Enabling Fast Networking in the Public Cloud

Alireza Sanaee*†
University of Cambridge and Huawei
Cambridge, United Kingdom
ss3230@cam.ac.uk

Vahab Jabrayilov*
Columbia University
New York, USA
vjabrayilov@cs.columbia.edu

Ilias Marinos‡
NVIDIA
London, United Kingdom
ilias.marinos@gmail.com

Farbod Shahinfar
Politecnico di Milano
Milan, Italy
farbod.shahinfar@polimi.it

Divyanshu Saxena
The University of Texas at Austin
Austin, USA
dsaxena@cs.utexas.edu

Gianni Antichi
Politecnico di Milano
Milan, Italy
Queen Mary University of London
London, United Kingdom
gianni.antichi@polimi.it

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

Abstract

Despite a decade of research, most high-performance userspace network stacks remain impractical for public cloud tenants developing their applications atop Virtual Machines (VMs). We identify two root causes: (1) reliance on specialized NIC features (e.g., flow steering, deep buffers) absent in commodity cloud vNICs, and (2) rigid execution models ill-suited to diverse application needs. We present Machnet, a high-performance and flexible userspace network stack designed for public cloud VMs. Machnet uses only a minimal set of vNIC features that any major cloud provider supports. It also relies on a microkernel architecture to enable flexible application execution. We evaluate Machnet across three major public clouds and on production-grade applications, including a key-value store, an HTTP server, and a state-machine replication system. We release Machnet at https://github.com/microsoft/machnet.

CCS Concepts: • **Networks** → **Network architectures**; *Cloud computing*; • **Computer systems organization** → *Distributed architectures*; • **Software and its engineering** → *Operating systems.*

*Equal contribution.
†Work done while at Queen Mary University of London.
‡Work done while at Microsoft.

ASPLOS '26, Pittsburgh, PA, USA

ACM ISBN 979-8-4007-2359-9/2026/03
https://doi.org/10.1145/3779212.3790158

Keywords: User-level networking, DPDK, Remote Procedure Call (RPC), Low-latency systems, Network stack optimization, Cloud infrastructure, Virtualization

ACM Reference Format:
Alireza Sanaee, Vahab Jabrayilov, Ilias Marinos, Farbod Shahinfar, Divyanshu Saxena, Gianni Antichi, and Kostis Kaffes. 2026. Enabling Fast Networking in the Public Cloud. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '26), March 22–26, 2026, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3779212.3790158

1 Introduction

Cloud-hosted distributed applications running in data centers (e.g., distributed data stores, stream processing engines, caching systems, backends of productivity software) are hungry for fast inter-server communication [29, 40, 41, 68]. Developers (i.e., cloud tenants) naturally turn to the rich ecosystem of kernel-bypass ("userspace") networking stacks-such as mTCP [58], libvma [6], eRPC [60], TAS [62], Demikernel [84]- to cut inter-machine networking latency. However, they soon discover that existing systems fail to meet their needs in the most common deployment setting: virtual machines (VMs) in the public cloud. *What prevents these stacks from working for cloud tenants?*

First, cloud tenants without bare-metal access can only use virtual network interfaces (vNICs) available in cloud VMs. The problem is that such vNICs lack many features that state-of-the-art high-performance userspace network stacks need. Their restricted feature set stems from the public clouds' need to support (a) network virtualization and (b) a consistent interface across a decade of deployed NIC generations. Notably, we find that no vNIC offered by leading cloud providers supports "flow steering," a crucial feature for mapping network flows to CPU cores that has been a

NIC feature	Description	Approximate year of introduction	Network stacks that used the feature
Flow steering	Redirect matching packets to a specific receive queue	2009 (Intel 82599ES)	eRPC [60], Snap [70], R2P2 [63]
RSS reconfiguration	Query/modify the RSS key or redirection table	2009 (Intel 82599ES)	TAS [62], RSS++ [25], mTCP [58]
TX DMA from app memory	Transmit packets directly from DMA-registered user memory	2009 (Mellanox ConnextX2)	Cornflakes [79], eRPC [60]
Remote DMA	Access remote server memory without involving remote CPU	2009 (Mellanox ConnextX2)	SocksDirect [67], mRPC [33]
Deep RX queues	RX queues with thousands of entries, to prevent host-side packet drops	2009 (Mellanox ConnectX2)	eRPC [60]
Multi-packet RQs	Receive multiple packets with one RX queue descriptor	2014 (Mellanox ConnectX4)	eRPC [60], Virtuoso [80]

Table 1. *Long slack between hardware support and cloud adoption: Examples of NIC features that are not exposed to VMs created in public clouds of major providers. Existing stacks heavily rely on them since some of these features have been available for more than a decade to this date.*

standard in bare-metal interfaces since 2009 [3]. Other missing functionalities in vNICs include configurable support for receive-side scaling (RSS), RDMA, and multi-packet receive queues (Table 1). Although newer NIC models may support all these features, the financial impracticality of replacing older, widely deployed models prevents their universal availability in the near future [4].

Second, we identify that the architectural complexity of userspace stacks is also a limiting factor. These stacks are based on either the library operating system (libOS) [58, 60] or the sidecar [62, 70] model. The libOS approach statically links the stack into the application, complicating policy enforcement (e.g., congestion control [59, 60]) and pushing developers toward low-level systems languages [27, 84]. The sidecar model addresses some of these issues by running the stack as a separate process, allowing greater flexibility. Existing sidecars, however, co-design multiple components such as the CPU scheduler, queue management, and networking stack to squeeze the last drops of performance [48, 70, 75]. Such extremes are often unnecessary in public clouds [83] and make deployment and tuning hard for non-experts, frequently relying on service-time assumptions or application-specific tuning [63, 75, 84].

Assuming the role of a cloud tenant with no ability to modify the vNIC, we design Machnet, a high-performance and easy-to-use userspace network stack that relies only on vNIC features available across all cloud providers. To do this, we first model a *Least Common Denominator (LCD)* vNIC, a minimal, cross-cloud feature set that any kernel-bypass vNIC provides. This LCD model is the practical "network device ISA" for the public cloud. We show that it consists only of (a) plain Ethernet packet I/O; (b) opaque RSS (no key inspection or modification); (c) at most one TX/RX queue per CPU core; (d) and at most 256 descriptors per queue. The academic community has largely overlooked these constraints: even recent systems targeting cloud settings (e.g., Junction [47]) depend on non-LCD features and direct NIC access, placing them out of reach for typical cloud tenants.

The key challenge is delivering high performance and ease of use while staying within this minimal feature set. For example, to scale Machnet over multiple CPU cores, we design a new flow-to-core mapping technique called RSS−− that requires only the LCD NIC's opaque RSS functionality.

To preserve application flexibility, Machnet uses a simplified sidecar that mediates NIC access for multiple processes and threads, supports interrupt-based execution, exposes bindings for high-level languages, and is easily deployed as a Docker container. We show that, despite the extra hop, Machnet achieves latency almost as low as existing userspace stacks that rely on advanced hardware features unavailable to cloud tenants. While engineered to deliver high performance using only LCD features, Machnet can opportunistically leverage advanced NIC capabilities when exposed without changing the programming model or sacrificing portability.

We demonstrate Machnet in the wild, running it on the three major public clouds and evaluating its performance on three production-level applications: a key-value store, an HTTP server, and state-machine replication. Machnet is open-sourced and available at https://github.com/microsoft/machnet.

2 Related Work and Motivation

We next discuss the main factor that has discouraged developers from using existing fast userspace stacks in public cloud VMs: incompatibility with cloud vNICs (§2.1) and the needs of cloud applications (§2.2). We then detail the features an ideal portable stack should provide along with characteristics we do not target (§2.3).

2.1 Incompatibility with cloud vNICs

Cloud vNICs are an abstraction layer above bare-metal NICs and their virtual functions (VFs). This is the fundamental reason why userspace stacks designed for bare-metal NICs and VFs might not work on vNICs.

An important factor in public clouds is that the fleet's network hardware spans multiple generations of hardware from multiple vendors, e.g., Microsoft's Azure cloud uses NICs spanning 12 years, from ConnectX-3 devices released in 2011 [12] to Microsoft's MANA devices released in 2023 [17, 18]. To provide a consistent virtualized network with manageable complexity, the vNICs exposed to tenants typically provide a uniform feature set [15, 19, 39]. Note that exceptions to this exist for specialized VM types targeting particular use cases (e.g., vNICs in VMs for High-Performance Computing support RDMA).

