes the entire application heap transparently DMA-capable without explicit, device-specific registration. Finally, Demikernel gives (M3) use-after-free protection, which, together with its other features, lets applications that do not update in place, like Redis, leverage zero-copy I/O with no application changes. Combined with its zero-copy network and storage stacks and fine-grained CPU multiplexing, *Demikernel supports single-core run-to-completion for a request (e.g., Redis PUT) from the NIC to the application to disk and back without copies.*

4.2 PDPIX: A Portable Datapath API

Demikernel extends POSIX with the *portable datapath interface* (PDPIX). To minimize changes to existing µs-scale applications, PDPIX limits POSIX changes to ones that minimize overheads or better support kernel-bypass I/O. PDPIX system calls go to the datapath OS and no longer require a kernel crossing (and thus we call them PDPIX library calls or *libcalls*). PDPIX is queue-oriented, not file-oriented; thus, system calls that return a file descriptor in POSIX return a queue descriptor in PDPIX.

I/O Queues. To reduce application changes, we chose to leave the socket, pipe and file abstractions in place. For example, PDPIX does not modify the POSIX `listen`/`accept` interface for accepting network connections; however, `accept` now returns a queue descriptor, instead of a file descriptor, through which the application can accept incoming connections. `queue()` creates a light-weight in-memory queue, similar to a Go channel [25].

While these library calls seem like control path operations, they interact with I/O and thus are implemented in the datapath OS. For example, incoming connections arrive as network I/O and must be processed efficiently by the datapath networking stack. Likewise, sockets remain on the datapath because they create I/O queues associated with network connections that the datapath OS needs to dispatch incoming packets.

Network and Storage I/O. `push` and `pop` are datapath operations for submitting and receiving I/O operations, respectively. To avoid unnecessary buffering and poor tail latencies, these libcalls take a scatter-gather array of memory pointers. They are intended to be a complete I/O operation, so the datapath OS can take a fast path and immediately issue or return the I/O if possible. For example, unlike the POSIX `write` operation, Demikernel immediately attempts to submit I/O after a `push`. Both `push` and `pop` are non-blocking and return a *qtoken* indicating their asynchronous result. Applications use the qtoken to fetch the completion when the operation has been successfully processed via the `wait_*` library calls. Naturally, an application can simulate a blocking library call by calling the operation and immediately waiting on the qtoken.

Memory. Applications do not allocate buffers for incoming data; instead, `pop` and `wait_*` return scatter-gather arrays with pointers to memory allocated in the application's DMA-capable heap. The application receives memory ownership of the buffers and frees them when no longer needed.

PDPIX requires that all I/O must be from the DMA-capable heap (e.g., not on the stack). On `push`, the application grants ownership of scatter-gather buffers to the Demikernel datapath OS and receives it back on completion. Use-after-free protection guarantees that I/O buffers are not be freed until both the application and datapath OS free them.

UAF protection does not offer write-protection; the application must respect `push` semantics and not modify the buffer until the qtoken returns. We chose to offer only UAF protection as there is no low cost way for Demikernel to provide full write-protection. Thus, UAF protection is a compromise: it captures a common programming pattern but does not eliminate all coordination. However, applications that do not update in place (i.e., their memory is immutable, like Redis's keys and values) require no additional code to support zero-copy I/O coordination.

Scheduling. PDPIX replaces `epoll` with the asynchronous `wait_*` call. The basic `wait` blocks on a single qtoken; `wait_any` provides functionality similar to `select` or `epoll`, and `wait_all` blocks until all operations complete. This abstraction solves two major issues with POSIX `epoll`: (1) `wait` directly returns the data from the operation so the application can begin processing immediately, and (2) assuming each application worker waits on a separate qtoken, `wait` wakes only one worker on each I/O completion. Despite these semantic changes, we found it easy to replace an `epoll` loop with `wait_any`. However, `wait_*` is a low-level API, so we hope to eventually implement libraries, like libevent [54], to reduce application changes.

```
// Queue creation and management
int qd = socket(...);
int err = listen(int qd, ...);
int err = bind(int qd, ...);
int qd = accept(int qd, ...);
int err = connect(int qd, ...);
int err = close(int qd);
int qd = queue();
int qd = open(...);
int qd = creat(...);
int err = lseek(int qd, ...);
int err = truncate(int qd, ...);
```

```
// I/O processing, notifcation and memory calls
qtoken qt = push(int qd, const sgarray &sga);
qtoken qt = pop(int qd, sgarray *sga);
int ret = wait(qtoken qt, sgarray *sga);
int ret = wait_any(qtoken *qts,
                   size_t num_qts,
                   qevent **qevs,
                   size_t *num_qevs,
                   int timeout);
int ret = wait_all(qtoken *qts, size_t num_qts,
                   qevent **qevs, int timeout);
void *dma_ptr = malloc(size_t size);
free(void *dma_ptr);
```

Figure 2. *Demikernel PDPIX library call API.* PDPIX retains features of the POSIX interface – `...` represents unchanged arguments – with three key changes. To avoid unnecessary buffering on the I/O datapath, PDPIX is queue-oriented and lets applications submit complete I/O operations. To support zero-copy I/O, PDPIX queue operations define clear zero-copy I/O memory ownership semantics. Finally, PDPIX replaces `epoll` with `wait_*` to let libOSes explicitly assign I/O to workers.

5 Demikernel Datapath Library OS Design

Figure 3 shows the Demikernel datapath OS architecture: each OS consists of interchangeable library OSes that run on different kernel-bypass devices with a legacy kernel. While each library OS supports a different kernel-bypass device on a different legacy kernel, they share a common architecture and design, described in this section.

5.1 Design Overview

Each Demikernel libOS supports a single kernel-bypass I/O device type (e.g., DPDK, RDMA, SPDK) and consists of an I/O processing stack for the I/O device, a libOS-specific memory allocator and a centralized coroutine scheduler. To support both networking and storage, we integrate libOSes into a single library for both devices (e.g., RDMAxSPDK).

We implemented the bulk of our library OS code in Rust. We initially prototyped several libOSes in C++; however, we found that Rust performs competitively with C++ and achieves ns-scale latencies while offering additional benefits. First, Rust enforces memory safety through language features and its compiler. Though our libOSes use unsafe code to bind to C/C++ kernel-bypass libraries and applications,

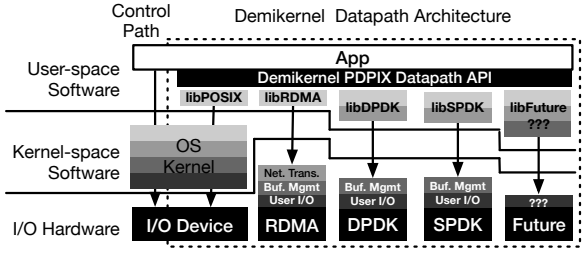


Figure 3. *Demikernel kernel-bypass architecture.* Demikernel accommodates heterogenous kernel-bypass devices, including potential future hardware, with a flexible library OS-based datapath architecture.We include a libOS that goes through the OS kernel for development and debugging.

Rust ensures memory safety internally within our libOSes. We also appreciated Rust's improved build system with portability across platforms, compared to the difficulties that we encountered with CMake. Finally, Rust has excellent support for co-routines, which we are being actively developing, letting us use language features to implement our scheduling abstraction and potentially contribute back to the Rust community. The primary downside to using Rust is the need for many cross-language bindings as kernel-bypass interfaces and µs-scale applications are still largely written in C/C++.

Each libOS has a memory allocator that allocates or registers DMA-capable memory and performs reference counting for UAF protection. Using the memory allocator for memory management is a trade-off that provides good insight into application memory but requires that applications use our memory allocator. Other designs are possible; for example, having the libOSes perform packet-based refcounting. Our prototype Demikernel libOSes use Hoard [4], a popular memory allocator that is easily extensible using C++ templates [5]. We intend to integrate a more modern memory allocator (i.e., mimalloc [46]) in the future.

Demikernel libOSes use Rust's async/await language features [85] to implement asynchronous I/O processing within coroutines. Rust leverages support for generators to compile imperative code into state machines with a transition function. The Rust compiler does not directly save registers and swap stacks; it compiles coroutines down to regular function calls with values "on the stack" stored directly in the state machine [55]. This crucial benefit of using Rust makes a coroutine context switch lightweight and fast (≈12 cycles in our Rust prototype) and helps our I/O stacks avoid a real context switch on the critical path. While Rust's language interface and compiler support for writing coroutines is well-defined, Rust does not currently have a coroutine runtime. Thus, we implement a simple coroutine runtime and scheduler within each libOS

that optimizes for the amount of I/O processing that each kernel-bypass devices requires.

