Wave: Offloading Resource Management to SmartNIC Cores

Jack Tigar Humphries*
jhumphri@cs.stanford.edu
Stellar Development Foundation
San Francisco, CA, USA

Neel Natu
neelnatu@google.com
Google, Inc.
Sunnyvale, CA, USA

Kostis Kaffes
kkaffes@cs.columbia.edu
Columbia University
New York, NY, USA

Stanko Novaković
stanko@google.com
Google, Inc.
Seattle, WA, USA

Paul Turner
pjt@google.com
Google, Inc.
Sunnyvale, CA, USA

Henry M. Levy
hanklevy@google.com
Google, Inc.
Seattle, WA, USA
University of Washington
Seattle, WA, USA

David Culler
dculler@google.com
Google, Inc.
Sunnyvale, CA, USA
University of California, Berkeley
Berkeley, CA, USA

Christos Kozyrakis
kozyraki@stanford.edu
Stanford University
Stanford, CA, USA

Abstract

SmartNICs are increasingly deployed in datacenters to offload tasks from server CPUs, improving the efficiency and flexibility of datacenter security, networking and storage. Optimizing cloud server efficiency in this way is critically important to ensure that virtually all server resources are available to paying customers. Userspace system software, specifically, decision-making tasks performed by various operating system subsystems, is particularly well suited for execution on mid-tier SmartNIC ARM cores. To this end, we introduce Wave, a framework for offloading userspace system software to processes/agents running on the SmartNIC. Wave uses Linux userspace systems to better align system functionality with SmartNIC capabilities. It also introduces a new host-SmartNIC communication API that enables offloading of even μs-scale system software. To evaluate Wave, we offloaded preexisting userspace system software including kernel thread scheduling, memory management, and an RPC stack to SmartNIC ARM cores, which showed a performance degradation of 1.1%-7.4% in an apples-to-apples comparison with on-host implementations. Wave recovered host resources consumed by on-host system software for memory management (saving 16 host cores), RPCs (saving 8 host cores), and virtual machines (an 11.2% performance improvement). Wave highlights the potential for rethinking system software placement in modern datacenters, unlocking new opportunities for efficiency and scalability.

CCS Concepts: • **Computer systems organization**;

Keywords: Operating systems, SmartNICs

ACM Reference Format:
Jack Tigar Humphries, Neel Natu, Kostis Kaffes, Stanko Novaković, Paul Turner, Henry M. Levy, David Culler, and Christos Kozyrakis. 2025. Wave: Offloading Resource Management to SmartNIC Cores. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3676642.3736113

1 Introduction

Cloud providers continuously optimize server efficiency to ensure that virtually all server resources are available to paying customers. They do this by offloading virtualization and network stacks to accelerators on SmartNICs and other specialized offload devices, such as the AWS Nitro [1], NVIDIA BlueField [2], Fungible DPU [3], AMD Pensando DPU [4], and Intel Mount Evans [5]. For example, AWS offloads virtualized I/O and cloud security to their Nitro device to improve host utilization and reduce jitter. Recent work [6–12] offloads the host operating system network data plane, freeing host compute resources and enabling acceleration.

*Work completed while at Google, Inc. and Stanford University.

ASPLOS '25, Rotterdam, Netherlands

ACM ISBN 979-8-4007-1080-3/2025/03
https://doi.org/10.1145/3676642.3736113

Although some of these benefits could be realized with additional on-host compute, cloud providers deploy SmartNICs in many new datacenter machines because they offer the simplest and cheapest way to provide a uniform security and management view across machines regardless of vendor, architecture type, and maturity/availability of software [13]. Cloud providers also have tighter control over SmartNIC design than host architecture. They purchase large volumes of network hardware, so they work closely with hardware vendors to expedite decision-making [14] about accelerator logic, memory size, core count, and more. In contrast, host CPU architectures are often designed by external vendors for a wide range of cases and customers, so cloud providers cannot tailor these architectures to their specific needs.

Beyond acceleration, cloud providers also deploy mid-tier ARM cores inside SmartNICs, such as 16 ARM cores in Intel Mount Evans [5]. Although networking and storage tasks for virtual machines are being successfully offloaded to SmartNIC accelerators and datapaths, the systems community is still exploring ways to fully utilize these mid-tier ARM cores [10, 15–18]. Offloaded applications suffer from the low compute performance of ARM cores and high host-SmartNIC communication latency, occasionally experiencing worse end-to-end performance even with the additional SmartNIC resources [18]. Additionally, the complexity [19–21] involved in offloading applications across PCIe likely makes it impractical on a datacenter-wide basis, and, crucially, security concerns prevent many potential uses: providers typically restrict these ARM cores from running arbitrary code [22, 23].

Userspace system software is particularly well-suited for execution on SmartNIC ARM cores. This software consists of the *decision-making tasks performed by various subsystems*—such as choosing the next thread to schedule or determining the physical memory frame to allocate for a virtual page. Because system software is deployed uniformly across all datacenter machines, the effort required to offload it is invested only once, subsequently providing consistent benefits across the entire infrastructure. Moreover, significant existing work on microkernels [24–41], multikernels [42–50], exokernels [51], and Linux userspace subsystems [52–71] has already established the necessary separation of such systems, simplifying this transition. ARM cores align naturally with this decision-making since decisions typically involve straightforward heuristics, offering a good balance between power efficiency and computational demand. Finally, because cloud providers maintain full control over the host system software, this software is inherently trusted to operate within privileged execution environments on SmartNICs.

The main challenge in offloading system software is overcoming the high PCIe interconnect latency in a manner that works for a wide range of software with different requirements. Thread scheduling and RPC stacks require sub-μs communication [62, 72] as cores remain idle until policy decisions are enforced; memory management requires high-throughput communication to move an entire address space of page table entries from the host to the SmartNIC and a decision for each migrated page back to the host. PCIe presents a distinct challenge because microkernels, some multikernels, and Linux userspace subsystems like ghOSt [62] and Snap [58] target coherent CPUs. Even those adapted for non-coherent CPUs [44–49] benefit from the much lower latencies of the on-chip CPU interconnect. The higher PCIe latency arises from packet-based protocols, flow control, bridge logic, and lower clock rates compared to internal CPU interconnects. Previous approaches, such as Floem [21] and iPipe [73], optimize primarily for throughput using DMA-based queues, which do not satisfy the strict latency and reliability guarantees essential for system software.

We introduce Wave, a framework for offloading system software to the SmartNIC, and use Wave to offload three preexisting system software components. Wave provides a general mechanism for host-SmartNIC communication that enables offloading of even μs-scale system software.

Our first key insight is that *offloading system software to a SmartNIC requires the same API and guarantees that ghOSt [62] provides for running system software in userspace.* The ghOSt API supports atomic commits of policy decisions made by userspace software. This strong guarantee becomes even more essential when the userspace system software that makes decisions operates across a high-latency PCIe interconnect. Wave implements this API with a unidirectional queue for sending messages from the host to the SmartNIC and another unidirectional queue for sending decisions back.

Our next key insight is that *different system software patterns must use different memory mechanisms that provide different performance guarantees.* Wave leverages DMA for high-throughput transfers, which is required by memory management, and MMIO for low-latency communication, which is needed by the thread scheduler and the RPC stack. Even with MMIO, Wave uses different types of page table entries (write-combining and write-through) for different data types. It uses write-combining entries to further reduce the latency of the host-to-NIC queue with a batching optimization and write-through entries to hide the latency of the NIC-to-host queue with prefetching and caching optimizations. Wave also precomputes and prestages decisions to hide the remaining host-SmartNIC communication latency.

As in ghOSt, Wave implements system software in userspace *agents* running on the SmartNIC across the PCIe interconnect. We prototype Wave on Intel Mount Evans [5] and demonstrate practical offload of preexisting system software, including kernel thread scheduling, memory management, and an RPC stack.

We show that Wave communication/notification mechanisms are sufficiently fast (426ns for an agent on the SmartNIC to write a thread scheduling decision and send an interrupt) to make it practical to offload system software to

SmartNIC ARM cores. We further show that Wave's optimizations are essential to hiding the PCIe latency, improving the RocksDB [74] key-value store throughput by ~350% compared to an offloaded system with no optimizations. These optimizations make offload practical, with a small performance degradation of 1.1-7.4% compared to an on-host implementation. Finally, we show that Wave enables practical offload of compute-heavy software, such as machine-learning-based memory management software, which reduces RocksDB's memory footprint by 79%. This compute-heavy software can now leverage SmartNIC compute, saving 16 host cores. We also use Wave to offload an RPC stack, freeing 8 host cores, and reduce interference with virtual machines, improving their turbo performance by 11.2%.

In summary, our key contributions include:

- Transparently offloading three preexisting userspace system software components with full compatibility with the Linux kernel and unmodified applications.
- Building a host-SmartNIC communication API that uses diverse mechanisms to manage the PCIe latency gap and enable μs-scale offloaded system software.
- Providing apples-to-apples comparisons of offloaded vs. on-host system software components that show small offload performance degradation of 1.1-7.4%.
- Saving host resources with offload, e.g., recovering 16 host cores from memory management and 8 from RPC.

2 Offloading System Software

2.1 SmartNICs and IPU Background

SmartNIC Capabilities. SmartNICs are PCIe cards featuring a System-on-Chip (SoC), which consists of a network interface card, compute capabilities in the form of ARM CPUs or programmable P4 pipelines, and hardware accelerators for compression, encryption, RDMA, NVMe virtualization, NVMe-over-fabrics, hardware clock synchronization, etc. [1, 2, 5]. The interface between the host and currently available SmartNICs is limited to memory-mapped IO (MMIO), direct memory access (DMA), and interrupts. The host can access the SoC's DRAM via MMIO operations, which are synchronous memory operations over the PCIe interface that take on the order of 1μs. DMA allows the SmartNIC to place data in the host memory without involving the host CPU, and vice versa. Finally, the SmartNIC can send MSI-X signals (interrupts) to host cores. We analyze the overheads of these operations in our setup in §7.1.

Offloads. Recent work offloads infrastructure functionality to SmartNIC accelerators to leverage their computational power, turning SmartNICs into the *Infrastructure Processing Units (IPUs)* of the datacenter. The AWS Nitro [1] accelerates virtual machine control planes and data planes. AccelNet [10], Dagger [6], FlexTOE [7], 1RMA [8], and eRSS [9] offload a host OS network stack data plane to the SmartNIC.

However, significant barriers prevent utilization of SmartNIC ARM CPUs. Prior work [10, 15–17] identifies the weak performance of the general-purpose ARM cores relative to the host as a significant obstacle to offload. Floem [21] and Mind the Gap [18] demonstrate that host-SmartNIC communication via the slow PCIe interconnect further degrades end-to-end workload performance–in some cases performing worse than with no offload at all–and propose optimizations. Beyond this, offloading workloads piecemeal is not a cohesive vision for the entire datacenter since individual workloads often run on only a subset of the machines.