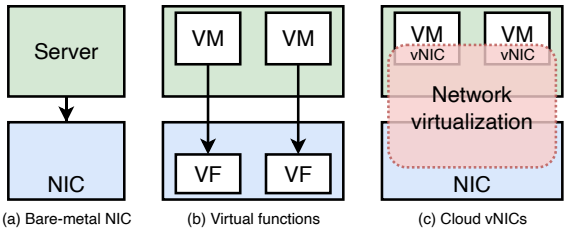


Figure 1. *A logical comparison of bare-metal NICs, virtual functions, and cloud vNICs.*

Figure 1 shows a qualitative comparison of three types of bare-metal NIC, virtual functions (VFs), and vNICs. Bare-metal NICs expose the NIC's entire suite of features to the user, ranging from low-level knobs like RSS and flow steering to entire protocol layers like RDMA and Transport Layer Security (TLS) [76]. VFs are isolated NIC slices that share the physical NIC's hardware resources [20], which typically expose most but not all the physical NIC's features [15, 19, 39]. Simple virtualization environments (e.g., on-premise data centers) typically dedicate a VF to each VM; the VM can access this VF from the guest's userspace if needed.

In contrast, guest VMs in public clouds do not have raw access to the virtual function since that would bypass the cloud's network virtualization layer. The abstraction layer provides network virtualization [36, 46, 65], which includes (1) tenant features like bring-your-own-IPs, firewalls, routing, and access control lists, and (2) cloud operator features like inter-tenant isolation, transparent network upgrades, servicing, monitoring, and VM migration.

Since Microsoft has published details about their Azure vNIC implementation [45, 46], we use it as a representative example. In their ConnectX-based vNICs, the network virtualization layer spans an FPGA datapath, host hypervisor software, and the guest VM's paravirtual device driver [46]. During network I/O, guest applications must use Azure's paravirtual "netvsc" drivers for both kernel- and kernel-bypass I/O instead of the virtual function's corresponding raw ConnectX drivers. The paravirtual driver includes support for network virtualization events such as servicing (e.g., for the FGPA SmartNIC), monitoring, VM migration, etc. In AWS, their purpose-built Nitro devices implement this layer [11]. Google Cloud Platform (GCP) uses dedicated CPU cores for network virtualization [70].

Existing fast userspace stacks often aim to exploit advanced features available in bare-metal NICs: this is the case for example of flow steering [58, 60, 84], multi-packet receive queues [60, 80], or deep receive queues [60], to name a few. One may expect those stacks to work as-is on vNICs with minor modifications. Unfortunately, we find that this is not the case, and significant design and implementation changes are required. The problem is that while the features they rely upon have been ubiquitous in bare-metal NICs for over a decade (e.g., Intel's 82599 Ethernet controllers released in 2009 support flow steering [3]), they are not yet part of the virtualized NICs exposed to guest VMs.

In Table 1, we list the features used in existing projects that are not supported by a large span of NICs deployed currently in the cloud.

Consequence. A userspace network stack for the cloud needs to leverage *only* the features available at cloud vNICs. Those depend on the network virtualization layer, which exposes only a small subset of the features available on bare-metal NICs.

2.2 Incompatibility with cloud applications

Cloud applications display a range of characteristics that match the dynamic and distributed nature of modern computing environments [35]. These applications are typically written in various programming languages, including high-level ones such as JavaScript, C#, Go, and Python [49, 57]. They are logically composed of diverse components, including databases [41], key-value stores [5, 72], web servers [30], and authentication services [55], each serving distinct functions. Multiple components are often co-located in the same VM to minimize the communication costs [38]. Furthermore, administrators need to leverage oversubscription, effectively running multiple processes or threads per core to maximize resource utilization [35]. As we more extensively discuss in §3.2, the problem is that most of the userspace networking stacks (full system solutions such as Shinjuku, or Demikernel [59, 84]) follow the libOS model [58, 60, 79], effectively compiling the networking stack together with the application, which in turn takes full ownership of the NIC. This model is incompatible with the traits of cloud applications.

Consequence. A userspace network stack for the cloud should not be integrated with applications to ensure support for multiple programming languages and the ability to serve multiple applications at the same time.

2.3 Requirements from a cloud-native userspace stack.

In the following, we summarize the characteristics an ideal userspace stack designed for the cloud should provide:

- **R1. NIC agnostic.** The stack should not depend on any NIC features other than those available in the vast majority of cloud vNICs (§3.1).
- **R2: Support for diverse applications.** The stack should support many processes at the same time since rarely a VM hosts just a single application (§3.2).
- **R3: Low-latency.** The stack should not compromise on performance: it should be comparable with existing approaches that rely on specific NIC features (§3.3).

Non-goal 1: Highly communication-intensive workloads. This paper does not target massive-scale communication-intensive applications like storage and machine learning, already served by RDMA and other fabrics [13, 16, 43]. Instead, we target the long tail of cloud applications like distributed databases, caches, and stream-processing engines. Based on our discussion with developers, we found that the key metric of interest in these applications is *latency*. For example, while existing userspace stacks projects target tens of millions of requests per second [40, 60, 68, 84] (at the cost of restricted application execution models), production workloads are often far less demanding in terms of message rate: an analysis of Twitter's production cache workload shows that most servers run under 100k requests per second [83]. Google's Snap paper mentions a server handling a few million requests per second at peak load [70].

Implication. We are willing to sacrifice message rate and bandwidth in favor of compatibility with a wide range of vNICs and application execution models as long as latency remains competitive [26, 37, 41].

Non-goal 2: Compatibility with unmodified applications. In our discussions with engineering teams, we found that few expect to run unmodified applications on a userspace stack. Instead, developers are often willing to rewrite the networking parts of their application and, in some cases, even co-design the application with the stack. This is intuitive for two reasons. First, reducing latency often necessitates changes to the application's threading model, e.g., to avoid context switches, which go hand-in-hand with changes to the networking model. Second, most applications do not directly use low-level networking APIs (e.g., POSIX) but project-specific middleware (e.g., gRPC or a handwritten messaging layer), limiting the rewriting effort to a small part of the code.

Implication. We do not target binary compatibility with the BSD sockets API. An API that resembles it is sufficient.

Non-goal 3: Build a new OS.. A large body of research has focused on building full operating systems to support microsecond-scale applications in data centers such as Junction [47], ZygOS [78], IX [27], Caladan [48], Shenango [75], and Shinjuku [59]. These systems have seen little adoption in practice as most developers are hesitant to migrate to a new OS; changing the networking stack is a much easier ask. Furthermore, they are mostly incompatible with both virtualized CPU environments and cloud vNICs.

Implication. Our goal is to build a fast networking stack, not a new OS.

3 Design Principles

In this section, we discuss the principles that drove our design. Those allow us to meet the requirements we set for a new cloud-native userspace stack (§2.3).

I/O interface	Ethernet frames
#RX/TX queues	Number of CPU cores
TX/RX queue size	256 descriptors
List of offloads	Opaque receive side scaling

Table 2. *Least common denominator NIC model in Machnet.*

3.1 LCD model for cloud NICs

We must understand how to model a cloud NIC to meet the first requirement. We define the "Least Common Denominator" (LCD) model, the common set of features available in all modern kernel-bypass capable NICs, including the vNICs of the three major cloud providers (Amazon EC2, Microsoft Azure, and Google Cloud Platform).

Our insight is that for broad adoption of a stack, it must run seamlessly across bare-metal NICs, VFs, and vNICs; the LCD model captures the common features that stack designers can rely on. The LCD NIC model (Table 2) provides OS-bypass Ethernet packet I/O over a number of TX and RX queues up to the number of CPU cores, with queues containing up to 256 descriptors each. Receive-side scaling is supported, but this NIC does not expose the RSS key or the RSS indirection table to the guest VM for inspection or modification.

Note that individual vNICs may support features beyond the LCD model and may be worth individually optimizing. For example, most vNICs in Azure VMs support registering application memory with the NIC for zero-copy transmission (see below). However, vNICs in other clouds (e.g., Google Cloud Platform, Amazon EC2) do not support this feature, so we currently do not include it in the LCD model.

Queues and descriptors. Since a cloud provider must virtualize the NIC's queues and SRAM among tenants, each tenant gets a limited piece of these resources. Across all three major cloud providers, we observed that the number of TX/RX queue pairs equals the number of server vCPUs rented by the tenant, and the number of descriptors per queue is 256. This precludes designs like eRPC [60] that create very large receive queues to avoid host-side packet drops. The LCD NIC requires posting and polling one descriptor per packet; descriptor coalescing optimizations such as batching multiple packets per descriptor [60] are unavailable.

Implication. The new userspace stack should work as a separate process, multiplexing the available NIC queues among multiple threads.