5.2 I/O Processing

The primary job of Demikernel libOSes is µs-scale I/O processing. Every libOS has an I/O stack with this processing flow but different amounts and types of I/O processing and different background tasks (e.g., sending acks). As noted previously, Demikernel I/O stacks minimize latency by *polling*, which is CPU-intensive but critical for microsecond latency. Each stack uses a fast-path I/O coroutine to poll a single hardware interface (e.g., RDMA `poll_cq`, DPDK `rte_rx_burst`), then multiplexes sockets, connections, files, etc. across them.

Taking inspiration from TAS [39], Demikernel I/O stacks optimize for an *error-free fast-path* (e.g., packets arrive in order, send windows are open), which is the common case in the datacenter [37, 34]. Unlike TAS, Demikernel I/O stacks share application threads and aim for *run-to-completion*: the fast-path coroutine processes the incoming data, finds the blocked qtoken, schedules the application coroutine and processes any outgoing messages before moving on to the next I/O. Likewise, Demikernel I/O stacks inline outgoing I/O processing in `push` (in the application coroutine) and submit I/O (to the asynchronous hardware I/O API) in the error-free case. Although a coroutine context switch occurs between the fast-path and application coroutines, it does not interrupt run-to-completion because Rust compiles coroutine switches to a function call.

Figure 4 shows normal case I/O processing for the DPDK TCP stack, but the same applies to all libOS I/O stacks. To begin, the application: (1) calls `pop` to ask for incoming data. If nothing is pending, the libOS (2) allocates and returns a qtoken to the application, which calls `wait` with the qtoken. The libOS (3) allocates a blocked coroutine for each qtoken in `wait`, then immediately yields to the coroutine scheduler. We do not allocate the coroutine for each queue token unless the application calls `wait` to indicate a waiting worker. If the scheduler has no other work, it runs the fast-path coroutine, which (4) polls DPDK's `rte_rx_burst` for incoming packets. If the fast-path coroutine finds a packet, then it (5) processes the packet immediately (if error-free), signals a blocked coroutine for the TCP connection and yields.

The scheduler runs the unblocked application coroutine, which (6) returns the incoming request to the application worker, which in turn (7) processes the request and pushes any response. On the error-free path, the libOS (8) inlines the outgoing packet processing and immediately submits the I/O to DPDK's `rte_tx_burst`, then (9) returns a qtoken to indicate when the `push` completes and the application will regain ownership of the memory buffers. To restart the loop, the application (1) calls `pop` on the queue to get the next I/O and then calls `wait` with the new qtoken. To serve multiple

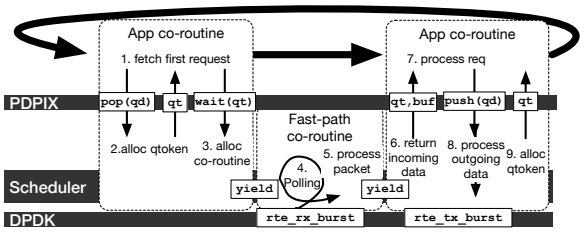


Figure 4. *Example Demikernel I/O processing flow for DPDK.*

connections, the application calls `pop` on all connections and then `wait_any` with the returned qtokens.

We use DPDK as an example; however, the same flow works for any asynchronous kernel-bypass I/O API. The fast path coroutine yields after every *n* polls to let other I/O stacks and background work run. For non-common-case scenarios, the fast-path coroutine unblocks another coroutine and yields.

5.3 Memory Management

Each Demikernel libOS uses a device-specific, modified Hoard for memory management. Hoard is a pool-based memory allocator; Hoard memory pools are called *superblocks*, which each hold fixed-size *memory objects*. We use superblocks to manage DMA-capable memory and reference counting: the superblock header holds reference counts and meta-data for DMA-capable memory (e.g., RDMA `rkeys`).

As noted in Section 2, different kernel-bypass devices have different requirements for DMA-capable memory. For example, RDMA requires memory registration and a memory region-specific `rkey` on every I/O, so Catmint's memory allocator provides a `get_rkey` interface to retrieve the rkey on every I/O. `get_rkey(void *)` registers an entire superblock on the first invokation and stores the rkey in the superblock header. Likewise, Catnip and Cattree allocate every superblock with memory from the DPDK mempool for the DMA-capable heap.

For UAF protection, all Demikernel allocators provide a simple reference counting interface: `inc_ref(void *)` and `dec_ref(void *)`. Note that these interfaces are not part of PDPIX but are internal to Demikernel libOSes. The libOS I/O stack calls `inc_ref` when issuing I/O on a memory buffer and `dec_ref` when done with the buffer. As noted earlier, Demikernel libOSes may have to hold references for a long time; for example, the TCP stack can safely `dec_ref` only after the receiver has acked the packet.

Hoard keeps free objects in a LIFO linked list with the head pointer in the superblock header, which Demikernel amends with a bitmap of per-object reference counts. To minimize overhead, we use a single bit per-object, representing one reference from the application and one from the libOS. If the libOS has more than one reference to the object (e.g., the object is used for multiple I/Os), it must keep a reference table to track when it is safe to `dec_ref`.

Zero-copy I/O offers a significant performance improvement only for buffers over 1 kB in size, so Demikernel OSes

perform zero-copy I/O only for buffers over that size. Hoard superblocks make it easy to limit reference counting and kernel-bypass DMA support to superblocks holding objects larger than 1 kB, minimizing additional meta-data.

5.4 Coroutine Scheduler

A Demikernel libOS has three coroutine types, reflecting common CPU consumers: (1) a *fast-path I/O processing coroutine* for each I/O stack that polls for I/O and performs fast-path I/O processing, (2) several *background coroutines* for other I/O stack work (e.g., managing TCP send windows), and (3) one *application coroutine* per blocked qtoken, which runs an application worker to process a single request. Generally, each libOS scheduler gives priority to runnable application coroutines, and then to background coroutines and the fast-path coroutine, which is always runnable, in a FIFO manner. Demikernel libOSes are single threaded; thus, each scheduler runs one coroutine at a time. We expect the coroutine design will scale to more cores. However, Demikernel libOSes will need to be carefully designed to avoid shared state across cores, so we do not yet know if this will be a major limitation.

Demikernel schedulers offer a yield interface that lets coroutines express whether they are blocked and provide a readiness flag for the unblocking event. A coroutine can be in one of three states: running, runnable or blocked. To separate runnable and blocked coroutines, Demikernel schedulers maintain a readiness bit per coroutine. Following Rust's `Future` trait's [97] design, coroutines that block on an event (e.g., a timer, receiving on a connection), stash a pointer to a readiness flag for the event. Another coroutine triggers the event (e.g., by receiving a packet on the connection), sees the pointer, and sets the stashed readiness bit, signaling to the scheduler that the blocked coroutine is now runnable.

Implementing a ns-scale scheduler is challenging: it may manage hundreds or thousands of coroutines and have only hundreds of cycles to find the next to run. For example, heap allocations are unaffordable on the datapath, so the scheduler maintains a list of *waker blocks* that contains the readiness bit for 64 different coroutines in a bitset. To make ns-scale scheduling decisions, the scheduler must efficiently iterate over all set bits in each waker block to find runnable coroutines. We use Lemire's algorithm [47], which uses x86's `tzcnt` instruction to efficiently skip over unset bits. A microbenchmark shows that the scheduler can context switch between an empty yielding coroutine and find another runnable coroutine in 12 cycles.

5.5 Network and Storage LibOS Integration

A key Demikernel goal is fine-grained CPU multiplexing of networking and storage I/O processing with application processing and integrated zero-copy memory coordination across all three. For example, Demikernel lets Redis receive a PUT request from the network, checkpoint it to disk, and respond to the client without copies or thread context switches. Current kernel-bypass libraries do not achieve this goal because they separate I/O and application processing [30, 39] or do not support applications [43, 28, 96].

To support network and storage devices together, Demikernel integrates its network and storage libOSes. Doing so is challenging because not all kernel-bypass devices work well together. For example, though DPDK and SPDK work cooperatively, RDMA and SPDK were not designed to interact. SPDK shares the DPDK memory allocator, so initializing it creates a DPDK instance, which Catnip×Cattree shares betwen the networking and storage stacks. This automatic initialization creates a problem for RDMA integration because the DPDK instance will make the NIC inaccessible for RDMA. Thus, Catmint×Cattree must carefully blocklist all NICs for DPDK.

Demikernel memory allocator provides DMA-capable memory for DPDK network or SPDK storage I/O. We modify Hoard to allocate memory objects from the DPDK memory pool for SPDK and register the same memory with RDMA. We split the fast path coroutine between polling DPDK devices and SPDK completion queues in a round-robin fashion, allocating a fair share of CPU cycles to both given no pending I/O. More complex scheduling of CPU cycles between network and storage I/O processing is possible in the future. In general, portable integration between networking and storage datapath libOSes significantly simplifies µs-scale applications running across network and storage kernel-bypass devices.