2.2 Why System Software?

System software is a natural choice for offloading to the SmartNIC ARM cores. It requires general-purpose compute to support a wide range of workloads and hardware, with flexibility being paramount. It parallelizes easily with abstractions that can be shared or pipelined across several cores, such as address spaces in a memory manager and NUMA domains (e.g., chiplets and sockets) in a thread scheduler. System software reduces its interference with on-host workloads by running on the SmartNIC and leverages the unique network insight that a SmartNIC provides to optimize the entire machine. Because every server in the fleet runs system software, each can offload its system software to a SmartNIC, ensuring that any offload benefits apply *universally*.

2.3 How Is Split OS Decision-Making Accomplished?

Offloading system software to the SmartNIC requires the decision-making logic to be separated from the host kernel. Fortunately, a wealth of prior work has already established or leveraged such separation, moving OS decision-making to separate userspace processes, so much of the required structure is already in place. Microkernels [24–41], multikernels [42–50], exokernels [51], and userspace resource management systems [52–71] all illustrate that OS functionality can be effectively split out to run in userspace provided there are robust and efficient mechanisms to communicate with the kernel. Even Linux, a monolithic kernel, includes partial decoupling of certain services—enabling, for example, user-level drivers and specialized control planes—so additional invasive changes are not necessary.

Microkernels. Microkernels [24–41] offload OS services from the kernel to userspace processes, keeping only minimal functionality inside the kernel. Significantly, they emphasize optimized inter-process communication mechanisms since slow IPC in the OS-userspace critical path has traditionally been the drawback of these systems. Several major proposals have led to order of magnitude performance improvements [24, 26], including asynchronous communication; message-passing with a single copy operation; sending a *batch* of several messages; and passing messages to "port-like" IPC endpoint handlers rather than directly to threads.

Multikernels. Multikernels [42–50] like Barrelfish [42–49] propose a new OS design to work across CPUs with many cores, varying cache coherence protocols (or none at all), memory hierarchies, architectures, performance characteristics, and I/O configurations. Multikernels particularly aim to perform well on machines with large numbers of cores and so place one OS image on each core, with each core operating independently. Cores do not share memory with each other to limit coherence traffic and improve scalability. They communicate by passing messages via highly optimized channels, using techniques such as batching, pipelining, asynchrony, and cache-line alignment, taking care to avoid TLB flushes and cache pollution. Barrelfish is implemented on both cache-coherent [42, 43] and non-coherent machines [44–49].

Exokernel. Exokernel [51] exports hardware resources to untrusted userspace applications. Applications implement their own OS services to improve performance. While Exokernel proposes optimizations like asynchronous IPC that Wave can exploit, its focus is largely orthogonal to ours.

Userspace Resource Management. Recent userspace resource management systems bypass the kernel and its associated overheads and scaling constraints, instead implementing their own custom control and data planes in userspace [52–71]. Workloads running on these systems perform significantly better than on vanilla Linux, e.g., 3.6x better throughput and a 2x latency reduction in IX [52]. However, they sacrifice compatibility and require developers to re-architect and port their applications.

As a middle ground, a recent trend offloads to userspace only the control plane of specific subsystems and keeps the data plane in the kernel. This approach maintains compatibility and does not sacrifice performance, while letting users easily specialize and fine-tune policies to their needs. ghOSt [62] and Syrup [63] offload the kernel thread scheduling control plane to userspace. They make it easy to implement custom policies and offer benefits such as rebootless upgrades, fault isolation, and state encapsulation. CCP [64] proposes offloading the congestion control plane to a userspace agent to accelerate experimentation. userfaultfd [65] offloads the page fault handler to userspace applications, which is used to support live migration of virtual machines [75], while PageFlex [76] offloads only the reclamation policy. Userspace tools also have access to interfaces for scanning access and dirty page table bits, which can be used by memory policy decisions [71]. FUSE [66], which lets a userspace application expose a file system to the Linux kernel, has been used to implement over 100 filesystems [67]. UIO [68], Apple DriverKit [69], UMDF [70] and the VFIO driver framework [77] support userspace drivers.

Each system has its own ad hoc scheme for kernel-userspace state sharing and synchronization to prevent time-of-check to time-of-use bugs. State includes network streams, thread runnability, and page table entries. ghOSt uses queues and message barriers and userfaultfd uses file descriptors with blocking syscalls. All communicate over coherent shared memory, so they can use a single queue API, though their performance would degrade over PCIe.

2.4 Why offload to a SmartNIC rather than deploy a heterogeneous host CPU architecture?

Moving system software onto a SmartNIC might appear counterintuitive given the existence of heterogeneous architectures with efficiency cores, such as ARM big.LITTLE [78] and Apple M-series efficiency cores [79]. Some software, such as thread scheduling, consumes ~5% of host cycles [80], so why not dedicate 5 host efficiency cores out of 100+ to the software? One could imagine assigning userspace system software like the Shinjuku scheduler [53] or the ghOSt global agent [62] to dedicated efficiency cores, leaving high-performance cores exclusively for application workloads.

However, this approach is impractical because heterogeneous CPU architectures are not widely available in datacenters. Conversely, SmartNICs are deployed in many new datacenter machines since they offer a simple, cost-effective way to provide uniform security and management across diverse hardware from different vendors, architectures, and software maturities. The SmartNIC thus provides a consistent, universal platform for offloading system software, making it the logical and strategic choice. Additionally, since cloud providers control the design of SmartNICs [81], specialized accelerators tailored to their workloads can be integrated directly into the SmartNIC SoC. This enables tighter hardware-software co-design between I/O operations and system software. Furthermore, since all I/O events pass through SmartNICs in cloud deployments, leveraging this architecture for system software is both natural and advantageous.

3 Wave Design

Wave is a framework for offloading userspace system software to processes/agents running on the SmartNIC. The in-kernel mechanisms that such software relies on remain on the host, which invokes the SmartNIC software over PCIe.

3.1 Wave Overview

Figure 1 shows the high-level Wave design. Wave implements a shared memory queue abstraction backed by a choice of MMIO or DMA over the PCIe interconnect. The host kernel uses the queue to send the required state to the Wave agents, which run the software to make resource allocation decisions. Each agent can run a single system software component or combine software if beneficial. One such example is combining the RPC stack and thread scheduling in §7.3. The agents send their decisions back to the host kernel via a transaction-based API backed by another queue. The host kernel then enforces the decisions. Wave enables existing

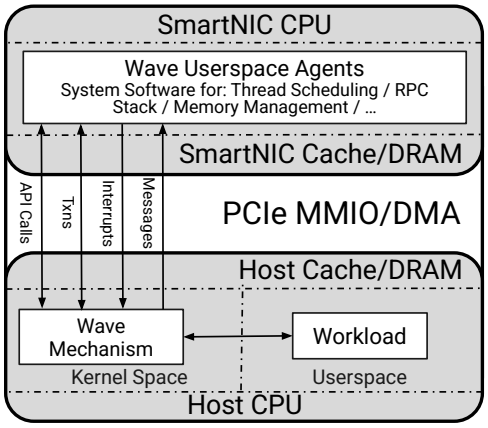


Figure 1. High-level Wave design.

Wave Shared API Calls (both the host and SmartNIC can call these)	
START_WAVE_AGENT()	
KILL_WAVE_AGENT()	
Wave Host API	**Wave SmartNIC API**
Queues	
	CREATE_QUEUE()
	DESTROY_QUEUE()
	ASSOC_QUEUE_WITH(agent, host core)
	SET_QUEUE_TYPE(mmio/dma async/dma sync)
Messages	
SEND_MESSAGES(q = queue)	POLL_MESSAGES(q)
Transactions	
PREFETCH_TXNS(q) (§5.4)	TXN_CREATE(q)
POLL_TXNS(q)	TXNS_COMMIT(q, send/skip msi-x)
Transaction Outcomes	
SET_TXNS_OUTCOMES(q)	POLL_TXNS_OUTCOMES(q)
Custom System Software APIs Can Be Included (As Applicable)	

Table 1. Key Wave API functions. We include certain parameters when they are not intuitive, but omit them otherwise to save space. Multiple messages and transactions can be batched, which is why some API function names are plural.

agent implementations, such as ghOSt agents [62], memory managers [65], and network agents [58], to run on the SmartNIC and enforce decisions on the host.

We now explain how Wave makes decisions using thread scheduling as an example. Figure 2 shows the lifetime of a Wave scheduling decision. A global polling agent model is used, though Wave scales to multiple agents [62].

❶ When Thread A triggers an event (e.g., blocking on a futex, allocating a virtual memory region, etc.) on host CPU core 0, the host sends a message to the agent on the SmartNIC. ❷ The message is added to a message queue, which is backed by the SmartNIC DRAM and accessible via the Wave shared memory abstraction. ❸ The agent, which runs in userspace on the SmartNIC CPU, polls for messages from the queue. Wave system software polls because decisions such as thread scheduling require low latency. Upon reading the message, the agent runs the system software to determine how to allocate resources for that subsystem. ❹ The agent writes its decision to the decision queue in SmartNIC memory. ❺ The agent sends an interrupt (MSI-X) to core 0 on the host to trigger a preemption. ❻ In the interrupt handler, the host kernel on core 0 reads the decision via the Wave shared memory queue, commits it atomically with a transaction, and enforces it, e.g., by context switching to thread B.

Compared with existing solutions that offload system software to userspace [24–49, 51–71], Wave must also achieve competitive performance while moving the high-latency PCIe interconnect directly into the decision-making fast path. We describe how we achieve this in §5.

3.2 Wave API

Wave provides the API in Table 1 for host-SmartNIC communication. The host uses the API to notify SmartNIC agents of state updates via messages and handle decision transactions. The SmartNIC configures queues with the host, reads host messages via polling, and sends decisions, issuing an MSI-X vector to kick the host, as needed. Similar to ghOSt [62], these decisions are committed atomically in the host kernel using transactions. For example, if an agent attempts to update page table entries for an application that simultaneously exits, the transaction will cleanly fail without corrupting host kernel state. The Wave API is general, facilitates batching, and supports a wide range of system software, as described in §4. Furthermore, Wave permits each system software component to implement private extensions to the API, e.g., a memory manager reads access bits from host page table entries, and a file system queries for attached devices.

Communication. Wave uses the lessons learned from prior work (§2) to implement low-latency communication across PCIe, which is essential for achieving performance competitive with that of on-host system software (§7.2). Queues are backed by DMA or MMIO, and DMA queues can operate synchronously or asynchronously. Multiple messages and transactions can be batched.

3.3 Security

Agents are isolated in userspace on the SmartNIC CPU, which restricts their ability to access memory and send MSI-X's. They are further isolated logically, i.e., they cannot read messages or stage decisions for resources they do not manage. Each system software component has an on-host watchdog that kills its agent(s) when it detects they are malfunctioning. For example, the thread scheduler (§4) watchdog terminates an agent that has not made a decision for >20 ms.

4 Offloading System Software with Wave

We offload three preexisting pieces of system software with Wave on Linux: thread scheduling [62], memory management [82], and an RPC stack [83, 84].