Flow-to-CPU core mapping. The LCD NIC supports only opaque receive side scaling (RSS), i.e., the NIC randomly hashes flows to RX queues, but the stack may not query or modify the NIC's RSS key or indirection table. In contrast, granular control over bare-metal NICs' flow-to-core mapping units has been widely used to scale network stack throughput over multiple cores and to implement rich CPU scheduling policies. For example, Snap [70] and eRPC [60] install NIC

flow rules that match on packet headers and redirect matched packets to a specific RX queue, which are then processed by a particular CPU core. TAS [62] and RSS++ [25] reconfigure the NIC's RSS indirection table to dynamically scale the number of CPU cores. mTCP [58] installs a special symmetric RSS key on the NIC [62]. IX [27] queries the NIC's RSS key to pick UDP source ports for flows that maximize CPU affinity.

Implication. The new userspace stack should use only opaque RSS while providing isolation between applications (no performance inference is created).

Direct memory access. During packet I/O with the LCD NIC, the stack copies packets between application memory and the NIC's TX/RX ring buffers. DMA-related optimizations are not available. For example, in Cornflakes [79], the NIC reads DMA-registered application memory to serialize objects into Protocol Buffers. The benefits of RDMA have been widely studied in numerous systems [33, 40, 67], but support for RDMA in guest VMs (e.g., for performance isolation [64]) is still in progress.

Implication. The new userspace stack should be built atop plain packet I/O instead of relying on zero-copy packet transmission or RDMA.

3.2 Diverse application execution models

Userspace stacks have historically sacrificed application execution flexibility for performance. For example, the libOS-based approach—used by Sandstorm, libvma, Seastar, OpenOnload, eRPC, and Demikernel [2, 6, 60, 69, 77, 84] —has been the standard, with a few notable exceptions [62, 70]. The libOS approach imposes several constraints on the application's execution model, the most important of which is that one application "owns" the NIC, so only one process can use the NIC at a time. Additionally, each application thread typically owns one NIC queue, so the small number of NIC queues on cloud vNICs limits the number of threads. Note that simply attaching many vNICs to the cloud VM is not a feasible solution to these problems since it goes against the standard VM deployment model. In Microsoft Azure, among general-purpose, computer-optimized, memory-optimized, and storage-optimized VMs, the maximum vNICs allowed for VMs is 8 [8]. Further, Oracle Cloud limits the number of vNICs to the number of vCPUs in a virtual machine [7], and VMware only allows up to 10 vNICs regardless [10].

For Internet service applications such as databases, caches, and stream-processing engines, the libOS model is too restrictive. One server often runs multiple applications written in various languages, with possibly numerous threads per process. For example, Google production workloads are incompatible with this model since they run hundreds of threads per core and use high-level languages such as Go [1].

No threading constraints. There is a need to support an arbitrary number of threads to accommodate modern multi-threaded cloud applications. This problem is challenging in libOS approaches that require applications to use a single-threaded event loop. Particularly, the number of threads would be limited to either the number of NIC queues or the number of CPU cores available to the virtual machine.

Implication. The new userspace stack should adopt a sidecar (microkernel) approach. Here, the sidecar owns the NIC. With this layer of indirection, multiple processes and threads can use the same NIC. The sidecar can send interrupts to applications, permitting non-poll mode processing.

Blocking receive. Userspace stacks typically require applications to continuously poll for incoming messages. Although this approach gets the best latency, it restricts application execution flexibility by limiting the number of threads to the number of CPU cores. The problem is that, in production workloads, many threads often share just one core [1]. For example, one workload we learned about required low latency when hosting thousands of database instances, each isolated in a separate process on a single VM; most of these expected little traffic. Each database needs low-latency messaging to replicate data to remote servers when active. This execution model is ill-suited to polling, as the number of database processes exceeds the number of CPU cores.

Implication. The new userspace stack should allow users to block on receive calls, i.e., application threads can sleep and be woken up by the sidecar when a message arrives.

3.3 Low-latency support

Linux, the most common OS in the cloud [46], is notoriously slow, and existing research attempted to improve its latency with disruptive changes [31]. While with this paper, we attempt to tackle this problem, designing the new stack following a sidecar model (as noted before) might raise performance concerns about the overhead imposed by the inter-process shared-memory communication (IPC) when data has to be delivered to the application. It is worth noting that the IPC is small, considering our target deployments in large cloud networks. Indeed, as we show later, an extra 1 µs is significant in small bare-metal testbeds, e.g., it is 43% of eRPC's 2.3 µs median latency, but cloud networks are large, with VMs often separated by multiple switches and long fiber-optic cables [52]. The extra 1 µs is relatively small, e.g., only 10% higher than eRPC in our measurement (§5).

4 Machnet

Machnet has two components: a "sidecar" userspace process that handles all kernel-bypass network I/O and a shim library that applications use to communicate with the sidecar (Figure 2). The shim library provides socket-like API calls for applications to send and receive messages and communicate with the sidecar over shared-memory channels, summarized in Table 3. The Machnet sidecar process handles all kernel-bypass network I/O and can scale to multiple cores without relying on flow steering.

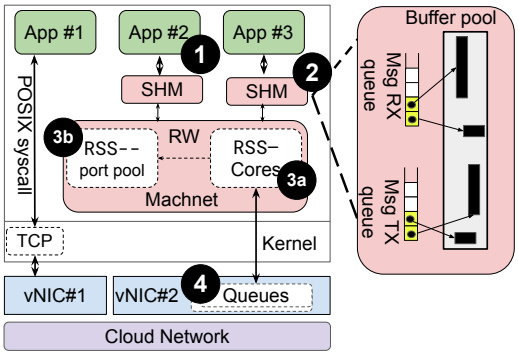


Figure 2. *Machnet architecture. Legacy application (App 1) can use traditional POSIX syscalls to communicate over Linux Kernel TCP, while multiple applications (App 2 and App 3) connect to the network over CNet using the shim library.*

Applications establish a shared-memory channel with the sidecar process ❶, facilitating message exchange with the Machnet sidecar. From a security perspective, these shared memory regions are entirely isolated, and a compromised application cannot manipulate the shared memory region of other applications. Each shared memory channel (SHM) ❷ comprises a pair of Message RX/TX queues, managing references to the head of incoming and outgoing buffers. Machnet employs RSS-- powered "engines" ❸a per CPU core to handle the processing of ingress/egress packets from associated RX/TX queues ❹. During connection setup, Machnet utilizes the RSS-- port pool ❸b to associate the flow with the correct engine. This association process occurs off the hot path (see §4.2.1 for further details).

4.1 Building a fast and easy-to-use networking sidecar

The Machnet sidecar is implemented in C++ and uses DPDK for kernel bypass packet I/O. We use C for the shim library to make the interface with other languages easier. The stack and shim library are implemented in ~11K lines of code, excluding language bindings.

Why DPDK instead of XDP?. The lack of flow steering in vNICs cripples AF_XDP in cloud VMs. AF_XDP is a recent Linux improvement that bypasses most kernel stack processing to reduce latency, which may be "good enough" for some applications.[1] In its best "zero-copy" case, the device driver delivers a packet directly to user-space memory, bypassing the networking code. We studied AF_XDP in detail on cloud VMs and implemented a Machnet variant based on DPDK's efficient AF_XDP driver, and found deficiencies, in line with prior reports [61].

[1]Another advance—io_uring—focuses on improving throughput by batching syscalls, which is irrelevant to our latency target.

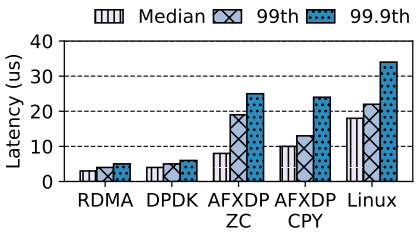


Figure 3. *This experiment measures the single message latency of each approach. DPDK performs notably better than existing options.*

Surprisingly, we found that AF_XDP does not use zero-copy in cloud VMs. The reason is interesting: *zero-copy AF_XDP requires flow-steering support in the NIC as a security measure*, to prevent packets destined for one application from being delivered to another [82]. Without zero-copy support, latency is not significantly better than that of the standard kernel network stack, and is up to far worse than with zero-copy. Second, to support multiple application processes, AF_XDP also requires a sidecar model (so it cannot avoid the IPC overhead) since the number of NIC queues limits the number of AF_XDP sockets.

Zero-copy is the primary challenge when using AF_XDP, but it is not the only factor contributing to performance issues. When an application uses AF_XDP, the kernel is responsible for putting messages in the correct memory regions when received from the network and transmitting application messages placed in the shared memory region by the application. For the application to experience minimal latency, kernel threads must work hand in hand and in sync with the application threads to avoid extra latency. Unfortunately, this does not happen in practice, especially in VMs, where threads are not guaranteed to get the actual physical CPU from the hypervisor, and slight hiccups will cause high tail latency.