6 Demikernel Library OS Implementations

We prototype two Demikernel datapath OSes: DEMILIN for Linux and DEMIWIN for Windows. Table 2 lists the library Oses that make up each datapath OS. The legacy kernels have little impact on the design of the two datapath OSes; instead, they primarily accommodate differences in kernel-bypass frameworks on Windows and Linux (e.g., the Linux and Windows RDMA interfaces are very different). DEMILIN supports RDMA, DPDK and SPDK kernel-bypass devices. It compiles into 6 shared libraries: Catnap, Catmint, Catnip, Cattree, Catmint×Cattree and Catnip×Cattree. It uses DPDK 19.08 [16], SPDK 19.10 [88], and the `rdmacm` and `ibverbs` interface included with the Mellanox OFED driver [61] 5.0.2 and Ubuntu 18.04. DEMIWIN currently supports only RDMA kernel-bypass devices with Catpaw and the Catnap POSIX libOS through WSL. DPDK and SPDK are not well supported on Windows; however, mainline support for both is currently in development. Catpaw is built on NDSPI v2 [63]. This section describes their implementation.

6.1 Catnap POSIX Library OS

We developed Catnap to test and develop Demikernel applications without kernel-bypass hardware, which is an important feature for building µs-scale datacenter applications. Demikernel's flexible library OS architecture lets us support such a

Table 2. *Demikernel library operating systems.* We implement two prototype Demikernel datapath OSes: DEMILIN for Linux and DEMIWIN for Windows. Each datapath OS consists of a set of library OSes (e.g., DEMIWIN includes Catpaw and Catnap), which offer portability across different kernel-bypass devices.

LibOS Name	Datapath OS	Kernel-bypass	LoC
Catpaw	DEMIWIN	RDMA	6752 C++
Catnap	DEMILIN, DEMIWIN	N/A	822 C++
Catmint	DEMILIN	RDMA	1904 Rust
Catnip	DEMILIN	DPDK	9201 Rust
Cattree	DEMILIN	SPDK	2320 Rust

libOS without increasing the overhead or complexity of our other libOSes. Catnap follows the flow shown in Figure 4 but uses POSIX `read` and `write` in non-blocking mode instead of `epoll` to minimize latency. Catnap supports storage with files in a similar fashion. Catnap does not require memory management since POSIX is not zero-copy, and it has no background tasks since the Linux kernel handles all I/O.

6.2 Catmint and Catpaw RDMA Library OSes

Catmint builds PDPIX queues atop the `rdma_cm` [79] interfaces to manage connections and the `ib_verbs` interface to efficiently send and receive messages. It uses two-sided RDMA operations to send and receive messages, which simplifies support for the `wait_*` interface. We use a similar design for Catpaw atop NSDPI.

We found that using one RDMA queue pair per connection was unaffordable [35], so Catmint uses one queue pair per device and implements connection-based multiplexing for PDPIX queues. It processes I/O following the common flow, using `poll_cq` to poll for completions and `ibv_post_wr` to submit send requests to other nodes and post receive buffers. The only slow path operation buffers sends for flow control; Catmint allocates one coroutine per connection to re-send when the receiver updates the send window.

Catmint implements flow control using a simple message-based send window count and a one-sided `write` to update the sender's window count. It currently only supports messages up to a configurable buffer size. Further, it uses a *flow-control coroutine* per connection to allocate and post receive buffers and remotely update the send window. The fast-path coroutine checks the remaining receive buffers on each incoming I/O and unblocks the flow-control coroutine if the remaining buffers fall below a fixed number.

6.3 Catnip DPDK Library OS

Catnip implements UDP and TCP networking stacks on DPDK according to RFCs 793 and 7323 [82, 6] with the Cubic congestion control algorithm [26]. Existing user-level stacks did not meet our needs for ns-scale latencies: mTCP [30], Stackmap [102], f-stack [20] and SeaStar [42] all report double-digit microsecond latencies. In contrast, Catnip can process an incoming TCP packet and dispatch it to the waiting application coroutine in 53ns.

Unlike existing TCP implementations [71, 30], it is able to leverage coroutines for a linear programming flow through the state machine. Because coroutines efficiently encapsulate TCP connection state, they allow asynchronous programming without managing significant global state.

The Catnip TCP stack is deterministic. Every TCP operation is parameterized on a time value, and Catnip moves time forward by synchronizing with the system clock. As a result, Catnip is able control all inputs to the TCP stack, including packets and time, which let us easily debug the stack by feeding it a trace with packet timings.

Figure 4 shows Catnip's I/O loop assuming sufficient send window space, congestion window space, and that the physical address is in the ARP cache; otherwise, it spawns a send coroutine. Established sockets have four background coroutines to handle sending outgoing packets, retransmitting lost packets, sending pure acknowledgments, and manage connection close state transitions. During normal operation, all are blocked. However, if there is an adverse event (e.g. packet loss), the fast-path coroutine unblocks the needed background coroutine. Additional coroutines handle connection establishment: sockets in the middle of an active or passive open each have a background coroutine for driving the TCP handshake.

For full zero-copy, Catnip cannot use a buffer to account for TCP windows. Instead, it uses a ring buffer of I/O buffers and indices into the ring buffer as the start and end of the TCP window. This design increases complexity but eliminates an unnecessary copy from existing designs. Catnip limits its use of unsafe Rust to C/C++ bindings. As a result, it is the first zero-copy, ns-scale *memory-safe* TCP stack.

6.4 Cattree SPDK Library OS

Cattree maps the PDPIX queue abstraction onto an abstract log for SPDK devices. We map each device as a log file; applications open the file and `push` writes to the file for persistence. Cattree keeps a read cursor for every storage queue. `pop` reads (a specified number of bytes) from the read cursor and `push` appends to the log. `seek` and `truncate` move the read cursor and garbage collect the log. This log-based storage stack worked well for our echo server and Redis's persistent logging mechanism, but we hope to integrate more complex storage stacks.

Cattree uses its I/O processing fast path coroutine to poll for completed I/O operations and deliver them to the waiting application qtoken. Since the SPDK interface is asynchronous, Cattree submits disk I/O operations inline on the application coroutine and then yields until the request is completed. It has no background coroutines because all its work is directly related to active I/O processing. Cattree is a minimal storage stack with few storage features. While it works well for our logging-based applications, we expect that more complex storage systems might be layered above it in the future.

7 Evaluation

Our evaluation found that the prototype Demikernel datapath OSes simplified µs-scale kernel-bypass applications while imposing ns-scale overheads. All Demikernel and application code is available at: https://github.com/demikernel/demikernel.

7.1 Experimental Setup

We use 5 servers with 20-core dual-socket Xeon Silver 4114 2.2 GHz CPUs connected with Mellanox CX-5 100 Gbps NICs and an Arista 7060CX 100 Gbps switch with a minimum 450 ns switching latency. We use Intel Optane 800P NVMe SSDs, backed with 3D XPoint persistent memory. For Windows experiments, we use a separate cluster of 14-core dual-socket Xeon 2690 2.6 GHz CPU servers connected with Mellanox CX-4 56 Gbps NICs and a Mellanox SX6036 56 Gbps Infiniband switch with a minimum 200 ns latency.

On Linux, we allocate 2 GB of 2 MB huge pages, as required by DPDK. We pin processes to cores and use the performance CPU frequency scaling governor. To further reduce Linux latency, we raise the process priority using `nice` and use the real-time scheduler, as recommended by Li [53]. We run every experiment 5 times and report the average; the standard deviations are minimal – zero in some cases – except for § 7.6 where we report them in Figure 12.

Client and server machines use matching configurations since some Demikernel libOSes require both clients and servers run the same libOS; except the UDP relay application, which uses a Linux-based traffic generator. We replicated experiments with both Hoard and the built-in Linux libc allocator and found no apparent performance differences.

Comparison Systems. We compare Demikernel to 2 kernel-bypass applications – testpmd [93] and perftest [84] – and 3 recent kernel-bypass libraries – eRPC [34] Shenango [71] and Caladan [23]. testpmd and perftest are included with the DPDK and RDMA SDKs, respectively, and used as raw performance measurement tools. testpmd is an L2 packet forwarder, so it performs no packet processing, while perftest measures RDMA NIC `send` and `recv` latency by pinging a remote server. These applications represent the best "native" performance with the respective kernel-bypass devices.

Shenango [71] and Caladan [23] are recent kernel-bypass schedulers with a basic TCP stack; Shenango runs on DPDK, while Caladan directly uses the OFED API [2, 59]. eRPC is a low-latency kernel-bypass RPC library that supports RDMA, DPDK and OFED with a custom network transport. We allocate two cores to Shenango and Caladan for fairness: one each for the IOKernel and application.