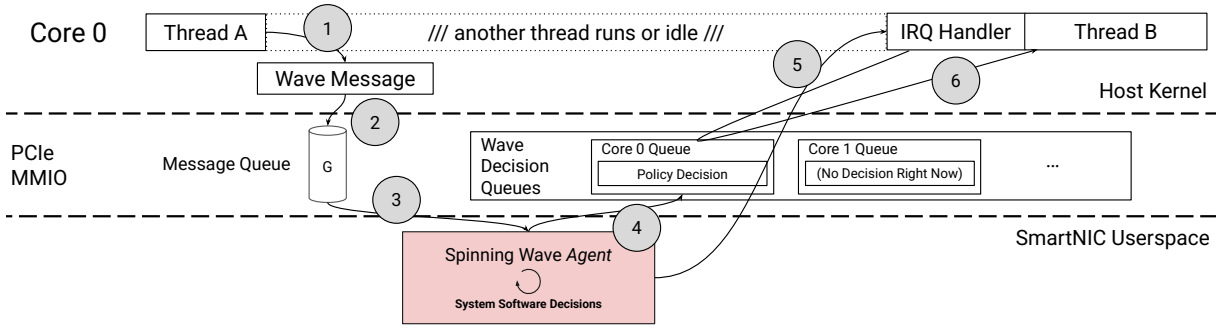


Figure 2. Wave schedules host cores across the PCIe interconnect.

4.1 Kernel Thread Scheduling

The *thread scheduler* places runnable threads onto idle cores and maintains the scheduling policy's run queues and data structures. Application threads operate on timescales ranging from several μs to seconds, though schedulers must be responsive on the μs-scale to ensure critical threads, such as network RX, receive CPU time. Policies include CFS [85], Caladan [57], Shinjuku [53], and others [54, 59, 86, 87].

To schedule threads, Wave offloads policies built with ghOSt [62], a general scheduling class in Linux that runs arbitrary policies in userspace *agents*. The agents receive thread state messages from the kernel (e.g., a thread blocked, woke up, etc.), make scheduling decisions, and commit those decisions to the kernel via a transaction API. The kernel then context switches to the scheduled threads. Communication between the kernel and userspace occurs via shared memory.

As shown in Figure 2, we move the ghOSt agents to the SmartNIC and keep the ghOSt kernel scheduling class on the host. The communication patterns are the same as in ghOSt, and to further improve latency, the SmartNIC agents are always awake and polling for messages.

Communication. The offloaded thread scheduler uses MMIO queues in both directions because scheduling decisions must be made in <1μs to avoid keeping host cores idle. For example, a GET request for the RocksDB [74] key-value store requires 4μs of CPU time, so two 1μs PCIe transfers to schedule a thread degrade throughput by 50%. Low PCIe throughput of ~13.5 KiB/s is required in both directions since the decisions are compact and generated only on thread events, such as when a thread blocks.

Implementation. The Wave thread scheduler integrates ghOSt with the Wave API in §3.2, including `SEND_MESSAGES()`, `PREFETCH_TXNS()` (§5.4), `POLL_TXNS()`, and `SET_TXNS_OUTCOMES()` on the host, and `POLL_MESSAGES()`, `TXN_CREATE()`, `TXNS_COMMIT()`, and `POLL_TXNS_OUTCOMES()` on the SmartNIC. To seamlessly support existing ghOSt policies, we extend the Wave API with ghOSt-specific APIs ("Custom APIs" in Table 1).

Optimizations. The Wave scheduler *prestages* (§5.4) one decision per core so the host can *prefetch* them when a thread blocks or yields. The scheduler eagerly prestages decisions when the run queue length is sufficiently deep (e.g., linear in the number of cores), improving scheduling throughput by 32% in §7.2.

4.2 Memory Management

The *memory manager* decides how to map virtual pages to physical pages. This includes when responding to allocation requests, swapping and compressing pages, and managing page table entries. A memory manager's data structures scale linearly with the size of each process address space, requiring significant memory. Policy algorithms, such as LRU, also require significant compute, so policy designers resort to approximations like the LRU CLOCK algorithm [88]. Other policies include SOL [82], AIFM [89], and AutoNUMA [90].

Design and Implementation. We offload memory management to the SmartNIC with Wave and keep the in-kernel mechanism, such as page fault handlers, page table entries, and TLB shootdowns, on the host. The host uses `SEND_MESSAGES()` to send PTEs, which contain mappings and dirty/access bits, to the SmartNIC. Wave agents make page migration decisions and use `TXN_CREATE()` and `TXNS_COMMIT()` to send updated mappings to the host. Upon receipt of the updated mappings, the host uses the `madvise()` syscall path in the kernel for migrations.

Communication. Transferring PTEs to the SmartNIC and updated mappings back requires a high throughput of 1+ Gbps, so both queues use DMA. Recall that page fault handlers and other fast-path mechanisms are kept on the host. The memory manager itself operates out-of-band, so policy overhead of hundreds of milliseconds is acceptable.

SOL Policy. Recent work proposes machine learning (ML) policies to improve system efficiency across multiple metrics [91–95], including DRAM consumption [82]. *ML requires significant compute, so it is costly to deploy without an offload framework like Wave.* SOL [82] is a recently proposed ML-based policy to classify hot (i.e., frequently accessed) memory pages into a "fast tier" (local DRAM) and cold pages into a "slow tier" (remote DRAM, non-volatile memory, or disk).

At startup, the SOL policy groups consecutive pages into batches for classification. The policy scans page access bits and classifies batches using *Thompson Sampling with a Beta distribution prior* [82, 96]. Each batch is scanned with a frequency ranging from once every 300ms to 9.6s, and batches are moved between memory tiers once per 38.4s epoch (i.e., 4x the slowest scanning frequency). To reduce overhead, SOL determines the optimal frequency to scan each batch's access bits as each scan requires (1) flushing the TLB and (2) policy computation. SOL's overhead is proportional to the address space size and other policy parameters. We offload the SOL policy with Wave and evaluate its performance in §7.4.

4.3 RPC Stack

Recent research [6, 7, 9–12, 18] and RDMA protocols [8, 97, 98] show clear value in offloading RPC stacks to SmartNICs. RPC stacks process packets in a few μs [72] and require significant compute for protocol processing, serialization, compression, and security. They have a large memory footprint due to large packet payloads and stream state. Thus, offload saves host resources and enables acceleration.

We use Wave to offload the Stubby RPC stack [83], similar to gRPC [84], including both a software-based packet-to-host-core steering policy and the data plane. The vanilla on-host Stubby system uses hardware-based RSS [99] on the NIC to steer packets to cores, the host Linux network stack for TCP processing, and CFS for scheduling. After vanilla Stubby processes an incoming RPC, it executes an application-specified callback function to handle the request. The application then executes an RPC callback to send an RPC response, again relying on the host Linux network stack.

We now describe how we offload both the packet steering policy and the data plane to the SmartNIC. As before, this offloaded software is encapsulated in a SmartNIC agent.

Path of an RPC. An arriving RPC packet is steered to the ARM cores on the SmartNIC. The SmartNIC uses its own Linux network stack for TCP processing. The RPC is then passed to the RPC agent, which determines which host core to steer the RPC to. The agent enqueues the RPC in an MMIO queue with `TXN_CREATE()` and sends it to the host core with `TXNS_COMMIT()`. We specify that `TXNS_COMMIT()` (see Table 1) does not send an MSI-X because the host will instead poll the queue to sustain high RPC throughput.

On the host, an RPC-enabled application links with a stub RPC library to make offload transparent. The application polls the request via `POLL_TXNS()`, handles it, and writes the RPC response to an MMIO queue via `SET_TXNS_OUTCOMES()`. The polling agent picks up the response via `POLL_TXNS_OUTCOMES()` and sends it via the SmartNIC Linux network stack.

MMIO for Communication. RPC stacks process RPCs in a few μs [72] before handing them to the application. §7.3 uses small RPC payloads, so our RPC stack processes a moderate throughput of 10s of MiB/s of RPCs. This combination of low latency and low throughput is what drove our decision to use MMIO for RPC host-SmartNIC communciation. A hybrid approach of MMIO with DMA for large packet payloads, proposed by prior work [100], or just DMA alone, would be better for workloads with larger payloads.

Queue Configuration. The Wave agent steers RPCs to specific host cores by stashing them in per-core SmartNIC-to-host queues. There are also per-core host-to-SmartNIC queues for host cores to transfer RPC responses to the agent.

Optimizations. Related policies and systems like Receive-Side Scaling [99], IX [52], Shinjuku [53], Shenango [54], Demikernel [60], and others [57, 84, 101] remove the RPC stack policy from the host kernel. This gap between the kernel and network workload causes load imbalance. The host wastes resources by either over-allocating CPU resources to handle traffic or dedicating host cores to making scheduling decisions after the SmartNIC transfers traffic to host memory [53, 58]. In §7.3, we show the benefits of co-locating the RPC stack with thread scheduling on the SmartNIC.

5 Closing the Host-SmartNIC Latency Gap

Offloading system software with Wave places the slow PCIe interconnect between the host and the SmartNIC agents. As described in §4, each system software component has distinct communication requirements, covering a full spectrum of latency and throughput needs. Fast communication over PCIe and efficient message-passing are extensively studied by prior work and are well understood. Therefore, this section first discusses key insights from this work and how they influenced Wave's communication design (§5.1). We then explore the communication mechanisms we considered (§5.2) before detailing our implementation of Wave's host-SmartNIC queues (§5.3). Wave re-uses the PCIe DMA queue implementation from Floem [21] and adds MMIO support. For MMIO, we explain how using different page table entry types further reduces latency (§5.3). Finally, we discuss how we hide the remaining latency to achieve fast system software decision-making (§5.4).

5.1 Prior Work on Fast PCIe Communication

Prior work [10, 15, 18, 21, 100, 102–105] demonstrates that optimizing the slow PCIe communication path is *critical* to making offload practical. Without these optimizations, end-to-end performance with offload is worse than not offloading at all, even with the additional SmartNIC resources.

Neugebauer et al. [100] propose a theoretical PCIe model and testbench for evaluating PCIe performance. They find that PCIe suffers from high latency (1000ns roundtrip) and that PCIe overheads degrade throughput by >20%. Optimizations are critical to improve performance, such as caching with DDIO (~15% throughput and ~70ns latency improvements), accounting for NUMA locality (10-20% throughput difference), using hugepages to avoid IOMMU overhead (up

to 70% throughput difference), descriptor batching, prefetching, disabling interrupts, and minimizing synchronization.

iPipe [73] uses two unidirectional queues for host-SmartNIC communication and demonstrates a 2-7x speedup when using asynchronous (i.e., non-blocking) DMA rather than synchronous and up to 8.7x speedup when batching transfers. iPipe also uses lazy queue head synchronization to avoid PCIe roundtrips. Floem [21] also implements unidirectional queues with optimizations like asynchronous synchronization (9-15x speedup), pruning unused packet data (1.2-3.1x speedup), caching, batching, and hiding I/O.