We ran experiments to understand the performance of AF_XDP, using bare-metal nodes to allow experiments with zero-copy (xl170 Cloudlab). We pin the interrupt (IRQ) and application threads on different CPU cores for best results. The first experiment measures round-trip latency with a single message in flight using two nodes, comparing RDMA, DPDK, AF_XDP zero-copy (ZC), AF_XDP copy, and standard Linux. Figure 3 shows that DPDK achieves 4.1x and 5.6x lower latency at the 99.9th percentile compared to AF_XDP ZC and Linux.

The second experiment measures the load latency of each approach via three nodes. A client on one node only measures the latency by sending single messages, and another workload generator residing on another node produces the background workload of 64-byte messages. A separate server node echoes messages back to the clients. We vary the background load and measure the latency. Figure 4 shows each

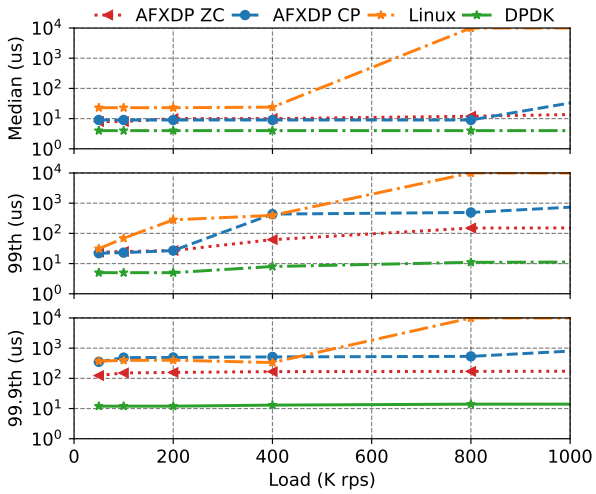


Figure 4. *DPDK performs better than AF_XDP zero-copy. Unfortunately, even the best non-DPDK option (AF_XDP ZC) is not available in cloud virtual machines due to the lack of flow steering support.*

approach's median, 99th, and 99.9th percentile latency. DPDK dominates AF_XDP by 10x at the 99.9th percentile.

Network protocol. Cloud vNICs favor connection-oriented protocols over connectionless protocols. When a guest VM initiates a new connection, an SDN policy evaluation pipeline processes the control packets (e.g., SYN) [45]. While this is transparent to the VM, it significantly increases first-packet latency and limits the number of connections per second that can be initiated [23, 36]. It is common for packets-per-second to drop by 100× or more when initiating new connections [24, 45]. Moreover, connectionless transports in combination with packet spraying via random UDP ports (e.g., Homa [71], NDP [53]) may run into SDN-related bottlenecks. For instance, Homa uses short-lived connections to enable its spraying mechanism. Unfortunately, every new connection created should go through an SDN slow path to fill up SDN flow tables, resulting in degraded performance down to 10K packets per second. [24].

To reduce the SDN overheads of the public cloud, we design Machnet to feature a connection-oriented protocol similar to TCP. Machnet provides reliable in-order message delivery, with fragmentation and reassembly and selective acknowledgments for retransmissions. We use UDP packets with an added Machnet header, implementing a flow-based message-oriented protocol that preserves message boundaries (e.g., unlike TCP's byte stream). We support message sizes up to 8 MB and allow multiple outstanding messages per flow. Overall, other transport protocols, as well as TCP, can be implemented in Machnet.

Packaging. We package Machnet as a Docker container so that users need not deal with DPDK, which is often difficult to install and run. DPDK is a large project with around two million lines of code, an unstable API, and dynamically-probed drivers, making its complexity comparable to a small operating system. As an example, a common usability problem occurs as follows: DPDK does not build the driver for ConnectX NICs if it does not find the NIC's userspace libraries; when this happens, applications build fine but fail with a cryptic error message at runtime when DPDK fails to locate a driver for the NIC. To further complicate matters, incompatible versions of these libraries are available from three different sources (upstream rdma-core, the distribution's rdma-core, and Mellanox OFED), causing confusion.

While these may seem minor issues, we believe this complexity has been a serious roadblock for adopting library OS approaches, which require application developers without DPDK backgrounds to go through a steep learning curve. Our Machnet container image includes a pre-built sidecar binary along with all required DPDK patches(e.g., Google Cloud Platform patch[51]), which can be downloaded and run with a small shell script. Application developers using Machnet do not interface with DPDK but instead use the sockets-like shim library API.

4.2 Scaling to multiple cores

The Machnet sidecar process can use multiple CPU cores to (1) scale performance to large VMs (e.g., with 100 Gbps networking) and (2) provide performance isolation among applications. We run one Machnet "engine" thread per core; each engine has an exclusive NIC RX/TX queue pair and runs in a busy-polling run-to-completion loop. We use a shared-nothing architecture for scalability (i.e., engines do not share flow state or shared memory channels). One dedicated shared memory channel exists between each application thread and a Machnet engine. Machnet's shim library API also allows each application thread to specify the engine to use (the default policy is round-robin). This is useful when binding a latency-critical application to a dedicated engine, while less latency-sensitive applications can share an engine.

Need for flow-to-core mapping. NIC-offloaded flow-to-core mapping is necessary to support (1) our shared-nothing multi-core architecture and (2) support for binding applications to specific engines. Without this, a packet may land at any engine core. This, in turn, requires shuffling packets or flows between engine cores, breaking the shared-nothing architecture and inter-application isolation. For example, an ACK packet may land on a different engine core than the one that sent the original data packet, requiring the receiver to either shuffle the packet to the correct core or pull the flow state from that core.

Snap [70] uses the NIC's flow steering hardware to map flows to engine cores. TAS [62] uses only RSS, but it cannot

assign flows to specific engine cores, which is necessary for performance isolation between applications or flows. In addition, TAS requires each engine to communicate with all application threads, which does not scale well to the large number of threads we target. Regardless, the problem is that cloud VMs do not expose flow steering to guest VMs at the time of writing.

We found that all major cloud providers support RSS, a stateless NIC mechanism that hashes packet headers (e.g., a Toeplitz hash of the four-tuple) to a receive queue. [2] RSS guarantees flow affinity to different cores, i.e., packets of the same flow will all hit the same receive queue. But it is challenging to know *a priori* which queue a packet will land on, as discussed next.

***Difficulty of inverting RSS.*.** Can we use RSS for scaling Machnet with its shared-nothing architecture? We initially implemented a novel mechanism that requires knowing the RSS key since we initially believed retrieving the RSS key would be part of the LCD NIC model (e.g., DPDK provides an API to retrieve the RSS key). While this approach worked on our initial set of test cloud VMs, we found that it is unreliable in practice. Some cloud NICs refuse to expose the RSS key. Others vary in the endianness of the computed RSS hash, introducing additional complexity by requiring hardware-specific code and testing paths.

Our mechanism worked as follows. Below, we use a client-server connection as an example, where the server is listening on a particular UDP port. During connection setup by the client, the server (i.e., the server-side Machnet process) communicates its RSS key and number of RX queues to the client. The client uses this and its RSS key to compute a UDP source port to hash all flow packets to the desired local and remote RX queues. IX uses a local variant of this technique, where the client chooses a UDP source port that causes server-to-client packets to hash to the core that sent the original client-to-server packet [27]. However, the local variant alone cannot support the application-to-engine affinity we require since it cannot control which server-side RX queue the packet lands on.

4.2.1 `RSS--`.

We created a new approach called `RSS--` that uses randomization to work despite the opaqueness of RSS in cloud NICs. Machnet repeatedly tries to establish a connection using different source and destination ports until it finds a combination that hashes to the desired RX queues. There are two exciting aspects of this approach.

***Decoupling flow identifiers from RSS.*.** Unlike bare-metal NICs that allow offloading to the hardware advanced filters over arbitrary packet header fields, cloud NICs only support basic RSS hashing on the standard four-tuple for packet steering. This couples RSS hashing and flow identifiers, limiting flexibility in how we can exploit randomization. Machnet breaks this coupling: instead of using UDP ports to define flows, we introduce two additional 16-bit port fields in the Machnet-specific packet header. As discussed below, this reduces the number of packets that Machnet must spray during connection setup. A side benefit is that it avoids reducing the number of supported flows between two servers as the number of RX queues increases.