7.2 Programmability for µs-scale Datacenter Systems

To evaluate Demikernel's impact on µs-scale kernel-bypass development, we implement four µs-scale kernel-bypass systems for Demikernel, including a UDP relay server built by a non-kernel-bypass, expert programmer. Table 3 presents a summary of the lines of code needed for POSIX and Demikernel versions. In general, we found Demikernel easier to use than POSIX-based OSes because its API is better suited to µs-scale datacenter systems and its OS services better met their needs, including portable I/O stacks, a DMA-capable heap and UAF protection.

Table 3. *LoC for* µs*-scale kernel-bypass systems.* POSIX and Demikernel versions of each application. The UDP relay also supports io_uring (1782 Loc), and TxnStore has a custom RDMA RPC library (12970 LoC).

OS/API	Echo Server	UDP Relay	Redis	TxnStore
POSIX	328	1731	52954	13430
Demikernel	291	2076	54332	12610

Echo Server and Client. To identify Demikernel's design trade-offs and performance characteristics, we build two echo systems with servers and clients using POSIX and Demikernel. This experiment demonstrates the benefits of Demikernel's API and OS management features for even simple µs-scale, kernel-bypass applications compared to current kernel-bypass libraries that preserve the POSIX API (e.g., Arrakis [73], mTCP [30], F-stack [20]).

Both echo systems run a single server-side request loop with closed-loop clients and support synchronous logging to disk. The Demikernel server calls `pop` on a set of I/O queue descriptors and uses `wait_any` to block until a message arrives. It then calls `push` with the message buffer on the same queue to send it back to the client and immediately frees the buffer. Optionally, the server can `push` the message to on-disk file for persistence before responding to the client.

To avoid heap allocations on the datapath, the POSIX echo server uses a pre-allocated buffer to hold incoming messages. Since the POSIX API is not zero-copy, both `read` and `write` require a copy, adding overhead. Even if the POSIX API were updated to support zero-copy (and it is unclear what the semantics would be in that case), correctly using it in the echo server implementation would be non-trivial. Since the POSIX server reuses a pre-allocated buffer, the server cannot re-use the buffer for new incoming messages until the previous message has been successfully sent and acknowledged. Thus, the server would need to implement a buffer pool with reference counting to ensure correct behavior. This experience has been corroborated by the Shenango [71] authors.

In contrast, Demikernel's clear zero-copy API dictates when the echo server receives ownership of the buffer, and its use-after-free semantics make it safe for the echo server to free the buffer immediately after the `push`. Demikernel's API semantics and memory management let Demikernel's echo server implementation process messages *without allocating or copying memory on the I/O processing path.*

TURN UDP Relay. Teams and Skype are large video conferencing services that operate peer-to-peer over UDP. To

support clients behind NATs, Microsoft Azure hosts millions of TURN relay servers [100, 29]. While end-to-end latency is not concern for these servers, the number of cycles spent on each relayed packet directly translate to the service's CPU consumption, which is significant. A Microsoft Teams engineer ported the relay server to Demikernel. While he has 10+ years of experience in the Skype and Teams groups, he was not a kernel-bypass expert. It took him *1 day* [38] to port the TURN server to Demikernel and 2 days each for `io_uring` [12] and Seastar [42]. In the end, he could not get Seastar working and had issues with `io_uring`. Compared to `io_uring` and Seastar, he reported that Demikernel was the simplest and easiest to use, and that PDPIX was his favorite part of the system. This experience demonstrated that, compared to existing kernel-bypass systems, *Demikernel makes kernel-bypass easier to use for programmers that are not kernel-bypass experts.*

Redis. We next evaluate the experience of porting the popular Redis [80] in-memory data structure server to Demikernel. This port required some architectural changes because Redis uses its own event processing loop and custom event handler mechanism. We implement our own Demikernel-based event loop, which replicated some functionality that already exists in Redis. This additional code increases the total modified LoC, but much of it was template code.

Redis's existing event loop processes incoming and outgoing packets in a single `epoll` event loop. We modify this loop to `pop` and `push` to Demikernel I/O queues and block using `wait_any`. Redis synchronously logs updates to disk, which we replace with a `push` to a log file without requiring extra buffers or copies.

The Demikernel implementation fixes several well-known inefficiencies in Redis due to `epoll`. For example, `wait_any` directly returns the packet, so Redis can immediately begin processing without further calls to the (lib)OS. Redis also processes outgoing replies in an asynchronous manner; it queues the response, waits on epoll for notification that the socket is ready and then writes the outgoing packet. This design is ineffcient: it requires more than one system call and several copies to send a reply. Demikernel fixes this inefficiency by letting the server immediately `push` the response into the outgoing I/O queue.

Demikernel's DMA-capable heap lets Redis directly place incoming PUTs and serve outgoing GETs to and from its in-memory store. For simple keys and values (e.g., not sets or arrays), Redis does not update in place, so Demikernel's use-after-free protection is sufficient for correct zero-copy I/O coordination. As a result, *Demikernel lets Redis correctly implement zero-copy I/O from its heap with no code changes.*

TxnStore. TxnStore [103] is a high-performance, in-memory, transactional key-value store that supports TCP, UDP, and RDMA. It illustrates Demikernel's benefits for a feature-rich, µs-scale system: TxnStore has double-digit µs latencies, implements transactions and replication, and uses the Protobuf library [76]. TxnStore also has its own implementation of RPC over RDMA, so this experiment let us compare Demikernel to custom RDMA code.

TxnStore uses interchangeable RPC transports. The standard one uses libevent for I/O processing, which relies on `epoll`. We implemented our own transport to replace libevent with a custom event loop based on `wait_any`. As a result of this architecture, the Demikernel port replicates significant code for managing connections and RPC, increasing the LoC but not the complexity of the port. TxnStore's RDMA transport does not support zero-copy I/O because it would require serious changes to ensure correctness. *Compared to the custom solution, Demikernel simplifies the coordination needed to support zero-copy I/O.*

7.3 Echo Application

To evaluate our prototype Demikernel OSes and their overheads, we measure our echo system. Client and server use matching OSes and kernel-bypass devices.

DEMILIN Latencies and Overheads. We begin with a study of Demikernel performance for small 64B messages. Figure 5 shows unloaded RTTs with a single closed-loop client. We compare the Demikernel echo system to the POSIX version, with eRPC, Shenango and Caladan. Catnap achieves better latency than the POSIX echo implementation because it polls `read` instead of using `epoll`; however, this is a trade-off, Catnap consumes 100% of one CPU even with a single client.

Catnip is 3.1µs faster than Shenango but 1.7µs slower than Caladan. Shenango has higher latency because packets traverse 2 cores for processing, while Caladan has run-to-completion on a single core. Caladan has lower latency because it uses the lower-level OFED API but sacrifices portability for non-Mellanox NICs. Similarly, eRPC has 0.2µs lower latency than Catmint but is carefully tuned for Mellanox CX5 NICs. This experiment demonstrates that, compared to other kernel-bypass systems, *Demikernel can achieve competitive* µs *latencies without sacrificing portability.*

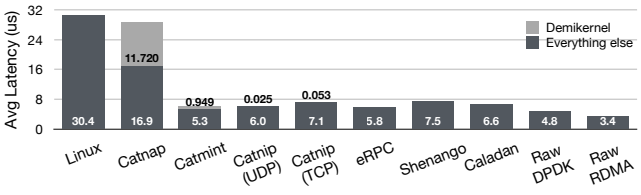
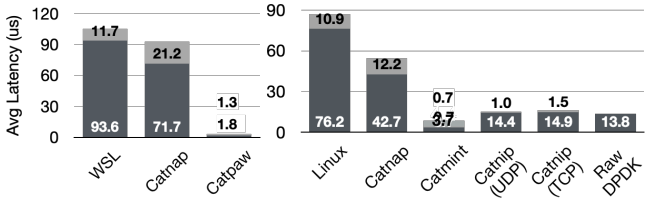


Figure 5. *Echo latencies on Linux (64*B*).* The upper number reports total time spent in Demikernel for 4 I/O operations: client and server send and receive; the lower ones show network and other latency; their sum is the total RTT on Demikernel. Demikernel achieves ns-scale overheads per I/O and has latencies close to those of eRPC, Shenango and Caladan, while supporting a greater range of devices and network protocols. We perform 1 million echos over 5 runs, the variance between runs was below 1%.



(a) DEMIWIN **(b)** DEMILIN in Azure VM

Figure 6. *Echo latencies on Windows and Azure (64*B*).* We demonstrate portability by running the Demikernel echo server on Windows and in a Linux VM with no code changes. We use the same testing methodology and found minimal deviations.