Wave leverages the insights from this prior work and builds on Floem's queue implementation for fast host-SmartNIC communication. DMA is best for high-throughput transfer that is latency insensitive because several MMIO accesses are required to initiate DMA. MMIO is best for low-latency, low-throughput communication. Wave similarly uses batching, caching, and prefetching; disables interrupts under heavy load; avoids unnecessary synchronization to reduce roundtrips; and accounts for NUMA locality.

OS Contexts. Microkernels [24–41], multikernels [42–50], exokernels [51], and userspace resource management systems [52–71] optimize kernel-userspace communication. These systems typically employ *message-passing* over shared-memory queues. Wave uses message-passing and leverages their optimizations, including asynchronous communication, limited copying, batching, and cache-line alignment.

5.2 Our SmartNIC's Communication Mechanisms

Our SmartNIC supports DMA and MMIO over non-coherent PCIe.

Direct Memory Access (DMA). The SmartNIC has a DMA engine that supports bidirectional memory transfer between host DRAM and the SmartNIC SoC DRAM. When the host CPU is the producer, entries are written to host DRAM and DMA'd to the SmartNIC SoC DRAM. Conversely, when the SmartNIC is the producer, the opposite occurs.

Memory-Mapped I/O (MMIO). Like most other SmartNICs, the SmartNIC that we use exposes an MMIO interface to the host so that the host can read/write a subset of the SmartNIC memory, which the SmartNIC cores otherwise have fast, coherent access to via the SoC. The MMIO interface is backed by a hardware function in the SmartNIC that operates without CPU intervention. As shown in Table 2, the direct CPU overhead of a 64-bit host MMIO write is ~50 ns and a read is ~750 ns. Writes have less CPU overhead than reads because they are not acknowledged, while reads must wait for the PCIe roundtrip to complete before the read values are available. The host can use the MMIO region to access the SmartNIC DRAM, but the SmartNIC cores cannot access host memory without DMA. The MMIO mechanism throughput and latency are bounded by the hardware capabilities of the PCIe interconnect and the SmartNIC. Even so, the host MMIO overheads are not trivial and could still become bottlenecks for microsecond-scale workloads. We reduce these overheads further in §5.3.1.

Coherent Interconnects. New coherent interconnects such as CXL [106], UPI [107], and NVLink [108] provide a shared coherent memory address space between the SmartNIC and host. Once such interconnects are widely used, they will accelerate host MMIO accesses. SmartNIC SoC memory will be cacheable on the host, so the host can hide read latency by prefetching and reusing MMIO reads. MMIO writes will also be faster since the host can store a batch of writes in the host cache hierarchy and flush the batch to PCIe. However, DMA will remain preferred for high-throughput, latency-insensitive workloads. MMIO with CXL requires CPU coordination for memory transfer, while DMA offloads this coordination to the DMA engine, freeing up CPU cycles. And even though coherent interconnects improve low-latency MMIO, the Wave optimizations in §5.3.1 and §5.4 remain necessary to make performant offload practical.

5.3 Host-SmartNIC Communication in Wave

Like microkernels and multikernels, Wave uses message-passing over shared-memory queues for host-SmartNIC communication. It re-uses the Floem [21] DMA unidirectional queue and adds support for MMIO to achieve the best of both worlds: low-latency and high-throughput communication. We further optimize the MMIO queue to reduce latency.

DMA. The producer writes one or more entries to the Floem queue and initiates a DMA transaction to send the entries to the consumer. The producer can wait synchronously until the transaction completes or it can check for completion *asynchronously*. Messages can be *batched*, and they are written to the *local NUMA node* of the recipient. To ensure the consumer does not read inconsistent entries while the producer is still writing, the producer sets a flag in each entry *after* it finishes writing the entry. If the consumer sees the per-entry flag set, the entry is valid to read.

MMIO. Wave MMIO message queues are identical to Floem DMA queues, with the same data layout and flag synchronization scheme. As only the SmartNIC exposes its DRAM via MMIO, the queues are always backed by SmartNIC DRAM, regardless of which side is the producer or consumer. Thus, the host accesses these queues via MMIO while the SmartNIC agents access them via local, coherent memory. MMIO queues also leverage optimizations like batching (§5.3.1), caching (§5.3.2) and prefetching (§5.4).

5.3.1 Reducing MMIO Latency.

MMIO is in the critical path of system software that requires very low (<1 μs) latency, such as thread scheduling (Figure 4) and an RPC stack (Figure 6). Host MMIO reads, in particular, are very expensive (750ns) due to the PCIe roundtrip. To further reduce MMIO overheads, Wave leverages the *caching* and *prefetching* optimizations proposed by prior work. To do so, it is

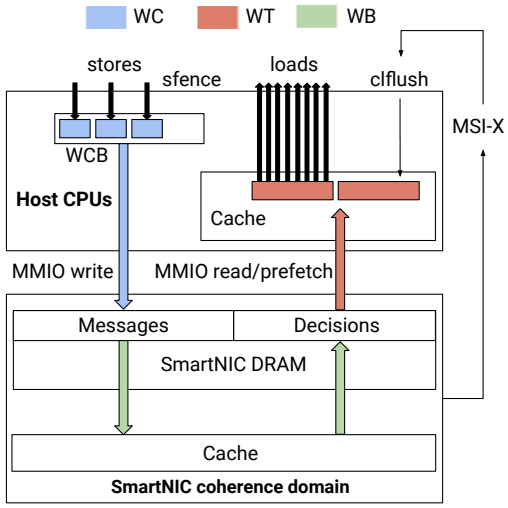


Figure 3. Wave improves performance by using caches and multiple page table entry types.

important to set page table entry (PTE) types to cache MMIO contents. Figure 3 shows how Wave sets PTE types and uses caches. Wave uses these PTE types:

Write-Back (WB). The CPU caches and implements coherence for this type of memory (e.g., local DRAM). This is the default mapping for userspace memory.

Write-Combining (WC). The CPU does not cache reads or implement coherence, instead loading directly from memory. Stores go to a write-combining buffer, so they complete much faster than going to memory. The buffer drains to memory periodically and is flushed explicitly with `sfence`.

Write-Through (WT). The CPU stores directly to memory. However, loads are cached, so subsequent loads from the same cache lines are fast.

SmartNIC CPU PTEs. MMIO queues are backed by SmartNIC DRAM. The agents have local, coherent access to this memory, which is mapped into their address space with WB.

Host CPU PTEs. There is non-coherent PCIe between the host CPU and SmartNIC memory, so the host cannot map the MMIO region with WB. However, the host can improve write performance with WC instead of having no caching at all (i.e., uncacheable PTEs). The host can now enqueue a message *batch* before the write-combining buffer is flushed.

5.3.2 Caching MMIO Content. A 64-bit host MMIO read takes ~750 ns, so it is expensive for the host to read decisions, especially those that span multiple words. To reduce overhead, we mark the MMIO queues as WT. On each 64-*bit* read to an MMIO address, the host CPU transfers the entire 64-*byte* cache line into its cache, with the same ~750 ns overhead. Subsequent reads to the same cache line hit the cache, making it cheap to read a batch of decisions.

With this approach, however, stale decisions linger in the host CPU cache, so Wave implements a software-based coherence protocol. When an agent stashes a new decision, the host flushes the stale data from the host CPU cache with `clflush`. For example, the host flushes upon receiving an MSI-X from the SmartNIC as it knows a new decision is available, so the currently cached decision is stale.

5.3.3 Conclusion. Writing messages is cheap with WC. Reading decisions is cheap with WT as read overhead is amortized across a cache line and subsequent reads hit the cache. We next hide the remaining read latency entirely.

5.4 Hiding the Remaining Latency

Despite our optimizations, systems offloaded with Wave still place PCIe in the critical path, which causes higher decision latency than on-host solutions. Wave lets these systems hide the remaining latency through prestaging and prefetching.

Prestaging Decisions. When the host requires a decision (Figure 2), it experiences a long wait of several μs before the SmartNIC agent receives a message and sends a decision back. During this wait, the host CPU cannot execute the workload, leading to low host utilization. A Wave agent can make decisions ahead of time–before the host requests a decision–and *prestage* those decisions in SmartNIC DRAM. When the host requires a decision, the host immediately checks the MMIO region for a decision and acts on it if present. The Wave agent is not invoked on the critical path, significantly reducing decision latency. This approach leads to a substantial improvement in workload performance (32% in §7.2) and decouples host utilization from decision latency.

When is prestaging possible? Prestaging is possible when the agent can make decisions before asked. A memory management agent can detect a strided host access, preallocate physical frames that will soon be needed, and then prestage new page table entries. A thread scheduler with a deep run queue can prestage a runnable thread on each core.

Prefetching MMIO Decisions from the Agent. When the host needs a decision, it updates kernel state and sends a message to the Wave agent on the SmartNIC; it then reads a prestaged decision. The decision queue MMIO is marked as WT, which supports prefetching. Rather than issue a blocking load for the prestaged decision right when the decision is needed, the host can now issue a memory prefetch for the decision *before* updating the kernel state and sending a message. This work takes about 1μs, which is sufficient to completely hide the MMIO read latency, so the subsequent host load for the decision will hit the host cache. Thus, when decisions are prestaged, prefetching a decision reduces the cost of MMIO reads to zero. §7.2 shows that with these optimizations, Wave closes the host-SmartNIC latency gap.

6 Lessons Learned from Wave Agents

We now summarize lessons learned from our experience offloading preexisting userspace system software with Wave.

Minimize or Hide PCIe Overheads. The PCIe interconnect is in the host-SmartNIC critical path. System software must account for PCIe overheads to remain performant, akin to how an OS must consider cross-NUMA transfers in a multi-socket machine. It should leverage insights from prior work on fast communication between OS contexts and across PCIe. These insights include choosing the right transport mechanism (setting DMA or MMIO via `SET_QUEUE_TYPE()`), minimizing state sharing [21], batching (e.g., multiple txns in `TXNS_COMMIT()`), asynchronous communication, caching, prefetching (`PREFETCH_TXNS()`), and NUMA locality.

Keep Agents Modular. Management, engineering, and debugging complexity is reduced when agents encapsulate policy state as they seamlessly operate across SmartNICs and operating systems, and in host userspace when a SmartNIC is unavailable. It is also easier to experiment with different system software components and offload techniques [21].

Keep Fault Recovery Simple. An agent may crash or be killed in preparation for an upgrade. An operator may wish to restart the agent or fall back to vanilla on-host system software. In both cases, recovery is easier when the host kernel is the *source of truth* for non-policy state, such as page table entries for a memory manager or thread TIDs for a scheduler. Both a restarted agent and vanilla on-host system software pull this information upon launch and continue running. Complicated checkpointing or state recovery increase complexity and create bugs that may require a machine reset.