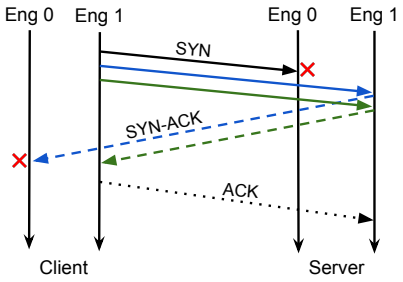


Figure 5. *SYN and SYN-ACK spraying in Machnet to establish a flow between engine #1 at the client and engine #1 at the server. This figure shows the unoptimized version where the client sprays n^2 SYN packets. The green SYN and SYN-ACK packets are the ones that hash to the correct RX queue.*

SYN and SYN-ACK spraying. We first discuss an unoptimized method that requires a relatively large number of packets to establish a connection: During the initial connection handshake, the client-side Machnet sends a batch of SYN packets with randomly selected UDP source and destination port pairs (Figure 5). The server responds with a corresponding SYN-ACK packet that hashes to the correct server RX queue for each SYN packet. The client uses the first SYN-ACK packet received at the correct RX queue to establish the connection and discard the rest. If no SYN packet reaches the correct engine, the client repeats after a timeout (300 ms by default) with different UDP port pairs.

How many SYN packets should the client send? Let us assume two Machnet machines with n engines each and a uniformly random RSS hash function. The probability of a SYN/SYN-ACK packet hashing to the correct server/client queue is $\frac{1}{n}$. Thus, the probability of a connection being established by a given packet is $\frac{1}{n*n}$. Using basic probability, the number of packets required to establish a connection with 95% likelihood is $\log(1-0.95)/\log(1-\frac{1}{n*n})$. This equals 47 packets for $n = 4$ engines and 191 for $n = 8$ engines. The remaining 5% of flows are established after a timeout and retry.

Reducing sprayed packets. With our decoupling of flow identifiers from RSS, the server does not simply need to reflect the client's SYN UDP port pair. Instead, we can use a different UDP port pair for the reverse direction. This does

[2] Since we use one RX queue per core, we use core and RX queue interchangeably.

not change the Machnet-specific port fields, so each side can still associate packets with the correct flow. The client sends some SYN packets with random UDP port pairs in the optimized method. On receiving the first SYN, the server responds with an equal number of SYN-ACK packets with its randomly chosen UDP port pairs. As long as both the SYN and SYN-ACK packets hash to the correct remote RX queue with probability at least $\sqrt{0.95}$, we keep the 95% probability of connection setup. This requires each side to send $\log(1-\sqrt{0.95})/\log(1-\frac{1}{n})$ packets; the total number of packets for the connection setup is two times this value, which equals 25 and 55 packets for $n = 4$ and $n = 8$ engines, respectively.

Communicating successful UDP ports. The server and client communicate the successful UDP port pairs (used for future packets in the flow) to each other in the SYN-ACK and ACK packets, respectively. The server includes the correctly-hashed SYN packet's UDP port pair in its SYN-ACK packets' payload. Similarly, the client includes the successful SYN-ACK packet's UDP port pair in the subsequent ACK packet and first data packet. Note that the client and server may use different UDP port pairs for each flow direction.

4.3 Opportunistic Hardware Offloading

While designed around the LCD NIC model for baseline portability, Machnet incorporates a modular architecture to opportunistically exploit advanced, platform-specific hardware and virtualization capabilities. We demonstrate this adaptability with a couple of examples. Machnet's default `RSS--` scaling mechanism (section 4.2.1) uses randomization to map flows to CPU cores because reliably inverting the RSS hash is not part of the LCD model; some cloud vNICs hide the RSS key or use inconsistent hash formats. However, when deployed in an environment where the RSS key is accessible, a provider-specific module activates. This module bypasses the default packet spraying and instead deterministically maps flows by calculating the precise source ports. Similarly, Machnet's I/O path adapts to take advantage of zero-copy transmission. The baseline implementation operates by copying packet data to conform to the LCD model (section 3.1), but it detects when running on Azure VMs that support direct memory registration. In this mode, it bypasses the standard data copy for outgoing packets, working directly with memory handles registered with the vNIC to reduce inter-process communication overhead. This strategy of pairing a universally compatible core with opportunistic, platform-specific optimizations ensures that Machnet achieves both broad portability and maximal performance on capable hardware.

4.4 API calls and descriptions

The APIs in Machnet are designed to closely resemble traditional BSD-like socket APIs summarized in Table 3.

Table 3. *Summary of Machnet API Functions.*

API Function	Description
`Machnet_init()`	Initializes the library.
`Machnet_attach()`	Attaches the shared memory.
`Machnet_bind()`	== BSD bind()
`Machnet_listen()`	== BSD listen().
`Machnet_connect()`	== BSD connect().
`Machnet_send()`	== BSD send().
`Machnet_recv()`	== BSD recv().

5 Evaluation

We show that Machnet meets all three requirements set in § 2.3. In § 5.2, we show that it is compatible with the vNICs of all three major cloud providers (AWS, Azure, GCP). In § 5.3.3, 5.3.1, 5.4.5, we show that Machnet is compatible with the diverse application execution models of cloud applications. Finally, in § 5.4, we show that Machnet achieves latency comparable to or better than other userspace network stacks that do not adhere to the LCD model. Throughout the evaluation, we compare Machnet to three alternatives, noting that Machnet is a network stack-only solution. We compared it against existing network stacks rather than full operating systems approaches, including Shenango [75], and Junction [47]:

Linux TCP/IP. We use this as a representative of OS-based approaches. We use Sockperf [9] for our experiments. Libraries built atop this (e.g., gRPC) will have higher latency and lower throughput than Linux TCP/IP.

eRPC. [60] eRPC is not an LCD-compatible network stack, and it only runs on a limited number of Azure VMs that support RSS reconfiguration. We use this (commit b8df1bc) to represent library-based stacks (e.g., Demikernel). Unlike Machnet, eRPC supports only one process per NIC and only one thread RX queue (= CPU core).

TAS. We use this [62] (commit d3926ba) as a representative for sidecar-based stacks (e.g., Snap, which is closed-source). *Differences.* Unlike Machnet, TAS's sidecar process uses at least two CPU cores, one for the fast path and one for the slow path. This presents a common problem for small 1–8 core cloud VMs (e.g., Cortez et al. reported that more than 95% of Azure VMs are 8-core or smaller [35]). While TAS supports multiple application processes, it does not provide performance isolation among them (§ 5.4.5) since TAS lacks a technique like `RSS--`.

Why no Snap. Unfortunately, we cannot compare it due to its proprietary nature. Also, Snap's broader scope overlaps with Machnet only in the PonyExpress component.

5.1 Evaluation setup

To demonstrate our LCD NIC model fits all modern DPDK-capable Ethernet NICs, we have tested Machnet on various

Cloud provider	Size	$p50^{th}$	$p99^{th}$	$p99.9^{th}$
Microsoft Azure	64 B	27 / 62	32 / 79	49 / 118
	32 kB	81 / 150	97 / 276	159 / 1288
Amazon EC2	64 B	48 / 98	53 / 131	57 / 274
	32 kB	224 / 158	240 / 298	257 / 354
Google Cloud	64 B	65 / 99	111 / 134	164 / 204
	32 kB	221 / 111	273 / 167	335 / 225

Table 4. *Machnet's round trip latency (µs) for echo messages. Each cell reports Machnet latency / Linux TCP latency (placeholder values).*

cloud and bare-metal NICs. Machnet works on Azure, AWS, and GCP, and it has been tested on several bare-metal NICs, including ConnectX-3 to ConnectX-6 and Intel E810.

Configuration. Unless mentioned otherwise, we evaluate Machnet on a cluster of VMs on Azure's public cloud. Most of our optimization effort went to Azure, which provides the best latency (§5.4.1). The VMs are in the same region and availability zone in all experiments. The VMs have accelerated networking and run Ubuntu 22.04.2 with Linux kernel 6.2. We do not enable proximity placement groups to keep results generally applicable, which can further reduce latency between VMs in the same availability zone. Our experiments use eight-core F8s_v2 VMs with 16 GB DRAM. We chose small VMs since those represent the large majority of VMs in public clouds, e.g., **(author?)** [35] reports that over 98% of Azure VMs have eight or fewer CPU cores.

Unless otherwise mentioned, we use the smallest configuration for the Machnet and TAS sidecars, i.e., one engine core for Machnet and two cores for TAS (one each for its fast and slow paths). For the Linux TCP/IP stack, Nagle's algorithm is disabled. We use huge pages for the DPDK-based stacks (i.e., Machnet, TAS, and eRPC) and pin stack processes to CPU cores. We omit the more invasive tuning techniques, such as isolated CPU cores (`isolcpus`) or interrupt steering.

5.2 Supported cloud platforms

Table 4 shows that Machnet successfully runs on all three major public clouds, achieving our primary goal of creating a userspace stack for this environment. We used AWS's ENA express[21] networking and Google Cloud's newer gVNIC DPDK driver[50]. For the experiment, we measure the round-trip latency of 64 B and 32 kB messages between two VMs in the same availability zone, each running a single-threaded echo application. Machnet achieves excellent tail latency (up to even $p99.9^{th}$) for small 64 B messages across Azure, EC2, and GCP. In larger 32 kB messages – an explicit non-goal – the results are mixed as Machnet suffers from the additional data copies caused by cross-core data movement: Machnet is better than TCP on Azure and EC2, and worse in GCP.