On average, Catmint imposes 250ns of latency overhead per I/O, while Catnip imposes 125ns per UDP packet and 200ns per TCP packet. Catmint trades off latency on the critical path for better throughput, while Catnip uses more background co-routines. In both cases, *Demikernel achieves* ns*-scale I/O processing overhead.*

DEMIWIN and Azure Latencies and Overhead. To further demonstrate portability, we evaluate the echo system on Windows and Azure. To our knowledge, Demikernel is the only kernel-bypass OS to portably support Windows and virtualized environments.

DEMIWIN supports Catnap through the WSL (the Windows Subsystem for Linux) POSIX interface and Catpaw. Figure 6a reports that Catnap again offers a small latency improvement by avoiding `epoll`, while Catpaw reduces latency by 27×. This improvement is extreme; however, WSL is not as optimized as the native Windows API.

Azure supports DPDK in general-purpose VMs and offers bare metal RDMA VMs with an Infiniband interconnect. Azure does not virtualize RDMA because networking support for congestion control over virtual networks is complex, demonstrating the pros and cons of heterogenous kernel-bypass devices in different environments.

Figure 6b shows DEMILIN in an Azure virtual machine. Catnap's polling design shows even lower latency in a VM because it spins the vCPU, so the hypervisor does not deschedule it. Catnip offers a 5× latency improvement over the Linux kernel. This is less than the bare metal performance improvement because DPDK still goes through the Azure virtualization layer in the SmartNIC [21] for vnet translation. Catmint runs bare metal on Azure over Infiniband; thus, it offers native performance. This experiment shows that *Demikernel lets* µs*-scale applications portably run on different platforms and kernel-bypass devices.*

DEMILIN Network and Storage Latencies. To show Demikernel's network and storage support, we run the same experiment with server-side logging. We use the Linux ext4 file system for the POSIX echo server and Catnap, and we integrate Catnip×Cattree and Catmint×Cattree. Figure 7 shows that writing every message synchronously to disk imposes a

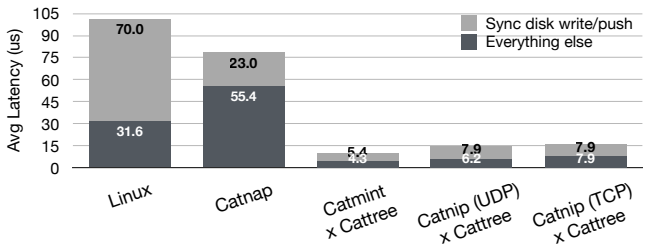


Figure 7. *Echo latencies on Linux with synchronous logging to disk (64*B*).* Demikernel offers lower latency to remote disk than kernel-based OSes to remote memory. We use the same testing methodology and found less than 1% deviation.

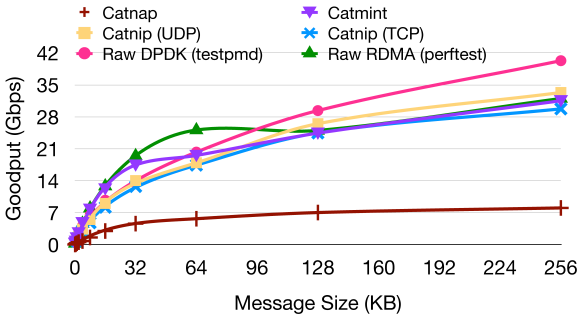


Figure 8. *NetPIPE Comparison.* We measure the bandwidth achievable with a single client and server sending and receiving messages of varying sizes, replicating NetPIPE [3].

high latency penalty on Linux and Catnap. Catnap's polling-based design lowers the overhead, but Catnip and Catmint are much faster. Due to its coroutine scheduler and memory management, Demikernel can process an incoming packet, pass it to the echo server, write it to disk and reply back in a tight run-to-completion loop without copies. As a result, a client sees lower access latencies to remote disk with Demikernel than remote memory on Linux. This experiment demonstrates that *Demikernel portably achieves* ns*-scale I/O processing, run-to-completion and zero-copy for networking and storage.*

DEMILIN Single Client Throughput Overheads. To study Demikernel's performance for varying message sizes, we use NetPIPE [68] to compare DEMILIN with DPDK testpmd and RDMA perftest. Figure 8 shows that testpmd offers 40.3Gbps for 256kB messages, while perftest achieves 37.7Gbps. testpmd has better performance because it is strictly an L2 forwarder, while perftest uses the RoCE protocol, which requires header processing and acks to achieve ordered, reliable delivery.

Catmint achieves 31.5Gbps with 256kB messages, which is a 17% overhead on perftest's raw RDMA performance. Catnip has 33.3Gbps for UDP and 29.7Gbps for TCP, imposing a 17% and 26% performance penalty, respectively, on testpmd for 256kB messages. Again, note that testpmd performs no packet processing, while Catnip has a software networking stack. Compared to native kernel-bypass applications, we found that *Demikernel provides OS management services with reasonable cost for a range of packet sizes.*

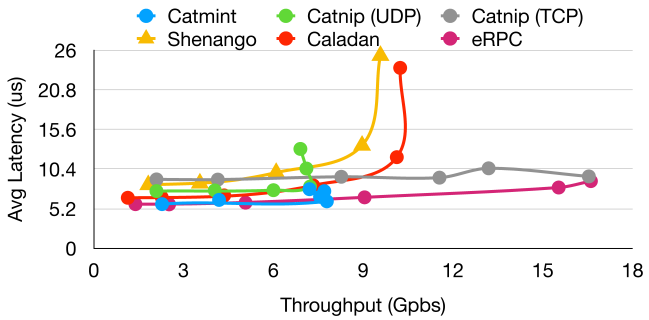


Figure 9. *Latency vs. throughput.* We skip RDMA (because perftest does not support parallelism) and the kernel-based solutions for readability. We use 1 core for Demikernel and eRPC, and 2 cores for Shenango and Caladan. Catmint and Catnip (UDP) are optimized for latency but not throughput, as we focused our efforts on improving Catnip's software TCP stack.

.

DemiLin Peak Throughput and Latency Overheads. To evaluate Demikernel's throughput impact, we compare to Shenango, Caladan and eRPC. We increase offered load and report both sustained server-side throughput and client-side latency in Figure 9. Catnip (UDP) achieves 70% of the peak throughput of Caladan on DPDK and Catmint has 46% the throughput of eRPC on RDMA. However, Catnip outperforms Caladan and is competitive with eRPC due to many optimizations in the TCP stack. We focused our initial optimization efforts on per-I/O datapath latency overheads since µs-scale datacenter systems prioritize datapath latencies and tail latencies; however, future optimizations could improve throughput, especially single-core throughput (e.g., offloading background coroutines to another core). Overall, this experiment shows that *Demikernel demonstrates that a high-level API and OS features do not have to come at a significant cost.*

7.4 UDP Relay Server

To evaluate the performance of a µs-scale system built by a non-kernel-bypass expert, we compare the performance of the UDP relay server on Demikernel to Linux and io_uring. We use a non0-kernel-bypass Linux-based traffic generator and measure the latency between sending the generated packet and receiving the relayed packet.

Figure 10 shows the average and p99 latencies for the UDP relay server. Since all versions use the same Linux-based client, the reduced latency directly translates to fewer CPU cycles used on the server. While `io_uring` gives modest improvements, Catnip reduces the server-side latency per request by 11µs on average and 13.7µs at the p99, demonstrating that *Demikernel lets non-kernel-bypass experts reap performance benefits for their* µs*-scale applications.*

7.5 Redis In-memory Distributed Cache

To evaluate a ported µs-scale application, we measure Redis using `GET` and `SET` operations with the built-in `redis-benchmark` utility. Figure 11 (left) reports the peak throughput for unmodified Redis and Demikernel Redis running in-memory. Catnap has 75-80% lower peak throughput because its polling-based design trades off latency for throughput. Catmint gives 2× throughput, and Catnip offers 20% better throughput.

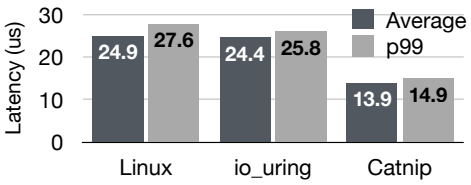


Figure 10. *Average and tail latencies for UDP relay.* We send 1 million packets and perform the experiment 5 times. Demikernel has better performance than io_uring and requires fewer changes.

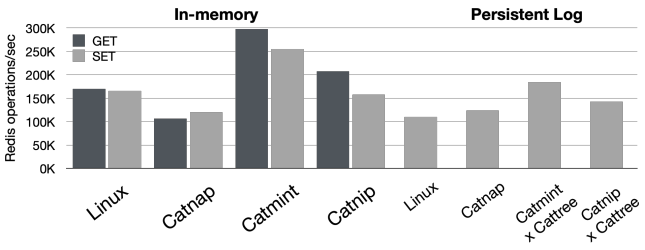


Figure 11. *Redis benchmark throughput in-memory and on-disk.* We use 64B values and 1 million keys. We perform separate runs for each operation with 500,000 accesses repeated 5 times. Demikernel improves Redis performance and lets it maintain that performance with synchronous writes to disk.