Focus on Host Partitions and Agent Scalability. Datacenter machines have hundreds of CPU cores, accelerators, TiBs of DRAM and flash, and 100+ Gbps NIC bandwidths. Multiple applications/tenants share these resources but want different policies. Developers should partition host resources into logical units, each with their own agent and policy, following the proven approach of ghOSt *enclaves*. The scheduling agent in §7.2 operates per CCX and the memory agent in §7.4 manages a single address space. Developers should also parallelize an agent with threads. Each memory agent thread in §7.4 manages an address space chunk.

7 Evaluation

Wave is implemented on the 4.15 Linux kernel on the host and the 5.10 kernel on the SmartNIC.[1] We use the Intel Mount Evans SmartNIC [5], which has an ARM Neoverse N1 [109] CPU @ 3.0 GHz with 16 physical cores and a network function with 200Gbps of throughput. The host has an AMD Zen3 CPU @ 2.45 GHz with 2 sockets, 64 physical cores per socket, and 2 hyperthreads per physical core. The host CPU frequency ranges from 2.45 GHz to up to 3.5 GHz with turbo boost. The CPU contains core complexes (i.e., AMD CCX's) with 8 physical cores and a private L3 cache per CCX. Unless otherwise noted, we run workloads on the first hyperthread of each host physical core and leave the second sibling idle.

[1]We use these versions of Linux on our production machines. Wave and the offloaded system software also work with the newest versions of Linux.

MMIO	
1. Host MMIO 64-bit Read (Uncacheable)	750 ns
2. Host MMIO 64-bit Write (Uncacheable)	50 ns
MSI-X	
3. MSI-X Send (Register Write)	70 ns
4. MSI-X Send (Ioctl + Register Write)	340 ns
5. MSI-X Receive	350 ns
6. MSI-X End-to-End	1,600 ns

Table 2. Hardware microbenchmarks. Results are rounded to 1-2 leading digits. MSI-X End-to-End includes PCIe latency.

We present microbenchmarks that show Wave has low communication overheads that make offload practical. We then offload preexisting system software for thread scheduling, an RPC stack, and memory management to the SmartNIC and provide an apples-to-apples comparison to on-host software. Due to our optimizations, end-to-end performance is only slightly worse with Wave despite PCIe overheads. When applications leverage newly freed host resources due to the offloaded software, they outperform on-host deployments.

7.1 Microbenchmarks

Results of MMIO and MSI-X Evaluation. In Table 2, host MMIO reads are 750ns due to the PCIe roundtrip and writes are 50ns as they are not acknowledged. Wave uses SmartNIC memory for communication and, thus, NIC-side latencies are those of a regular memory access. MSI-X overheads are comparable to interprocessor interrupts, though end-to-end latency is higher due to the one-way PCIe trip.

Results of Scheduling Evaluation. Table 3 contextualizes these overheads by comparing a scheduler offloaded with Wave with on-host ghOSt for a FIFO thread scheduling policy. With no optimization, Wave's overheads are several μs higher than ghOSt's, though optimizations (§5) bring Wave quite close.

Conclusion. Wave has cheap communication and notification mechanisms that make offload practical. Wave's microbenchmarks will never match on-host due to the inherent cost of offload across PCIe, though they come close.

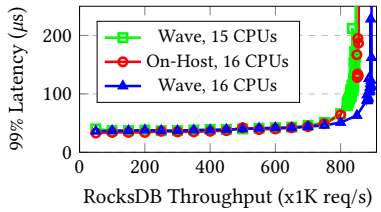
7.2 Thread Scheduler

We offload three preexisting ghOSt policies with Wave and compare them with their on-host implementations in ghOSt to demonstrate the overheads of offload.

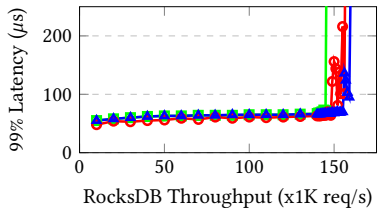
7.2.1 Ported ghOSt Policies. We first port a **run-to-completion FIFO policy** and use it to schedule RocksDB [74], an in-memory key-value store with μs-scale requests. We chose this policy because it requires little compute but interacts extensively with the workload, stressing Wave's API and PCIe queues and making the cost of offload clear.

Offloaded Kernel Thread Scheduler with Wave	
1. Open a Decision in Agent & Send MSI-X	
Baseline (§5.3)	1,013 ns
with WB PTEs on SmartNIC (§5.3.1)	426 ns
2. Context Switch Overhead on Host	
Baseline (§5.3)	13,310-13,530 ns
with WB PTEs on SmartNIC (§5.3.1)	9,940-10,160 ns
and with WC/WT PTEs on Host (§5.3.1)	6,100-6,910 ns
and with Pre-Staging & Prefetching (§5.4)	3,320-4,040 ns
On-Host ghOSt Scheduler	
3. Open a Decision in Agent & Send Interrupt	770 ns
4. Context Switch Overhead on Host	
Baseline	4,380-4,990 ns
with Pre-Staging (§5.4)	2,350-3,260 ns

Table 3. Scheduling microbenchmarks. We ran context-switch benchmarks five times and report the range of medians. "Prestaging" has more variability as prestages may fail.



(a) FIFO scheduling policy for 10μs GET requests.



(b) Shinjuku for 99.5% 10μs GET and 0.5% 10ms RANGE.

Figure 4. Wave implements μs-scale preemptive scheduling and is competitive with on-host scheduling.

We next port a **Shinjuku [53] policy** and use it to schedule RocksDB. We chose Shinjuku because it also maintains a FIFO queue but preempts threads that exceed a time slice, making the overhead of MSI-X clear.

Last, we port **our custom kernel scheduling policy for GCE, our production virtual machine service** [110], to Wave. vCPUs in our VM service run for several milliseconds continuously before requiring scheduler intervention. This policy shows that Wave's API is sufficiently expressive to support production policies and that Wave suffers negligible loss of performance when scheduling ms-scale workloads.

For all policies, we perform an apples-to-apples comparison between the policy on the SmartNIC and the policy on the host. When the policy is on the SmartNIC, we do not let the workload use the newly freed host resources that the policy would have consumed if it were on the host. This clarifies the cost of offload. Subsequently, we let the workload use these newly freed resources to show the benefit of offload. In all cases, both offloaded and on-host policies either consistently use or do not use the prestaging optimization together. Offloaded policies alone may use the Wave-specific optimizations of WC/WT page table entries and WT prefetching.

7.2.2 FIFO Evaluation. We compare a FIFO policy in both Wave and on-host ghOSt and drive RocksDB with 10 μs GET requests. In both cases, one agent thread is pinned to a core and schedules RocksDB. Both prestage threads (§5.4).

Comparison Scenarios. We compare three scenarios: *On-Host:* On-host ghOSt with 16 host cores. One runs a ghOSt agent and the other 15 run worker threads. *Wave-15:* Wave with 15 host cores, all of which run worker threads, the same number as On-Host. *Wave-16:* Wave with 16 host cores, all of which run worker threads. Wave frees a host core that on-host ghOSt dedicates to an agent, so this core runs workers.

Results of Apples-to-Apples Comparison. In Fig. 4a, Wave-15's tail latency is 3μs higher than On-Host and saturates 1.1% lower due to PCIe overhead. Wave-16 saturates 4.6% higher than On-Host due to the extra host core freed.

Results of Applying Optimizations (§5). We repeat Wave-16 without optimizations and successively add them.

	Saturation Tput
Baseline (No Optimizations, §5)	258,000
+ SmartNIC WB PTEs (§5.3.1)	520,000 (+102%)
+ Host WC/WT PTEs (§5.3.1)	680,000 (+31%)
+ Prestage and Prefetch (§5.4)	895,000 (+32%)

Optimizing PCIe communication with WC/WT page table entries still falls far short of Wave-16's saturation throughput in Figure 4a. When latency is *hidden* with prestaging and prefetching, throughput further increases by 32%. Wave-16 now saturates 4.6% higher than On-Host, which is lower than an expected improvement of (16 / 15) - 1 = 6.7% from an extra host core due to PCIe overhead.

7.2.3 Shinjuku Scheduling Policy Evaluation. Shinjuku [53] is a round-robin policy with time-based preemption. We use this policy to show that MSI-X interrupts are a reasonable alternative to inter-processor interrupts.

Results of MSI-X Evaluation. Shinjuku preempts requests that exceed a time slice so short requests do not suffer inflated latency when stuck behind long requests. The load generator produces 99.5% of requests as 10μs GETs and 0.5% as 10ms RANGE queries. The preemption time slice is 30μs. Both Wave and on-host ghOSt prestage. However, prefetching in Wave (§5.4) is ineffective when a preemption occurs

as the host immediately reads the next decision upon receipt of the MSI-X, so it cannot overlap the PCIe read with work.

Results of Apples-to-Apples Comparison. In Fig. 4b, Wave-15 has tail latency 5μs higher than On-Host and saturates 7.6% lower due to PCIe overhead. Wave-16 leverages the extra host core to saturate 1.9% higher than On-Host, but this falls short of the expected 6.7% improvement mentioned. Wave-15 and Wave-16 both perform worse relative to On-Host than with FIFO due to lack of prefetch on preemption.

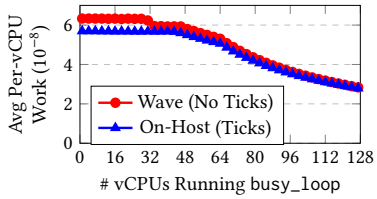
7.2.4 Virtual Machine Scheduling Policy Evaluation.

Virtual machines run on top of a hypervisor in the host kernel, with their vCPUs scheduled by the host kernel's thread scheduler every few milliseconds. Even when a vCPU is idle and there are no other runnable tasks on a physical core, the host scheduler still receives timer ticks and wakes up before immediately yielding back to the guest. This unnecessary *interference* prevents the physical core from entering deep sleep states, *limiting the turbo boost potential of other cores.*

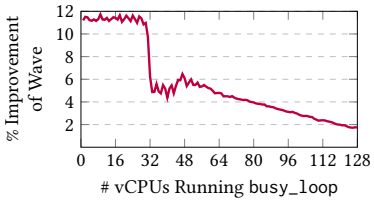
Results of Offloading Custom Kernel Scheduling. A prime candidate for offload is our custom kernel scheduling policy, designed specifically for GCE [110], our production virtual machine service. This policy, inspired by Tableau [111], prioritizes fair CPU sharing among all vCPUs while imposing an upper bound on tail latency. Under this policy, vCPUs run for a time quantum ranging from 5-10 ms but can be preempted at 1-ms granularity. This fine-grained control ensures fairness as vCPUs may consume varying amounts of CPU time within their assigned quantum.

Each host core independently schedules its tasks, so our production machines deliver timer ticks to every core once per millisecond. This high tick frequency prevents idle vCPUs from entering deeper power-saving modes, constraining the turbo boost capabilities of other cores. If the on-host scheduler used a single polling scheduling instance instead, we could disable timer ticks. However, ghOSt [62] shows that this model has scalability limits when co-located with the workload it schedules [62] due to hyperthread interference and coherence traffic. We also could not provide the instance's core to cloud clients or provide machine-sized VMs, which reduce noise and administrative overheads and are an original motivation for SmartNICs like the Nitro [1].