Note that the latencies across cloud providers should not be directly compared since we have not yet used comparable VM types or hardware generations. For example, we

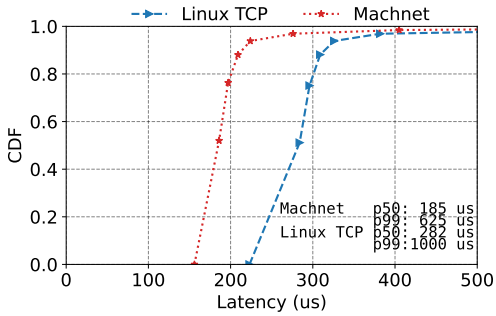


Figure 6. *Latency for Hashicorp's Golang-based Raft.*

enabled placement groups to reduce the inter-VM distance on Amazon EC2 but not on the other clouds. We believe that a large-scale evaluation of userspace networking in public clouds – now possible with Machnet – is an important direction for future work.

We next demonstrate Machnet's performance benefits and ease of use in end-to-end applications. In all cases, we make no changes to the core application code and only modify their networking calls to use Machnet. Since Machnet uses only the available networking feature-set of each cloud provider, then performance variations with Machnet are expected. For instance, the GCP DPDK driver misses some offloading features, resulting in lower performance, whilst those of Azure are available and deliver better performance.

5.3 Macrobenchmarks

5.3.1 Golang-based Raft over Machnet. State machine replication (SMR) [66, 73, 74] is an essential part of cloud applications, used in various applications ranging from configuration management control planes [22, 28, 44, 56], to data-intensive systems [34, 42, 81] that manage petabytes of data and millions of requests per second. In SMR, each server in a cluster of servers runs an identical sequence of commands, allowing the service to withstand failures. Since SMR protocols are often on the critical path of data-intensive systems, they must deliver low latency. Production-grade implementations of SMR are highly complex, so integrating them with Machnet is a good test of Machnet's generality. For our study, we chose Hashicorp's production Raft implementation [54] for two reasons. First, it is written in a non-systems language (Go), showing Machnet's broad applicability. Second, it provides a generic network interface that permits different implementations.

Golang high-level language binding. We use cgo to implement Golang bindings for Machnet's shim library. To evaluate its performance, we use a simple client-server echo application written in Go and compare its performance with its C++ counterpart for latency and throughput. We use a similar setup as in Section 5.4. Our experiments show that the two perform similarly, with Golang performing slightly

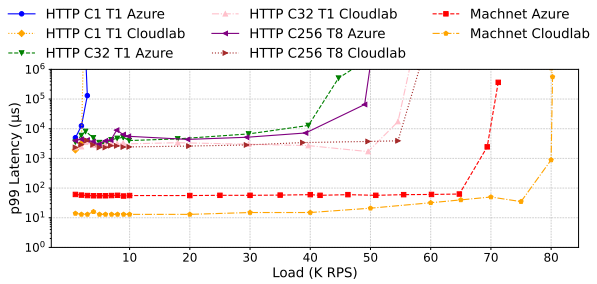


Figure 7. *$p99^{th}$ latency of an HTTP web server comparing Machnet to a standard Linux stack on both bare-metal (Cloudlab) and cloud (Azure) platforms. Machnet achieves lower latency and higher throughput than all tested Linux configurations (C=connections, T=threads).*

worse due to cgo's memory copy overhead. For 32 kB echo messages, the Go version has only 8% higher $p99^{th}$ latency and 5% lower throughput.

Raft benchmark. We compared our port of Hashicorp's Raft implementation to Machnet with Hashicorp's official Linux TCP implementation. Our Raft setup uses one client that issues commands (80 B each, similar to eRPC's evaluation [60]) to the Raft cluster, which replicates them to an in-memory log. We use three VMs, two running the SMR protocol (one leader and one follower) and one serving as a client load generator. The client sends a request to the leader and waits for a response; the leader responds after replicating the data to its follower. Each server VM has two active CPU cores: one for the application and one for Machnet.

Figure 6 shows that Machnet outperforms Linux kernel TCP, with 34% lower median latency and 37% lower median $p99^{th}$ latency. With Machnet, we achieve microsecond-scale latency even at the $p99^{th}$ percentile, where the Linux kernel TCP counterpart exceeds a millisecond.

5.3.2 Web Server Evaluation in Cloud Environment.

Figure 7 shows the throughput of the Mongoose web server in both bare metal and Azure cloud environments. Machnet's $p99^{th}$ latency remains exceptionally low, starting at approximately $12\mu s$ and staying below $30\mu s$ until it reaches a peak throughput of over 75,000 requests per second (RPS). This trend continues in the Azure cloud environment, where Machnet maintains a stable $p99^{th}$ latency of about $60 - 70\mu s$ up to nearly 68,000 RPS. The Linux stack on Azure performs at a much higher latency of over $4,000\mu s$ and struggles to handle loads beyond 40,000 RPS. In both environments, Machnet consistently delivers lower latency and supports a higher request load.

Figure 7 also highlights the inherent overhead and variability of a public cloud platform like Azure compared to a bare-metal environment like Cloudlab. Across all test configurations, for both Machnet and the Linux stack, the $p99^{th}$ latencies are consistently lower and more stable on Cloudlab. For instance, Machnet's baseline latency on Azure is approximately $60\mu s$, about five times higher than its $12\mu s$ latency on Cloudlab. A similar performance gap exists for the Linux stack, where the HTTP C256 T8 configuration on Azure has roughly double the baseline latency of its Cloudlab counterpart ($\sim 4,000\mu s$ vs. $\sim 2,000\mu s$). The graph also shows greater performance variability on Azure, visible in the more erratic plots of the HTTP configurations (e.g., HTTP C32 T1 Azure) compared to their smoother Cloudlab equivalents. While bare-metal infrastructure offers superior raw performance, these results show that Machnet's design effectively mitigates a significant portion of networking overhead, delivering substantial improvements even within a more variable cloud environment.

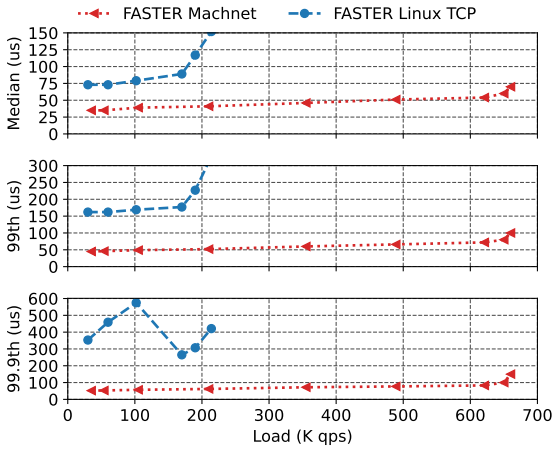


Figure 8. *Performance of the FASTER key-value store over Machnet and Linux TCP/IP. Machnet achieves 3.3x higher throughput and 80% lower $p99^{th}$ latency compared to Linux.*

5.3.3 FASTER key-value store over Machnet.

FASTER is a production-grade high-performance key-value store [14, 32]. For this experiment, we created a single-threaded server application that uses FASTER's C++-based in-memory hash table for storage (commit 0116754) and Machnet for networking. We pre-populate the server's hash table with 100 million eight-byte key-value pairs. Two client VMs generate the workload, which consists of 100% GET requests to emulate a read-heavy workload. We vary the load by (1) increasing the number of concurrent connections from each client VM from one to 20 and (2) increasing the number of outstanding requests per connection from one to eight. A third client VM acts as our latency probe, using one thread to send GET requests to the server one at a time.

Single-threaded performance. Figure 8 compares the throughput and latency achieved over Machnet and Linux's TCP/IP stack. The Machnet-based implementation achieves 3.3x higher throughput, with 700K RPS compared to Linux TCP's 210K RPS. At 210K RPS, Machnet's $p99^{th}$ latency is

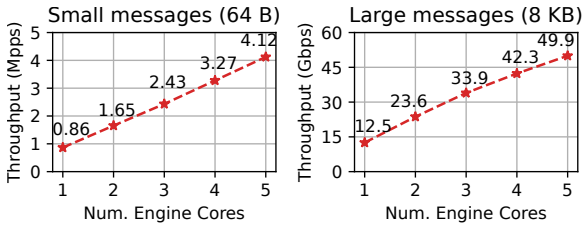


Figure 9. *CPU core scalability with large (8 KB) and small messages (64 B). Machnet cores and engines are the same.*

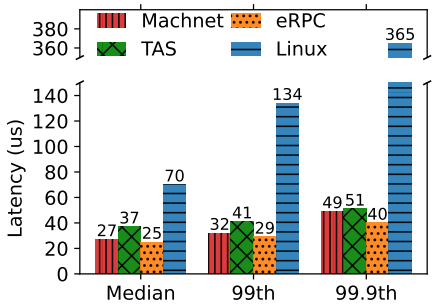


Figure 10. *Round-trip latency for 64-byte messages.*

80% lower, achieving 50 µs compared to Linux TCP's 250 µs. Another crucial improvement is in CPU utilization. At its peak throughput, the in-kernel TCP stack causes 100% CPU utilization on all eight cores. In contrast, the Machnet-based implementation only utilizes two CPU cores (application and one engine), i.e., 75% lower.