We add persistence to Redis by turning on its append-only file. We `fsync` after each `SET` operation for strong guarantees and fairness to Cattree, which does not buffer or cache in memory. Figure 11 (left) gives the results. In this case, Catnap's polling increases throughput because it decreases disk access latencies, which block the server from processing more requests. Notably, Catnip and Catmint with Cattree have throughput within 10% of unmodified Redis *without persistence*. This result shows that *Demikernel provides existing* µs*-scale applications portable kernel-bypass network and storage access with low overhead.*

7.6 TxnStore Distributed Transactional Storage

To measure the performance of Demikernel for a more fully featured µs-scale application, we evaluate TxnStore with YCSB-t workload F, which performs read-modify-write operations using transactions. We use a uniform Zipf distribution, 64 B keys, and 700 B values. We use the weakly consistent quorum-write protocol: every `get` accesses a single server, while every `put` replicates to three servers.

Figure 12 reports the average and 99th percentile tail transaction latency. Again, Catnap has 69% lower latency than TxnStore's TCP stack and 27% lower latency than its UDP stack due to polling. As a trade-off, Catnap uses almost 100% of a single CPU to poll for packets, while TxnStore uses 40%.

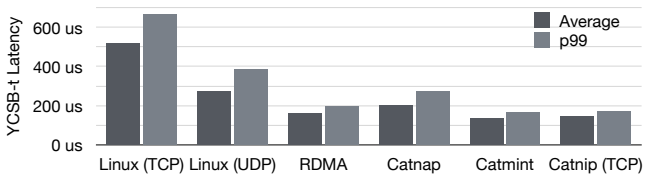


Figure 12. *YCSB-t average and tail latencies for TxnStore.* Demikernel offers lower latency than TxnStore's custom RDMA stack because we are not able to remove copies in the custom RDMA stack without complex zero-copy memory coordination.

Both Catnip and Catmint are competitive with TxnStore's RDMA messaging stack. TxnStore uses the `rdma_cm` [79]. However, it uses one queue pair per connection, requires a copy, and has other inefficiencies [36], so Catmint outperforms TxnStore's native RDMA stack. This experiment shows that *Demikernel improves performance for higher-latency* µs*-scale datacenter applications compared to a naive custom RDMA implementation.*

8 Related Work

Demikernel builds on past work in operating systems, especially library OSes [18, 74, 49] and other flexible, extensible systems [31, 89, 87], along with recent work on kernel-bypass OSes [73, 3] and libraries [30, 44]. Library operating systems separate protection and management into the OS kernel and user-level library OSes, respectively, to better meet custom application needs. We observe that kernel-bypass architectures offload protection into I/O devices, along with some OS management, so Demikernel uses library OSes for portability across heterogenous kernel-bypass devices.

OS extensions [99, 19, 22, 24] also let applications customize parts of the OS for their needs. Recently, Linux introduced `io_uring` [12, 11], which gives applications faster access to the kernel I/O stack through shared memory. LKL [77, 94] and F-stack [20] move the Linux and FreeBSD networking stacks to userspace but do not meet the requirements of µs-scale systems (e.g., accommodating heterogenous device hardware and zero-copy I/O memory management). Device drivers, whether in the OS kernel [91, 13] or at user level [48], hide differences between hardware interfaces but do not implement OS services, like memory management.

Previous efforts to offload OS functionality focused on limited OS features or specialized applications, like TCP [14, 65, 67, 39]. DPI [1] proposes an interface similar to the Demikernel libcall interface but uses flows instead of queues and considers network I/O but not storage. Much recent work on distributed storage systems uses RDMA for low-latency access to remote memory [17], FASST [37], and more [98, 64, 35, 92, 7, 40] but does not portably support other NIC hardware. Likewise, software middleboxes have used DPDK for low-level access to the NIC [95, 90] but do not consider other types of kernel-bypass NICs or storage.

Arrakis [73] and Ix [3] offer alternative kernel-bypass architectures but are not portable, especially to virtualized environments, since they leverage SR-IOV for kernel-bypass. Netmap [83] and Stackmap [102] offer user-level interfaces to NICs but no OS management. eRPC [34] and ScaleRPC [8] are user-level RDMA/DPDK stacks, while ReFlex [43], PASTE [28] and Flashnet [96] provide fast remote access to storage, but none portably supports both storage and networking.

User-level networking [51, 39, 70, 30, 42] and storage stacks [44, 81, 32] replace missing functionality and can be used interchangeably if they maintain the POSIX API; however, they lack features needed by µs-scale kernel-bypass systems, as described in Section 2. Likewise, recent user-level schedulers [75, 71, 33, 23, 9] assign I/O requests to application workers but are not portable and do not implement storage stacks. As a result, none serve as general-purpose datapath operating systems.

9 Conclusion And Future Work

Demikernel is a first step towards datapath OSes for µs-scale kernel-bypass applications. While we present an OS and architecture that implement PDPIX, other designs are possible. Each Demikernel OS feature represents a rich area of future work. We have barely scratched the surface of portable, zero-copy TCP stacks and have not explored in depth what semantics a µs-scale storage stack might supply. While there has been recent work on kernel-bypass scheduling, efficient µs-scale memory resource management with memory allocators has not been explored in depth. Given the insights of the datapath OS into the memory access patterns of the application, improved I/O-aware memory scheduling is certainly possible. Likewise, Demikernel does not eliminate all zero-copy coordination and datapath OSes with more explicit features for memory ownership are a promising direction for more research. Generally, we hope that Demikernel is the first of many datapath OSes for µs-scale datacenter applications.

10 Acknowledgements

It took a village to make Demikernel possible. We thank Emery Berger for help with Hoard integration, Aidan Woolley and Andrew Moore for Catnip's Cubic implementation, and Liam Arzola and Kevin Zhao for code contributions. We thank Adam Belay, Phil Levis, Josh Fried, Deepti Raghavan, Tom Anderson, Anuj Kalia, Landon Cox, Mothy Roscoe, Antoine Kaufmann, Natacha Crooks, Adriana Szekeres, and the entire MSR Systems Group, especially Dan Ports, Andrew Baumann, and Jay Lorch, who read many drafts. We thank Sandy Kaplan for repeatedly editing the paper. Finally, we acknowledge the dedication of the NSDI, OSDI and SOSP reviewers, many of whom reviewed the paper more than once, and the tireless efforts of our shepherd Jeff Mogul.

References

[1] ALONSO, G., BINNIG, C., PANDIS, I., SALEM, K., SKRZYPCZAK, J., STUTSMAN, R., THOSTRUP, L., WANG, T., WANG, Z., AND ZIEGLER, T. DPI: The data processing interface for modern networks. In *9th Biennial Conference on Innovative Data Systems Research CIDR* (2019).

[2] BARAK, D. The OFED package, April 2012. https://www.rdmamojo.com/2012/04/25/the-ofed-package/.

[3] BELAY, A., PREKAS, G., KLIMOVIC, A., GROSSMAN, S., KOZYRAKIS, C., AND BUGNION, E. IX: A protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (2014), USENIX Association.

[4] BERGER, E. D., MCKINLEY, K. S., BLUMOFE, R. D., AND WILSON, P. R. Hoard: A scalable memory allocator for multithreaded applications. *SIGARCH Comput. Archit. News 28*, 5 (Nov. 2000), 117–128.

[5] BERGER, E. D., ZORN, B. G., AND MCKINLEY, K. S. Composing high-performance memory allocators. *SIGPLAN Not. 36*, 5 (May 2001), 114–124.

[6] BORMAN, D., BRADEN, R. T., JACOBSON, V., AND SCHEFFENEGGER, R. TCP Extensions for High Performance. RFC 7323, 2014.

[7] CHEN, H., CHEN, R., WEI, X., SHI, J., CHEN, Y., WANG, Z., ZANG, B., AND GUAN, H. Fast in-memory transaction processing using RDMA and HTM. *ACM Trans. Comput. Syst. 35*, 1 (July 2017).

[8] CHEN, Y., LU, Y., AND SHU, J. Scalable RDMA RPC on reliable connection with efficient resource sharing. In *Proceedings of the Fourteenth EuroSys Conference 2019* (2019), Association for Computing Machinery.

[9] CHO, I., SAEED, A., FRIED, J., PARK, S. J., ALIZADEH, M., AND BELAY, A. Overload control for µs-scale RPCs with Breakwater. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)* (2020), USENIX Association.

[10] CORBET. Linux and TCP offload engines. LWN Articles, August 2005. https://lwn.net/Articles/148697/.