Evaluation Background. We compare this VM scheduling policy in Wave and on-host ghOSt. As VMs are scheduled at ms-granularity, neither policy uses prestaging, and the Wave policy also does not use prefetching. In both cases, two VMs are scheduled, each with 128 vCPUs. The VMs are multiplexed across 128 logical cores (64 physical cores) in a single socket. We limit our focus to one socket because turbo is constrained on a per-socket basis, i.e., one socket's activity does not impact the other sockets' clock frequency. In each VM, we run the `busy_loop` utility, which consumes cycles with arithmetic operations and system calls. We use this utility internally to characterize compute performance and generate turbo frequency curves for new hardware.



(a) For each scenario, there are two VMs with 128 vCPUs each, scheduled on a 128 core socket.



(b) The % improvement of Wave (No Ticks) over On-Host ghOSt (Ticks).

Figure 5. Virtual machine compute performance when scheduled by Wave (no timer ticks) vs. on-host ghOSt (ticks).

Results. In Figure 5, we run `busy_loop` on one vCPU in each VM with all other vCPUs idle. We spawn additional instances of `busy_loop`, one per vCPU, and run the utility on the first hyperthread of all physical cores before using neighboring hyperthreads. Figure 5a compares the work output of both the Wave scheduler and the on-host baseline, and Figure 5b shows the percentage improvement achieved by the Wave scheduler. The Wave scheduler shows an 11.2% improvement over on-host when 1 vCPU is active, 9.7% when 31 vCPUs are active, and 1.7% when 128 vCPUs are active. When many vCPUs are idle, active vCPUs save the tick overhead and experience a turbo boost. Once 32 vCPUs are active, AMD's turbo governor applies a smaller turbo boost. Once 64 vCPUs are active, the first sibling on each physical core is busy, and we start running `busy_loop` on the second siblings. When 128 vCPUs are active, no vCPUs receive a turbo boost, so the 1.7% improvement is solely timer tick overhead savings. Using one SmartNIC core for scheduling saves $1.7\% * 256 = 4.4$ cores per host; at fleet scale, even a small improvement like this creates significant cost savings.

7.3 RPC Stack

The SmartNIC is the entry point for all incoming packets, so it can make better packet steering and thread scheduling decisions that *save host compute and improve cache locality and tail latency.* We offload both the packet-to-host-core steering policy for Stubby [83] (§4.3) and the thread scheduler (§4.1) to the SmartNIC, co-locating them for improved coordination. The RPC data plane has already been offloaded in

prior work [6–12], so we retain this design while additionally offloading the packet-to-host-core steering policy.

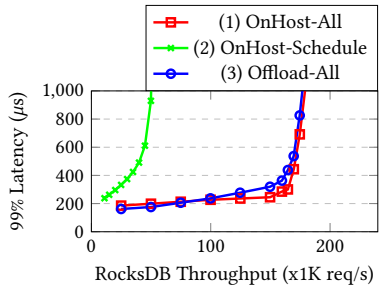
7.3.1 RocksDB Evaluation. RocksDB runs on the host and a load generator under the same top-of-rack switch serves the system with RocksDB RPC requests, with 99.5% of requests as 10μs GETs and 0.5% as 10ms RANGE queries. The RPC stack and RocksDB run in different processes and pass requests/replies via shared memory, i.e., host DRAM when the RPC stack is on-host and PCIe MMIO when offloaded.

Comparison Scenarios. We compare three scenarios:

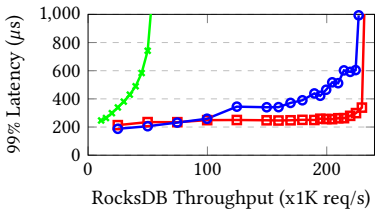
1. ***OnHost-All.*** The ghOSt scheduler and the RPC stack are both on host. All communication is via host shared memory. The RPC stack uses 8 cores, the scheduler one, and RocksDB 15. ghOSt performance degrades when it is located in a different CCX than RocksDB, so we co-locate ghOSt and RocksDB in the same two CCXs while the RPC stack runs in a third CCX.
2. ***OnHost-Scheduler.*** The scheduler runs on the host while the RPC stack is offloaded to the SmartNIC with Wave. The scheduler and RPC stack communicate via MMIO so that the scheduler knows about incoming RPCs, and RocksDB and the RPC stack also use MMIO communication. The 8 host cores used above for RPCs are now free. ghOSt uses 1 core, and RocksDB uses 15.
3. ***Offload-All.*** The scheduler and RPC stack both run on the SmartNIC with Wave, communicating via SmartNIC DRAM. RocksDB uses all 16 host cores and MMIO for cross-PCIe communication with the RPC stack.

Scheduler-RPC Synergy. We run the *single-queue Shinjuku policy* [53] and preempt threads that exceed a 30μs time slice. This ensures that in a highly dispersive workload, short requests do not get stuck behind long ones. Figure 6a shows that by running the RPC stack and scheduling on the same node (host or SmartNIC), OnHost-All and Offload-All achieve about identical performance, though Offload-All recovers 9 host cores. Although Offload-All dedicates one more host core to RocksDB, RocksDB suffers from PCIe overhead, and so performs comparably to OnHost-All. The offloaded RPC stack does not currently leverage prestaging and prefetching, and we attribute the latency gap to this. When the RPC stack alone is offloaded in OnHost-Schedule, the on-host scheduler must access the RPC headers via MMIO loads to schedule threads, saturating at a much lower throughput.

Results of Apples-to-Apples Comparison. RocksDB has one more host core in Offload-All (16 cores) than OnHost-All (15). In this experiment, throughput is linear with the number of host cores as the remaining resources on host and the SmartNIC are not saturated. The throughput with 15 cores is X*(15/16), so when Offload-All restricts RocksDB to 15 on-host cores, it performs 6.3% worse than OnHost-All.



(a) Single-queue Shinjuku.



(b) Multi-queue Shinjuku using RPC request SLO.

Figure 6. RocksDB performance when running the RPC stack and the scheduler on the host or on the SmartNIC.

7.3.2 Leveraging Network Insights. We port the *multi-queue Shinjuku policy*, which leverages RPC-specific information to provide better performance isolation between SLO classes. Each RPC request includes an SLO in its payload, which the RPC stack passes to the scheduler. The scheduler assigns the request to an idle RocksDB thread and adds the thread to a per-SLO run queue. The Wave agent then assigns threads to idle host cores. The preemption time slice is 30 μs.

Results of Multi-Queue Shinjuku Policy. In Figure 6b, Offload-All saturates 20.8% higher than single-queue Shinjuku because it leverages RPC information on the SmartNIC. Offload-All saturates within 2.2% of OnHost-All although the latter uses 9 more host cores. The 2.2% gap is due to PCIe overhead. In OnHost-Schedule, the overhead of reading the SLO (not just the RPC header) via PCIe dominates and leveraging the included SLO has no impact, so the gap widens between OnHost-Schedule and the other scenarios relative to single-queue Shinjuku. This shows that to offload/accelerate an RPC stack and effectively leverage the included SLOs, it is *essential* to also offload scheduling.

Results of Apples-to-Apples Comparison. When Offload-All restricts RocksDB to 15 host cores as above, it performs 7.4% worse than OnHost-All due to PCIe overhead.

7.3.3 Faster Interconnects Benefit Wave. PCIe is the bottleneck in Wave for μs-scale software. Coherent interconnects like CXL [106], UPI [107], and NVLink [108] improve performance and eliminate the need for software coherence.

Evaluation Background. To show that Wave systems take advantage of coherent interconnects, we emulate a UPI-attached SmartNIC using the host CPU in one socket, and we

use the other socket's CPU as the host. A UPI link connects both sockets, and we re-implement the optimizations in §5. We use AMD's HSMP frequency scaling driver to set the turbo boost limit to emulate the slower SmartNIC cores. The host socket runs at 3.5GHz, and we test three different frequency limits for the emulated SmartNIC socket: 3GHz (our SmartNIC SoC frequency), 2.5GHz and 2GHz.

We compare two scenarios. *Offload:* The Wave scheduler and RPC stack run on the emulated SmartNIC in one socket and RocksDB runs in the other socket. *On-Host:* The Wave scheduler and RPC stack along with RocksDB all run in the same socket, and we run the system at the default 3.5GHz host frequency. In both scenarios, RocksDB uses the same number of cores, so this is an apples-to-apples comparison.

Results of Apples-to-Apples Comparison. The slowdowns at saturation in offload relative to on-host are 1.3% (offload @ 3GHz), 2.5% (2.5GHz) and 3.5% (2GHz). At 3GHz, UPI yields an improvement of 0.9% over Wave running on our real PCIe-connected SmartNIC. Wave uses shared memory and PCIe effectively, performing well without hardware cache coherence but benefiting from it when available.

7.4 Memory Management

We next offload SOL [82], described in §4.2.

7.4.1 Evaluation Background. The Wave SOL agent uses DMA to transfer RocksDB's page table entries to the SmartNIC and page migration decisions to the host. The RocksDB database has 10 billion key-value pairs and is ~100 GiB. We group pages into 256 KiB batches (64 x 4 KiB pages), and scan each batch with periods ranging from 600ms, 1.2s, 2.4s, ..., 9.6s. Once per 38.4s epoch, fast batches are migrated to DRAM and slow batches are swapped to disk. There is one load generator thread that creates 10μs GET requests and 14 RocksDB workers, each affined to a unique core. We pin the load generator and RocksDB threads to host cores in the same two CCX's (8 physical cores per CCX) and schedule them with the default on-host Linux kernel scheduler. The agent runs either on the host (in different CCXs than the load generator and RocksDB) or on the SmartNIC. SOL requires significant compute well beyond one core, so we parallelize the policy and dedicate up to 16 cores. In each trial, the agent uses the same number of SmartNIC cores and host cores.

7.4.2 Results of Apples-to-Apples Comparison. The table below shows an apples-to-apples comparison of the per-iteration agent loop duration. The duration decreases with more cores. Portions of the SOL policy are serial, so the duration does not decrease linearly with the core count.

# Cores	Wave (ms)	On-Host (ms)
1	1,018	623
2	576	431
4	437	354
8	384	322
16	364	309

The Overheads. Transferring the page table entries with DMA for the entire RocksDB address space takes ~1ms. The entire address space is transferred on the first agent iteration, but subsequent transfers are faster as SOL learns which page batches to scan less frequently. Transferring page migration decisions to the host with DMA takes <1ms on all iterations as only a subset of pages is migrated. The agents spend most of their time on policy computation, so the offloaded software's per-iteration duration is higher than on-host because it uses weaker ARM cores rather than x86 host cores. Despite this higher duration, the offloaded software approaches the 300ms period in the SOL paper [82], making it practical to deploy SOL without sacrificing 16 host cores.