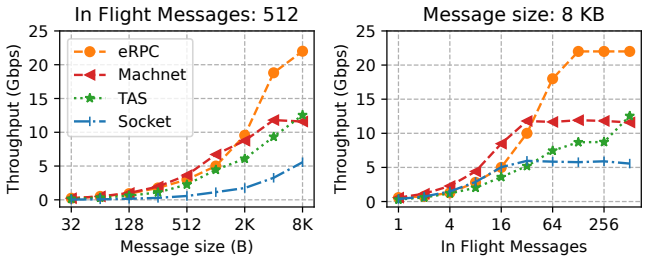
Multi-threaded performance. We evaluate Machnet's scalability to multiple engine threads based on RSS-- (Section 4.2.1) as follows. In this experiment, we built a custom application to demonstrate scalability in terms of higher throughput. Machnet scales to high throughput by allocating cores both in terms of high packet rates (important for small messages) and large bandwidths (for large messages). Figure 9 shows the scalability of Machnet by increasing the number of cores from 1 to 5. Machnet reaches 4.12 Mpps for 64 B messages and 49.9 Gbps for 8 kB messages.

5.4 Microbenchmarks

5.4.1 Latency. Figure 10 compares Machnet's latency with the alternatives in the latency experiment above, with 64 B messages. We make two observations.

First, Machnet's latency is comparable to eRPC, despite Machnet's use of only the LCD NIC model and support for diverse execution models. Specifically, Machnet's median and $p99^{th}$ latency is within 10% of eRPC and 23% at $p99.9^{th}$. Taking median latency as the example, Machnet's 27 µs latency is 2 µs higher than eRPC[3]. This extra latency comes primarily from SHM communication between the application and

[3]Demikernel [84] reports 15 µs lower round-trip latency on Azure. Our latencies are different because we use a different region.



(a) *Message size effect* **(b)** *Window size effect*

Figure 11. *Open-loop throughput experiment on large VMs (Azure D96as_v5) with **25 Gbps** network bandwidth capacity on a single CPU core. Although Machnet is not optimized for ultra-high throughput, it is still able to achieve reasonably high throughput for cloud VMs.*

Machnet's sidecar process, which costs around 250 ns for each one-way SHM crossing. Interestingly, while this overhead is significant compared to a small bare-metal testbed (e.g., eRPC reports 2.3 µs within a rack), in large cloud data centers, the network's base latency dwarfs this overhead.

Second, Linux TCP's latency is 2.6x, 4.2x, and 7.4x higher than Machnet at median, $p99^{th}$, and $p99.9^{th}$, respectively. This confirms prior findings that kernel-bypass's latency benefits, arising from avoiding context switches and heavy kernel processing, are significant even in cloud datacenter scenarios [41, 84].

5.4.2 Throughput. We compared the single-core throughput of Machnet against Linux's network-stack and state-of-the-art kernel bypass systems, i.e., TAS [62] and eRPC [60]. Since both message size and the number of concurrent messages in flight affect throughput, we varied both parameters. Figure 11a shows the throughput sustained by each system for different message sizes while having at most 512 requests in flight between client and server. Server response size is 64 B. Figure 11b shows the achieved throughput for the case of having a variable number of in-flight messages. Each request is 8 kB while the server response is still 64 B. Machnet's throughput is comparable with state-of-the-art kernel bypass network stacks (similar to TAS, worse than eRPC) for variable message sizes and number of concurrent requests, despite it being optimized for latency rather than throughput.

5.4.3 RSS-- connection setup latency. We measured Machnet's connection setup latency on bare-metal machines (to avoid cloud-induced spikes) by varying the number of client and server engines from 1 to 8 across 10K sequential connections to understand the overhead of our RSS-- design (packet-spraying technique with multiple retries). Figure 12 shows that RSS-- achieves median connection setup latencies below 1 ms. Latency increases at the tail due to higher chances of selecting the wrong source port. If no SYN packet

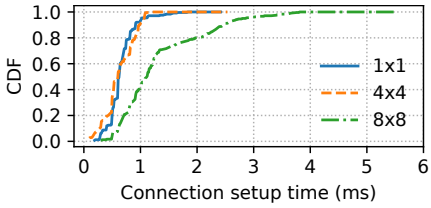


Figure 12. *Connection setup latency in Machnet. In all cases, the median of connection setup time is less than ~ 1 ms (legend: number of sender and receiver engines).*

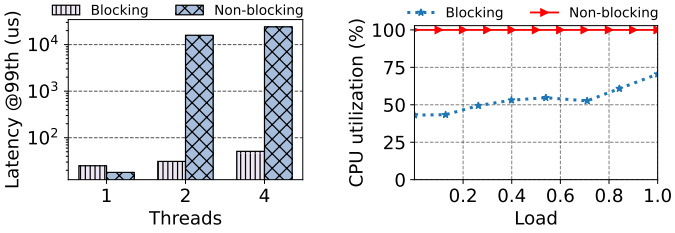
from the initial batch reaches the remote engine, or no SYN-ACK returns to the sender, Machnet triggers a SYN timeout and retries with an exponentially larger batch.

5.4.4 Impact of advanced NIC features. To make the tradeoffs explicit, we now summarize how each missing feature affects performance, and point to the microbenchmarks that quantify the impact.

Flow steering only affects connection establishment. The lack of flow steering impacts only connection setup. Figure 12 quantifies this cost: even at 8×8 sender/receiver engines, median connection setup time remains below ~1 ms, with tail increases only when the initial probe batch misses the target queue and Machnet retries.

Datapath acceleration features mostly affect peak throughput but not latency. eRPC leverages several datapath acceleration features, notably TX DMA from application memory (enabling zero-copy transmission) as well as deep/multi-packet receive queues. These capabilities show up in our microbenchmarks. On the throughput side, zero-copy together with deep RX queues and multi-packet receive descriptors reduces per-packet descriptor processing overhead and helps avoid host-side drops at high rates; these optimizations are not possible under the LCD model's constrained queues/descriptors. Accordingly, in the open-loop throughput experiment (Figure 11), Machnet's single-core throughput is similar to TAS but 45% below eRPC for the large message/window configurations. However, zero-copy has only a minimal effect on latency for small messages. In the 64 B ping-pong latency experiment, eRPC achieves only 2μs lower round-trip latency than Machnet (Figure 10), consistent with avoiding Machnet's extra IPC hop.

5.4.5 Supporting diverse execution models. Performance isolation Machnet avoids inter-application performance interference by using RSS−− to map flows to specific Machnet engines. TAS supports only random flow-to-engine mapping (determined by the NIC's randomized RSS), and thus suffers from interference. Our experiment to demonstrate this works is as follows. We run four single-threaded echo applications at a server VM: three throughput-intensive and one latency-sensitive. Client VM#1 generates high-rate traffic to the throughput-intensive applications, using 16 flows with eight 64 B in-flight messages per flow. Client



(a) *Tail latency* **(b)** *CPU utilization*

Figure 13. *Effectiveness of Machnet's blocking receives.*

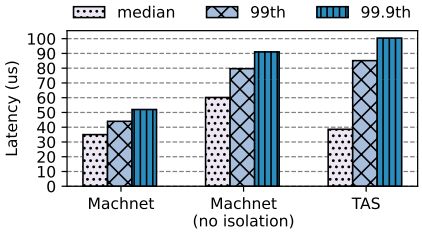


Figure 14. *Inter-application performance isolation in Machnet.*

VM#2 acts as a latency probe, sending one 64 B echo at a time to the latency-sensitive application. Since only sidecar-based approaches support multiple concurrent applications, we exclude eRPC from this evaluation.

We use two engines for Machnet at the server, one paired with the throughput-oriented applications and one dedicated to the latency-sensitive application. For TAS, we use two fast-path cores to match Machnet's fast-path cores in addition to TAS's one slow-path core. Figure 14 shows that with Machnet, client VM#2 achieves nearly 50% better tail latency than TAS. We also show the latency with Machnet without isolation, i.e., when flows are randomly mapped to engines. Without isolation, Machnet's tail latency, too, suffers from interference, showing the importance of RSS−−.