[11] CORBET, J. Ringing in a new asynchronous I/O API. lwn.net, Jan 2019. https://lwn.net/Articles/776703/.

[12] CORBET, J. The rapid growth of io_uring. lwn.net, Jan 2020. https://lwn.net/Articles/810414.

[13] CORBET, J., RUBINI, A., AND KROAH-HARTMAN, G. *Linux Device Drivers: Where the Kernel Meets the Hardware*. " O'Reilly Media, Inc.", 2005.

[14] CURRID, A. TCP offload to the rescue: Getting a toehold on TCP offload engines—and why we need them. *Queue 2*, 3 (May 2004), 58–65.

[15] DEMOULIN, M., FRIED, J., PEDISICH, I., KOGIAS, M., LOO, B. T., PHAN, L. T. X., AND ZHANG, I. When idling is ideal: Optimizing tail-latency for highly-dispersed datacenter workloads with Persephone. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2021), Association for Computing Machinery.

[16] Data plane development kit. https://www.dpdk.org/.

[17] DRAGOJEVIĆ, A., NARAYANAN, D., CASTRO, M., AND HODSON, O. FaRM: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (2014), USENIX Association.

[18] ENGLER, D. R., KAASHOEK, M. F., AND O'TOOLE, J. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (1995), Association for Computing Machinery.

[19] EVANS, J. A scalable concurrent malloc (3) implementation for FreeBSD. In *Proceedings of the BSDCan Conference* (2006).

[20] F-Stack. http://www.f-stack.org/.

[21] FIRESTONE, D., PUTNAM, A., MUNDKUR, S., CHIOU, D., DABAGH, A., ANDREWARTHA, M., ANGEPAT, H., BHANU, V., CAULFIELD, A., CHUNG, E., CHANDRAPPA, H. K., CHATURMOHTA, S., HUMPHREY, M., LAVIER, J., LAM, N., LIU, F., OVTCHAROV, K., PADHYE, J., POPURI, G., RAINDEL, S., SAPRE, T., SHAW, M., SILVA, G., SIVAKUMAR, M., SRIVASTAVA, N., VERMA, A., ZUHAIR, Q., BANSAL, D., BURGER, D., VAID, K., MALTZ, D. A., AND GREENBERG, A. Azure accelerated networking: SmartNICs in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)* (2018), USENIX Association.

[22] FLEMING, M. A thorough introduction to eBPF. lwn.net, December 2017. https://lwn.net/Articles/740157/.

[23] FRIED, J., RUAN, Z., OUSTERHOUT, A., AND BELAY, A. Caladan: Mitigating interference at microsecond timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)* (2020), USENIX Association.

[24] File system in user-space. https://www.kernel.org/doc/html/latest/filesystems/fuse.html.

[25] *A Tour of Go: Channels*. https://tour.golang.org/concurrency/2.

[26] HA, S., RHEE, I., AND XU, L. Cubic: a new tcp-friendly high-speed tcp variant. *ACM SIGOPS Operating Systems Review 42*, 5 (2008).

[27] HERBERT, T., AND DE BRUIJN, W. *Scaling in the Linux Networking Stack*. kernel.org. https://www.kernel.org/doc/Documentation/networking/scaling.txt.

[28] HONDA, M., LETTIERI, G., EGGERT, L., AND SANTRY, D. PASTE: A network programming interface for non-volatile main memory. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)* (2018), USENIX Association.

[29] INTERNET ENGINEERING TASK FORCE. *Traversal Using Relays around NAT*. https://tools.ietf.org/html/rfc5766.

[30] JEONG, E., WOOD, S., JAMSHED, M., JEONG, H., IHM, S., HAN, D., AND PARK, K. mTCP: a highly scalable user-level TCP stack for multicore systems. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (2014), USENIX Association.

[31] JIN, Y., TSENG, H.-W., PAPAKONSTANTINOU, Y., AND SWANSON, S. KAML: A flexible, high-performance key-value SSD. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)* (2017), IEEE.

[32] KADEKODI, R., LEE, S. K., KASHYAP, S., KIM, T., KOLLI, A., AND CHIDAMBARAM, V. SplitFS: Reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (2019), Association for Computing Machinery.

[33] KAFFES, K., CHONG, T., HUMPHRIES, J. T., BELAY, A., MAZIÈRES, D., AND KOZYRAKIS, C. Shinjuku: Preemptive scheduling for µsecond-scale tail latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)* (2019), USENIX Association.

[34] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. Datacenter RPCs can be general and fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)* (2019), USENIX Association.

[35] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Using RDMA efficiently for key-value services. *SIGCOMM Comput. Commun. Rev. 44*, 4 (Aug. 2014), 295–306.

[36] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Design guidelines for high performance RDMA systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)* (2016), USENIX Association.

[37] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. FaSST: Fast, scalable and simple distributed transactions with two-sided RDMA datagram RPCs. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (2016), USENIX Association.

[38] KALLAS, S. Turn server. Github. https://github.com/seemk/urn.

[39] KAUFMANN, A., STAMLER, T., PETER, S., SHARMA, N. K., KRISHNAMURTHY, A., AND ANDERSON, T. TAS: TCP acceleration as an OS service. In *Proceedings of the Fourteenth EuroSys Conference 2019* (2019), Association for Computing Machinery.

[40] KIM, D., MEMARIPOUR, A., BADAM, A., ZHU, Y., LIU, H. H., PADHYE, J., RAINDEL, S., SWANSON, S., SEKAR, V., AND SESHAN, S. Hyperloop: Group-based NIC-Offloading to accelerate replicated transactions in multi-tenant storage systems. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (2018), Association for Computing Machinery.

[41] KIM, H.-J., LEE, Y.-S., AND KIM, J.-S. NVMeDirect: A user-space I/O framework for application-specific optimization on NVMe SSDs. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)* (2016), USENIX Association.

[42] KIVITY, A. Building efficient I/O intensive applications with Seastar, 2019. https://github.com/CoreCppIL/CoreCpp2019/blob/master/Presentations/Avi_Building_efficient_IO_intensive_applications_with_Seastar.pdf.

[43] KLIMOVIC, A., LITZ, H., AND KOZYRAKIS, C. ReFlex: Remote flash = local flash. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (2017), Association for Computing Machinery.

[44] KWON, Y., FINGLER, H., HUNT, T., PETER, S., WITCHEL, E., AND ANDERSON, T. Strata: A cross media file system. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), Association for Computing Machinery.

[45] LAGUNA, I., MARSHALL, R., MOHROR, K., RUEFENACHT, M., SKJELLUM, A., AND SULTANA, N. A large-scale study of MPI usage in open-source HPC applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2019), Association for Computing Machinery.

[46] LEIJEN, D., ZORN, B., AND DE MOURA, L. Mimalloc: Free list sharding in action. In *Asian Symposium on Programming Languages and Systems* (2019).

[47] LEMIRE, D. Iterating over set bits quickly, Feb 2018. https://lemire.me/blog/2018/02/21/iterating-over-set-bits-quickly/.

[48] LESLIE, B., CHUBB, P., FITZROY-DALE, N., GÖTZ, S., GRAY, C., MACPHERSON, L., POTTS, D., SHEN, Y.-T., ELPHINSTONE, K., AND HEISER, G. User-level device drivers: Achieved performance. *Journal of Computer Science and Technology 20*, 5 (2005), 654–664.

[49] LESLIE, I., MCAULEY, D., BLACK, R., ROSCOE, T., BARHAM, P., EVERS, D., FAIRBAIRNS, R., AND HYDEN, E. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas in Communications 14*, 7 (1996), 1280–1297.

[50] LESOKHIN, I. tls: Add generic NIC offload infrastructure. LWN Articles, September 2017. https://lwn.net/Articles/734030.

[51] LI, B., CUI, T., WANG, Z., BAI, W., AND ZHANG, L. Socksdirect: Datacenter sockets can be fast and compatible. In *Proceedings of the ACM Special Interest Group on Data Communication* (2019), Association for Computing Machinery.

[52] LI, B., RUAN, Z., XIAO, W., LU, Y., XIONG, Y., PUTNAM, A., CHEN, E., AND ZHANG, L. KV-Direct: High-performance in-memory key-value store with programmable NIC. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), Association for Computing Machinery.

[53] LI, J., SHARMA, N. K., PORTS, D. R. K., AND GRIBBLE, S. D. Tales of the tail: Hardware, OS, and application-level sources of tail latency. In *Proceedings of the ACM Symposium on Cloud Computing* (2014), Association for Computing Machinery.

[54] libevent: an event notification library. http://libevent.org/.

[55] MANDRY, T. How Rust optimizes async/await I, Aug 2019. https://tmandry.gitlab.io/blog/posts/optimizing-await-1/.