Effect on RocksDB. SOL reduces RocksDB's memory usage from ~102 GiB DRAM at startup to ~21.3 GiB DRAM (79% reduction) after 3 epochs. Each GET request is 10μs, so SOL's impact on RocksDB performance is minimal, with a median GET latency of 12μs and a tail (99%) of 31μs.

8 Conclusion

Wave demonstrates that system software is a practical and effective workload for mid-tier SmartNIC ARM cores, enabling cloud providers to recover host resources while sacrificing minimal performance. By leveraging a tailored host-SmartNIC communication API and optimizing for both high-throughput and low-latency interactions, Wave manages the PCIe bottleneck and achieves efficient offload of system software. Ultimately, Wave highlights the potential for rethinking system software placement in modern datacenters, unlocking new opportunities for efficiency and scalability.

Acknowledgments

We thank our anonymous reviewers, Thomas Wenisch, Jeff Mogul, Steve Gribble, and our shepherd, Dan Tsafrir, for their helpful feedback. Jack Humphries was partially supported by the NSF Graduate Research Fellowship. Jack Humphries and Christos Kozyrakis were partially supported by the Stanford Platform Lab and its affiliates, and by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

References

[1] AWS Nitro System. https://aws.amazon.com/ec2/nitro/. Last accessed: 2020-11-29.
[2] NVIDIA BlueField Networking Platform. https://www.nvidia.com/en-us/networking/products/data-processing-unit/. Last accessed: 2024-10-06.
[3] Fungible. https://www.fungible.com/product/dpu-platform/. Last accessed: 2022-12-06.
[4] AMD Pensando. https://www.amd.com/en/accelerators/pensando. Last accessed: 2022-12-06.
[5] Intel® Infrastructure Processing Unit (Intel® IPU) ASIC E2000. https://www.intel.com/content/www/us/en/products/details/network-io/ipu/e2000-asic.html. Last accessed: 2023-08-04.

[6] Nikita Lazarev, Shaojie Xiang, Neil Adit, Zhiru Zhang, and Christina Delimitrou. Dagger: Efficient and fast rpcs in cloud microservices with near-memory reconfigurable nics. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 36–51, New York, NY, USA, 2021. Association for Computing Machinery.

[7] Rajath Shashidhara, Tim Stamler, Antoine Kaufmann, and Simon Peter. FlexTOE: Flexible TCP offload with Fine-Grained parallelism. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 87–102, Renton, WA, April 2022. USENIX Association.

[8] Arjun Singhvi, Aditya Akella, Dan Gibson, Thomas F. Wenisch, Monica Wong-Chan, Sean Clark, Milo M. K. Martin, Moray McLaren, Prashant Chandra, Rob Cauble, Hassan M. G. Wassel, Behnam Montazeri, Simon L. Sabato, Joel Scherpelz, and Amin Vahdat. 1RMA: Re-Envisioning Remote Memory Access for Multi-Tenant Datacenters. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 708–721, Virtual Event, USA, 2020. Association for Computing Machinery.

[9] Alexander Rucker, Muhammad Shahbaz, Tushar Swamy, and Kunle Olukotun. Elastic rss: Co-scheduling packets and cores using programmable nics. In *Proceedings of the 3rd Asia-Pacific Workshop on Networking 2019*, APNet '19, page 71–77, New York, NY, USA, 2019. Association for Computing Machinery.

[10] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. Azure accelerated networking: Smartnics in the public cloud. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, NSDI'18, page 51–64, USA, 2018. USENIX Association.

[11] Maroun Tork, Lina Maudlej, and Mark Silberstein. Lynx: A smartnic-driven accelerator-centric architecture for network servers. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 117–131, New York, NY, USA, 2020. Association for Computing Machinery.

[12] Stephen Ibanez, Alex Mallery, Serhat Arslan, Theo Jepsen, Muhammad Shahbaz, Changhoon Kim, and Nick McKeown. The nanopu: A nanosecond network stack for datacenters. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, pages 239–256, 2021.

[13] Anthony Liguori. The Nitro Project – Next Generation AWS Infrastructure. Hot Chips: A Symposium on High Performance Chips, 2019.

[14] Intel and Google Cloud jointly launch data center accelerator chip. https://www.datacenterdynamics.com/en/news/intel-and-google-cloud-jointly-launch-data-center-accelerator-chip.

[15] Xingda Wei, Rongxin Cheng, Yuhan Yang, Rong Chen, and Haibo Chen. Characterizing off-path SmartNIC for accelerating distributed systems. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 987–1004, Boston, MA, July 2023. USENIX Association.

[16] Kartik Srinivasan. The Rise of SmartNICs. https://semiengineering.com/the-rise-of-smartnics/.

[17] Jack Zhao, Miguel Neves, and Israat Haque. On the (dis)Advantages of Programmable NICs for Network Security Services. In *2023 IFIP Networking Conference (IFIP Networking)*, pages 1–9, 2023.

[18] Jack Tigar Humphries, Kostis Kaffes, David Mazières, and Christos Kozyrakis. Mind the gap: A case for informed request scheduling at the nic. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, HotNets '19, page 60–68, New York, NY, USA, 2019. Association for Computing Machinery.

[19] Yiming Qiu, Jiarong Xing, Kuo-Feng Hsu, Qiao Kang, Ming Liu, Srinivas Narayana, and Ang Chen. Automated smartnic offloading insights for network functions. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 772–787, New York, NY, USA, 2021. Association for Computing Machinery.

[20] Zerui Guo, Jiaxin Lin, Yuebin Bai, Daehyeok Kim, Michael Swift, Aditya Akella, and Ming Liu. Lognic: A high-level performance model for smartnics. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '23, page 916–929, New York, NY, USA, 2023. Association for Computing Machinery.

[21] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. Floem: A programming system for nic-accelerated network applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 663–679, Carlsbad, CA, October 2018. USENIX Association.

[22] Intel partners with Google to deploy 'Mount Evans' ASIC-based IPU. https://venturebeat.com/business/intel-partners-with-google-to-deploy-mount-evans-asic-based-gpu.

[23] No AWS operator access. https://docs.aws.amazon.com/whitepapers/latest/security-design-of-aws-nitro-system/no-aws-operator-access.html.

[24] Gernot Heiser and Kevin Elphinstone. L4 microkernels: The lessons from 20 years of research and deployment. *ACM Trans. Comput. Syst.*, 34(1), April 2016.

[25] Vikram Narayanan, Tianjiao Huang, David Detweiler, Dan Appel, Zhaofeng Li, Gerd Zellweger, and Anton Burtsev. RedLeaf: Isolation and communication in a safe operating system. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 21–39. USENIX Association, November 2020.

[26] Kevin Elphinstone and Gernot Heiser. From l3 to sel4 what have we learnt in 20 years of l4 microkernels? In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 133–150, New York, NY, USA, 2013. Association for Computing Machinery.

[27] Haibo Chen, Xie Miao, Ning Jia, Nan Wang, Yu Li, Nian Liu, Yutao Liu, Fei Wang, Qiang Huang, Kun Li, Hongyang Yang, Hui Wang, Jie Yin, Yu Peng, and Fengwei Xu. Microkernel goes general: Performance and compatibility in the HongMeng production microkernel. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 465–485, Santa Clara, CA, July 2024. USENIX Association.

[28] Gernot Heiser and Ben Leslie. The okl4 microvisor: convergence point of microkernels and hypervisors. In *Proceedings of the First ACM Asia-Pacific Workshop on Workshop on Systems*, APSys '10, page 19–24, New York, NY, USA, 2010. Association for Computing Machinery.

[29] Dan Hildebrand. An architectural overview of qnx. In *Proceedings of the Workshop on Micro-Kernels and Other Kernel Architectures*, page 113–126, USA, 1992. USENIX Association.

[30] Kevin Boos, Namitha Liyanage, Ramla Ijaz, and Lin Zhong. Theseus: an experiment in operating system structure and state management. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1–19. USENIX Association, November 2020.

[31] Dong Du, Zhichao Hua, Yubin Xia, Binyu Zang, and Haibo Chen. Xpc: architectural support for secure and efficient cross process call. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, page 671–684, New York, NY, USA, 2019. Association

for Computing Machinery.
[32] Zircon. https://fuchsia.dev/fuchsia-src/concepts/kernel.
[33] Jinyu Gu, Xinyue Wu, Wentai Li, Nian Liu, Zeyu Mi, Yubin Xia, and Haibo Chen. Harmonizing performance and isolation in microkernels with efficient intra-kernel isolation and communication. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 401–417. USENIX Association, July 2020.
[34] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. Minix 3: a highly reliable, self-repairing operating system. *SIGOPS Oper. Syst. Rev.*, 40(3):80–89, July 2006.
[35] Robert Kaiser and Stephan Wagner. Evolution of the PikeOS Microkernel. 02 2007.
[36] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 207–220, New York, NY, USA, 2009. Association for Computing Machinery.
[37] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. Policy/mechanism separation in hydra. *SIGOPS Oper. Syst. Rev.*, 9(5):132–140, November 1975.
[38] Jochen Liedtke. Improving ipc by kernel design. *SIGOPS Oper. Syst. Rev.*, 27(5):175–188, December 1993.
[39] J. Liedtke. On micro-kernel construction. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, page 237–250, New York, NY, USA, 1995. Association for Computing Machinery.
[40] Jing Liu, Anthony Rebello, Yifan Dai, Chenhao Ye, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Scale and performance in a filesystem semi-microkernel. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 819–835, New York, NY, USA, 2021. Association for Computing Machinery.
[41] David L. Black, David B. Golub, Daniel P. Julin, Richard F. Rashid, Richard P. Draves, Randall W. Dean, Alessandro Forin, Joseph Barrera, Hideyuki Tokuda, Gerald Malan, and David Bohman. Microkernel Operating System Architecture and Mach. 06 1991.
[42] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: a new os architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 29–44, New York, NY, USA, 2009. Association for Computing Machinery.
[43] Gerd Zellweger, Simon Gerber, Kornilios Kourtis, and Timothy Roscoe. Decoupling cores, kernels, and operating systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 17–31, Broomfield, CO, October 2014. USENIX Association.
[44] Simon Peter, Adrian Schüpbach, Dominik Menzi, and Timothy Roscoe. Early experience with the barrelfish OS and the single-chip cloud computer. In Diana Göhringer, Michael Hübner, and Jürgen Becker, editors, *3rd Many-core Applications Research Community (MARC) Symposium. Proceedings of the 3rd MARC Symposium, Ettlingen, Germany, July 5-6, 2011*, pages 35–39. KIT Scientific Publishing, Karlsruhe, 2011.
[45] Andrew Baumann, Chris Hawblitzel, Kornilios Kourtis, Tim Harris, and Timothy Roscoe. Cosh: Clear OS data sharing in an incoherent world. In *2014 Conference on Timely Results in Operating Systems (TRIOS 14)*, Broomfield, CO, October 2014. USENIX Association.
[46] Barrelfish: Exploring a Multicore OS. https://www.microsoft.com/en-us/research/blog/barrelfish-exploring-multicore-os/.
[47] Dominik Menzi. Support for heterogeneous cores for barrelfish. Master's thesis, ETH Zurich, 2011.
[48] Barrelfish technical note 001. Technical report, ETH Zurich, 2013.
[49] Barrelfish technical note 005. Technical report, ETH Zurich, 2013.
[50] Ankit Bhardwaj, Chinmay Kulkarni, Reto Achermann, Irina Calciu, Sanidhya Kashyap, Ryan Stutsman, Amy Tai, and Gerd Zellweger. Nros: Effective replication and sharing in an operating system. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 295–312. USENIX Association, July 2021.
[51] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, page 251–266, New York, NY, USA, 1995. Association for Computing Machinery.
[52] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 49–65, Broomfield, CO, October 2014. USENIX Association.
[53] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for μsecond-scale tail latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 345–360, Boston, MA, February 2019. USENIX Association.
[54] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 361–378, Boston, MA, February 2019. USENIX Association.
[55] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. Reflex: Remote flash ≈ local flash. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, page 345–359, New York, NY, USA, 2017. Association for Computing Machinery.
[56] George Prekas, Marios Kogias, and Edouard Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 325–341, New York, NY, USA, 2017. Association for Computing Machinery.
[57] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 281–297. USENIX Association, November 2020.
[58] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Mike Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Mike Ryan, Erik Rubow, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: a microkernel approach to host networking. In *In ACM SIGOPS 27th Symposium on Operating Systems Principles*, New York, NY, USA, 2019.
[59] Max Demoulin, Josh Fried, Isaac Pedisich, Marios Kogias, Boon Thau Loo, Linh Thi Xuan Phan, and Irene Zhang. When idling is ideal: Optimizing tail-latency for highly-dispersed datacenter workloads with perséphone. In *SOSP 2021*, October 2021.
[60] 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, and Anirudh Badam. The demikernel datapath os architecture for microsecond-scale datacenter systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 195–211, New York, NY, USA, 2021. Association for Computing Machinery.
[61] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe user-level access to privileged CPU features. In *10th USENIX Symposium on Operating*