[56] MAREK. Epoll is fundamentally broken 1/2, Feb 2017. https://idea.popcount.org/2017-02-20-epoll-is-fundamentally-broken-12/.

[57] MARTY, M., DE KRUIJF, M., ADRIAENS, J., ALFELD, C., BAUER, S., CONTAVALLI, C., DALTON, M., DUKKIPATI, N., EVANS, W. C., GRIBBLE, S., KIDD, N., KONONOV, R., KUMAR, G., MAUER, C., MUSICK, E., OLSON, L., RUBOW, E., RYAN, M., SPRINGBORN, K., TURNER, P., VALANCIUS, V., WANG, X., AND VAHDAT, A. Snap: A microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (2019), Association for Computing Machinery.

[58] MELLANOX. BlueField Smart NIC. http://www.mellanox.com/page/products_dyn?product_family=275&mtag=bluefield_smart_nic1.

[59] MELLANOX. *Mellanox OFED for Linux User Manual*. https://www.mellanox.com/related-docs/prod_software/Mellanox_OFED_Linux_User_Manual_v4.1.pdf.

[60] MELLANOX. An introduction to smart NICs. The Next Platform, 3 2019. https://www.nextplatform.com/2019/03/04/an-introduction-to-smartnics/.

[61] MELLANOX. Mellanox OFED RDMA libraries, April 2019. http://www.mellanox.com/page/mlnx_ofed_public_repository.

[62] MELLANOX. *RDMA Aware Networks Programming User Manual*, September 2020. https://community.mellanox.com/s/article/rdma-aware-networks-programming--160--user-manual.

[63] MICROSOFT. *Network Direct SPI Reference*, v2 ed., July 2010. https://docs.microsoft.com/en-us/previous-versions/windows/desktop/cc904391(v%3Dvs.85).

[64] MITCHELL, C., GENG, Y., AND LI, J. Using one-sided RDMA reads to build a fast, CPU-Efficient key-value store. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)* (2013), USENIX Association.

[65] MOGUL, J. C. TCP offload is a dumb idea whose time has come. In *9th Workshop on Hot Topics in Operating Systems (HotOS IX)* (2003), USENIX Association.

[66] MOON, Y., LEE, S., JAMSHED, M. A., AND PARK, K. AccelTCP: Accelerating network applications with stateful TCP offloading. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)* (2020), USENIX Association.

[67] NARAYAN, A., CANGIALOSI, F., GOYAL, P., NARAYANA, S., ALIZADEH, M., AND BALAKRISHNAN, H. The case for moving congestion control out of the datapath. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks* (2017), Association for Computing Machinery.

[68] *Network Protocol Independent Performance Evaluator*. https://linux.die.net/man/1/netpipe.

[69] NETRONOME. Agilio CX SmartNICs. https://www.netronome.com/products/agilio-cx/.

[70] OPENFABRICS INTERFACES WORKING GROUP. RSockets. GitHub. https://github.com/ofiwg/librdmacm/blob/master/docs/rsocket.

[71] OUSTERHOUT, A., FRIED, J., BEHRENS, J., BELAY, A., AND BALAKRISHNAN, H. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)* (2019), USENIX Association.

[72] PANDA, A., HAN, S., JANG, K., WALLS, M., RATNASAMY, S., AND SHENKER, S. Netbricks: Taking the V out of NFV. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (2016), USENIX Association.

[73] PETER, S., LI, J., ZHANG, I., PORTS, D. R. K., WOOS, D., KRISHNAMURTHY, A., ANDERSON, T., AND ROSCOE, T. Arrakis: The operating system is the control plane. *ACM Trans. Comput. Syst. 33*, 4 (Nov. 2015).

[74] PORTER, D. E., BOYD-WICKIZER, S., HOWELL, J., OLINSKY, R., AND HUNT, G. C. Rethinking the library OS from the top down. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (2011), Association for Computing Machinery.

[75] PREKAS, G., KOGIAS, M., AND BUGNION, E. ZygOS: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), Association for Computing Machinery.

[76] Protocol buffers. https://developers.google.com/protocol-buffers/.

[77] PURDILA, O., GRIJINCU, L. A., AND TAPUS, N. LKL: The linux kernel library. In *9th RoEduNet IEEE International Conference* (2010), IEEE.

[78] RDMA CONSORTIUM. A RDMA protocol specification, October 2002. http://rdmaconsortium.org/.

[79] *RDMA communication manager*. https://linux.die.net/man/7/rdma_cm.

[80] Redis: Open source data structure server, 2013. http://redis.io/.

[81] REN, Y., MIN, C., AND KANNAN, S. CrossFS: A cross-layered direct-access file system. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)* (2020), USENIX Association.

[82] Transmission Control Protocol. RFC 793, 1981. https://tools.ietf.org/html/rfc793.

[83] RIZZO, L. Netmap: A novel framework for fast packet I/O. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)* (2012), USENIX Association.

[84] *RDMA CM connection and RDMA ping-pong test*. http://manpages.ubuntu.com/manpages/bionic/man1/rping.1.html.

[85] RUST. *The Async Book*. https://rust-lang.github.io/async-book/.

[86] SESHADRI, S., GAHAGAN, M., BHASKARAN, S., BUNKER, T., DE, A., JIN, Y., LIU, Y., AND SWANSON, S. Willow: A user-programmable SSD. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (2014), USENIX Association.

[87] SIEGEL, A., BIRMAN, K., AND MARZULLO, K. Deceit: A flexible distributed file system. In *[1990] Proceedings. Workshop on the Management of Replicated Data* (1990), pp. 15–17.

[88] Storage performance development kit. https://spdk.io/.

[89] STRIBLING, J., SOVRAN, Y., ZHANG, I., PRETZER, X., KAASHOEK, M. F., AND MORRIS, R. Flexible, wide-area storage for distributed systems with WheelFS. In *6th USENIX Symposium on Networked Systems Design and Implementation (NSDI 09)* (2009), USENIX Association.

[90] SUN, C., BI, J., ZHENG, Z., YU, H., AND HU, H. NFP: Enabling network function parallelism in NFV. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (2017), Association for Computing Machinery.

[91] SWIFT, M. M., MARTIN, S., LEVY, H. M., AND EGGERS, S. J. Nooks: An architecture for reliable device drivers. In *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop* (2002), Association for Computing Machinery.

[92] TARANOV, K., ALONSO, G., AND HOEFLER, T. Fast and strongly-consistent per-item resilience in key-value stores. In *Proceedings of the Thirteenth EuroSys Conference* (2018), Association for Computing Machinery.

[93] *Testpmd Users Guide*. https://doc.dpdk.org/guides/testpmd_app_ug/.

[94] THALHEIM, J., UNNIBHAVI, H., PRIEBE, C., BHATOTIA, P., AND PIETZUCH, P. rkt-io: A direct I/O stack for shielded execution. In *Proceedings of the Sixteenth European Conference on Computer Systems* (2021), Association for Computing Machinery.

[95] TOOTOONCHIAN, A., PANDA, A., LAN, C., WALLS, M., ARGYRAKI, K., RATNASAMY, S., AND SHENKER, S. ResQ: Enabling SLOs in network function virtualization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)* (2018), USENIX Association.

[96] TRIVEDI, A., IOANNOU, N., METZLER, B., STUEDI, P., PFEFFERLE, J., KOURTIS, K., KOLTSIDAS, I., AND GROSS, T. R. FlashNet: Flash/network stack co-design. *ACM Trans. Storage 14*, 4 (Dec. 2018).

[97] TURON, A. Designing futures for Rust, Sep 2016. https://aturon.github.io/blog/2016/09/07/futures-design/.

[98] WEI, X., DONG, Z., CHEN, R., AND CHEN, H. Deconstructing RDMA-enabled distributed transactions: Hybrid is better! In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)* (2018), USENIX Association.

[99] WELCH, B. B., AND OUSTERHOUT, J. K. Pseudo devices: User-level extensions to the Sprite file system. Tech. Rep. UCB/CSD-88-424, EECS Department, University of California, Berkeley, Jun 1988.

[100] WIKIPEDIA. *Traversal Using Relays around NAT*, May 2021. https://en.wikipedia.org/wiki/Traversal_Using_Relays_around_NAT.

[101] YANG, J., IZRAELEVITZ, J., AND SWANSON, S. FileMR: Rethinking RDMA networking for scalable persistent memory. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)* (2020), USENIX Association.

[102] YASUKATA, K., HONDA, M., SANTRY, D., AND EGGERT, L. Stackmap: Low-latency networking with the OS stack and dedicated NICs. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)* (2016), USENIX Association.

[103] ZHANG, I., SHARMA, N. K., SZEKERES, A., KRISHNAMURTHY, A., AND PORTS, D. R. Building consistent transactions with inconsistent replication. *ACM Transactions on Computer Systems 35*, 4 (2018), 12.