Systems Design and Implementation (OSDI 12), pages 335–348, Hollywood, CA, October 2012. USENIX Association.

[62] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Jack Turner, and Christos Kozyrakis. ghost: Fast and flexible user-space delegation of linux scheduling. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles CD-ROM*, page 588–604, New York, NY, USA, 2021.

[63] Kostis Kaffes, Jack Tigar Humphries, David Mazières, and Christos Kozyrakis. Syrup: User-defined scheduling across the stack. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 605–620, Virtual Event, Germany, 2021. Association for Computing Machinery.

[64] Akshay Narayan, Frank Cangialosi, Prateesh Goyal, Srinivas Narayana, Mohammad Alizadeh, and Hari Balakrishnan. The Case for Moving Congestion Control Out of the Datapath. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, HotNets '17, page 101–107, Palo Alto, CA, USA, 2017. Association for Computing Machinery.

[65] userfaultfd. https://man7.org/linux/man-pages/man2/userfaultfd.2.html. Last accessed: 2023-08-03.

[66] libfuse. https://github.com/libfuse/libfuse. Last accessed: 2020-08-15.

[67] Vasily Tarasov, Abhishek Gupta, Kumar Sourav, Sagar Trehan, and Erez Zadok. Terra incognita: On the practicality of User-Space file systems. In *7th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 15)*, Santa Clara, CA, July 2015. USENIX Association.

[68] The Userspace I/O HOWTO. https://www.kernel.org/doc/html/v4.14/driver-api/uio-howto.html. Last accessed: 2020-11-10.

[69] DriverKit. https://developer.apple.com/documentation/driverkit. Last accessed: 2023-08-04.

[70] Overview of UMDF. https://learn.microsoft.com/en-us/windows-hardware/drivers/wdf/overview-of-the-umdf. Last accessed: 2023-08-04.

[71] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Vancouver, BC Canada, March 2023.

[72] Roni Haecki, Radhika Niranjan Mysore, Lalith Suresh, Gerd Zellweger, Bo Gan, Timothy Merrifield, Sujata Banerjee, and Timothy Roscoe. How to diagnose nanosecond network latencies in rich end-host stacks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 861–877, Renton, WA, April 2022. USENIX Association.

[73] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading distributed applications onto smartnics using ipipe. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 318–333, New York, NY, USA, 2019. Association for Computing Machinery.

[74] RocksDB. https://rocksdb.org. Last accessed: 2020-11-27.

[75] mm: userfaultfd: add new UFFDIO_SIGBUS ioctl. https://lwn.net/ml/linux-kernel/20230511182426.1898675-1-axelrasmussen@google.com. Last accessed: 2023-08-03.

[76] Anil Yelam, Kan Wu, Zhiyuan Guo, Suli Yang, Rajath Shashidhara, Wei Xu, Stanko Novaković, Alex C. Snoeren, and Kimberly Keeton. PageFlex: Flexible and Efficient User-space Delegation of Linux Paging Policies with eBPF. In *USENIX Annual Technical Conference (ATC '25)*. USENIX Association, 2025.

[77] VFIO: Virtual Function I/O. https://www.kernel.org/doc/html/v5.9/driver-api/vfio.html.

[78] big.LITTLE - ARM. https://www.arm.com/why-arm/technologies/big-little. Last accessed: 2020-11-27.

[79] M1 Overview. https://www.apple.com/ua/business/mac/pdf/Apple-at-Work-M1-Overview.pdf. Last accessed: 2023-08-09.

[80] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, page 158–169, New York, NY, USA, 2015. Association for Computing Machinery.

[81] The next wave of Google Cloud infrastructure innovation: New C3 VM and Hyperdisk. https://cloud.google.com/blog/products/compute/introducing-c3-machines-with-googles-custom-intel-ipu. Last accessed: 2024-10-06.

[82] Yawen Wang, Daniel Crankshaw, Neeraja J. Yadwadkar, Daniel Berger, Christos Kozyrakis, and Ricardo Bianchini. Sol: Safe on-node learning in cloud platforms. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 622–634, New York, NY, USA, 2022. Association for Computing Machinery.

[83] The Production Environment at Google, from the Viewpoint of an SRE . https://sre.google/sre-book/production-environment/.

[84] gRPC. https://grpc.io. Last accessed: 2023-08-03.

[85] The Linux Completely Fair Scheduler. https://www.kernel.org/doc/html/latest/scheduler/sched-design-CFS.html. Last accessed: 2023-08-09.

[86] Jonathan Corbet. An EEVDF CPU scheduler for Linux. https://lwn.net/Articles/925371/. Last accessed: 2024-07-01.

[87] Jonathan Corbet. Completing the EEVDF scheduler. https://lwn.net/Articles/969062/. Last accessed: 2024-07-01.

[88] F. J. Corbato. A paging experiment with the multics system. In *In Honor of P. M. Morse*, pages 217–228. MIT Press, 1969. Also as MIT Project MAC Report MAC-M-384, May 1968.

[89] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. AIFM: High-Performance, Application-Integrated far memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 315–332. USENIX Association, November 2020.

[90] Jonathan Corbet. AutoNUMA: the other approach to NUMA scheduling. https://lwn.net/Articles/488709/.

[91] Henrique Fingler, Isha Tarte, Hangchen Yu, Ariel Szekely, Bodun Hu, Aditya Akella, and Christopher J. Rossbach. Towards a machine learning-assisted kernel with lake. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 846–861, New York, NY, USA, 2023. Association for Computing Machinery.

[92] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, HotNets '16, page 50–56, New York, NY, USA, 2016. Association for Computing Machinery.

[93] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 270–288, New York, NY, USA, 2019. Association for Computing Machinery.

[94] Qijing Huang, Ameer Haj-Ali, William Moses, John Xiang, Ion Stoica, Krste Asanovic, and John Wawrzynek. AutoPhase: Compiler Phase-Ordering for HLS with Deep Reinforcement Learning. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 308–308, 2019.

[95] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. Neural adaptive video streaming with pensieve. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM

'17, page 197–210, New York, NY, USA, 2017. Association for Computing Machinery.

[96] William R. Thompson. ON THE LIKELIHOOD THAT ONE UNKNOWN PROBABILITY EXCEEDS ANOTHER IN VIEW OF THE EVIDENCE OF TWO SAMPLES. *Biometrika*, 25:285–294, 1933.

[97] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. RDMA over Commodity Ethernet at Scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 202–215, Florianopolis, Brazil, 2016. Association for Computing Machinery.

[98] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. Revisiting network support for rdma. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 313–326, New York, NY, USA, 2018. Association for Computing Machinery.

[99] Microsoft Corp. Receive Side Scaling. http://msdn.microsoft.com/library/windows/hardware/ff556942.aspx, 2018.

[100] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. Understanding pcie performance for end host networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 327–341, New York, NY, USA, 2018. Association for Computing Machinery.

[101] Data plane development kit. http://www.dpdk.org/. Last accessed: 2019-06-26.

[102] Anastasiia Ruzhanskaia, Pengcheng Xu, David Cock, and Timothy Roscoe. Rethinking Programmed I/O for Fast Devices, Cheap Cores, and Coherent Interconnects, 2024.

[103] Ming Liu, Simon Peter, Arvind Krishnamurthy, and Phitchaya Mangpo Phothilimthana. E3: Energy-Efficient microservices on SmartNIC-Accelerated servers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 363–378, Renton, WA, July 2019. USENIX Association.

[104] Bojie Li, Kun Tan, Layong (Larry) Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. Clicknp: Highly flexible and high performance network processing with reconfigurable hardware. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 1–14, New York, NY, USA, 2016. Association for Computing Machinery.

[105] Felix Xiaozhu Lin and Xu Liu. memif: Towards programming heterogeneous memory asynchronously. *SIGARCH Comput. Archit. News*, 44(2):369–383, March 2016.

[106] HOME | Compute Express Link. https://www.computeexpresslink.org.

[107] Intel(R) Xeon(R) Processor Scalable Family Technical Overview. https://www.intel.com/content/www/us/en/developer/articles/technical/xeon-processor-scalable-family-technical-overview.html. Last accessed: 2022-12-07.

[108] NVLink and NVSwitch. https://www.nvidia.com/en-us/data-center/nvlink. Last accessed: 2022-12-07.

[109] Andrea Pellegrini, Nigel Stephens, Magnus Bruce, Yasuo Ishii, Joseph Pusdesris, Abhishek Raja, Chris Abernathy, Jinson Koppanalil, Tushar Ringe, Ashok Tummala, Jamshed Jalal, Mark Werkheiser, and Anitha Kona. The Arm Neoverse N1 Platform: Building Blocks for the Next-Gen Cloud-to-Edge Infrastructure SoC. *IEEE Micro*, 40(2):53–62, 2020.

[110] Compute Engine | Google Cloud. https://cloud.google.com/products/compute. Last accessed: 2024-08-12.

[111] Manohar Vanga, Arpan Gujarati, and Björn B. Brandenburg. Tableau: A high-throughput and predictable vm scheduler for high-density workloads. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.