Blocking receive. We have found that a typical application pattern is one where there are more application threads needing network I/O than the number of CPU cores (Section 3.2), which requires support for blocking receive. We evaluate Machnet's blocking receive using an experiment in which a Machnet-based echo server application runs multiple (up to 4) threads, all sharing a single CPU core. A single-threaded client application on a different VM generates a workload with 64 B request-response messages.

To measure latency, we configure the open-loop client to send load at full rate to the server. Figure 13a shows that when the server applications use blocking receive, the client gets low tail latency even when the application threads share a single core. In contrast, with multiple non-blocking server threads polling at the same CPU core, tail latency spikes to

over 10 ms. This happens because a server thread may need to wait multiple Linux scheduling intervals before running.

Figure 13b shows that blocking receives reduces CPU utilization. We run the server with four threads for this experiment and configure the client to generate requests in an open loop, varying between 70K–700K requests per second (RPS). With blocking receives, the server core's CPU utilization stays below 75% even when handling 700K RPS. In contrast, the CPU utilization of busy polling is always 100%, regardless of the load.

6 Conclusion

We present Machnet, a userspace networking stack designed for the cloud. Unlike bare-metal NICs, cloud vNICs lack many expected features, motivating our Least Common Denominator NIC model, which treats features as absent-by-default. By moving beyond the traditional libOS model and supporting multiprocess, multithreaded, and language-diverse applications, Machnet allows cloud-portable userspace networking and opens the door for future large-scale evaluations and applications.

Acknowledgments

We thank our shepherd, Babak Falsafi; the anonymous reviewers; Anuj Kalia, especially for his outstanding and continuous contributions to this project; and Andrew Moore for his helpful feedback and support. Partially funded by the European Union. Views and opinions expressed are, however, those of the author(s) only and do not necessarily reflect those of the European Union or the European Health and Digital Executive Agency. Neither the European Union nor the granting authority can be held responsible for them.

In memory of Prateesh Goyal

A Artifact Appendix

A.1 Abstract

The artifact contains the source code for Machnet and instructions for building and measuring application benchmark performance. To support diverse software environments, we provide native language bindings for Python, JavaScript, C#, Go, and Rust. To facilitate ease of integration, the artifact includes reference examples demonstrating how to incorporate Machnet into custom applications. We also provide a set of scripts and instructions to reproduce the key results presented in the paper. These scripts automate the cloud VM provisioning and containerization.

A.2 Artifact check-list (meta-information)

- **Program:** DPDK-23.11.
- **Compilation:** GCC 10+.
- **Run-time environment:** Ubuntu 22.04 LTS.
- **Hardware:** Two servers with DPDK-compatible NICs connected by a network.
- **Experiments:** Application benchmarks to measure network throughput and latency: Echo, Raft, HTTP web-server, and FASTER
- **How much disk space required (approximately)?:** 30GB, including minimal Ubuntu OS 22.04 LTS.
- **How much time is needed to prepare workflow (approximately)?:** 30 minutes.
- **How much time is needed to complete experiments (approximately)?:** 2 hours.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** MIT.
- **Workflow automation framework used?:** No, but scripts are provided to automate deployment and measurements.
- **Archived (provide DOI)?:** https://doi.org/10.5281/zenodo.18248519

A.3 Description

A.3.1 How to access.

The artifact, which is zipped, is available on Zenodo: https://zenodo.org/records/18248519. The source code, scripts and extensive documentation as well as guides are also available on GitHub: https://github.com/microsoft/machnet.

A.3.2 Hardware dependencies.

Machnet requires machines equipped with DPDK-supported NICs and user-space packet I/O support. All experiments use two servers: one running the Machnet server application and the other acting as a load generator, as shown in Figure 15. The servers can be deployed either on bare-metal machines or as virtual machines, provided that DPDK requirements (e.g., hugepages and NIC passthrough) are satisfied.

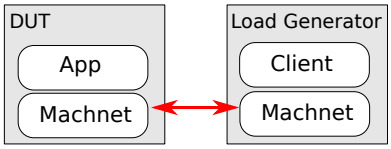


Figure 15. *Overview of experiment setup.*

We evaluated Machnet on major cloud platforms (Azure, AWS, and Google Cloud) as well as on bare-metal infrastructure provided by CloudLab. Table 5 summarizes the machine configurations used in our experiments.

For artifact evaluation, we recommend using CloudLab, as it is accessible to researchers providing direct access to bare-metal machines. In particular, CloudLab `xl170` nodes are equipped with Intel Xeon E5-2640v4 CPUs, 64 GB of DDR4 memory, and two Mellanox ConnectX-4 25 GbE NICs, which are fully compatible with DPDK and sufficient to reproduce all Machnet experiments.

A.4 Installation

Machnet must be installed on *both* machines participating in the experiment: one running the application server and one acting as the load generator.

Environment	Machine Type
Azure	F8s_v2, D96as_v5
AWS EC2	c7i.2xlarge
Google Cloud	c3-standard-4
Cloudlab	xl170

Table 5. *We tested Machnet in different environments and on different hardware. Details of which is availbe in this table.*

First clone the Machnet repository and install all required system dependencies using the provided installation script:

```
git clone --recursive https://github.com/microsoft/machnet.
    git
cd machnet
./scripts/machnet_install.sh
```

After dependencies are installed, Machnet is built from source using CMake. This step compiles the Machnet runtime, client library, and benchmark applications:

```
mkdir -p build
cd build
cmake -DCMAKE_BUILD_TYPE=Release ..
make -j
```

After a successful build, the Machnet runtime is launched as a standalone user-space process using the `machnet.sh` script. The Machnet runtime initializes DPDK, configures the user-space network datapath, and mediates all access to the NIC on behalf of applications:

```
./machnet.sh --mac <NIC MAC address> --ip <NIC IP address>
```

Prior to launching the runtime, the NIC must be unbound from the kernel network stack and bound to an appropriate user-space driver (e.g., `uio_hv_generic` on Azure or `vfio-pci` on bare-metal platforms). The NIC's IP and MAC addresses must be recorded before unbinding, as they are passed explicitly to the Machnet runtime at startup.

Once the Machnet runtime is running on both machines, applications link against the Machnet client library and communicate with the runtime via shared memory using a sockets-like API. Applications do not directly interact with DPDK and do not require DPDK-specific build or runtime configuration.

Detailed, platform-specific instructions for dependency installation, NIC configuration, launching the Machnet runtime, and running benchmarks are available at: https://github.com/microsoft/machnet/blob/main/docs/INTERNAL.md

A.5 Experiment workflow

Each experiment follows a fixed workflow. First, the Machnet runtime is launched on both machines after binding the dedicated NICs to user-space drivers. Next, the application or benchmark is started on top of the running Machnet runtimes.

Machnet includes an end-to-end microbenchmark, `msg_gen`, which is used in our evaluation to measure throughput and latency. After the Machnet runtime is running on both machines, `msg_gen` is executed in a standard client–server configuration. On the server machine, `msg_gen` is started in server mode:

```
./build/src/apps/msg_gen/msg_gen --local_ip <server NIC IP>
```

On the load-generator machine, `msg_gen` is started in client mode, specifying both the local and remote NIC IP addresses:

```
./build/src/apps/msg_gen/msg_gen \
  --local_ip <client NIC IP> \
  --remote_ip <server NIC IP>
```

The client reports message throughput and latency percentile statistics (e.g., P50, P99, and P99.9). Additional benchmark parameters, including message size, request rate, and experiment duration, can be configured via command-line options listed by:

```
./build/src/apps/msg_gen/msg_gen --help
```

In addition to `msg_gen`, the Machnet repository includes several example applications that demonstrate how to build and run services on top of the Machnet runtime. These applications are located in the `examples/` directory and can be executed following the same workflow as the microbenchmark (i.e., launching the Machnet runtime first, followed by the application). Each example includes minimal documentation and command-line options illustrating how to configure local and remote endpoints. Detailed instructions for running these applications are provided in the Machnet repository.

A.6 Evaluation and expected results

The performance results obtained from these experiments are expected to closely match those reported in the paper. When running on bare-metal platforms, performance may be equal to or better than the reported numbers, as bare-metal deployments avoid sources of variability common in cloud environments and typically provide lower and more stable latency. In contrast, experiments conducted on cloud virtual machines may exhibit higher variance due to factors such as multi-tenancy, noisy neighbors, and shared network infrastructure; nevertheless, the overall performance trends and relative comparisons should remain consistent. For reproducible evaluation with minimal variability, we recommend using CloudLab `xl170` bare-metal machines, which are easily accessible to researchers and provide hardware configurations sufficient to reproduce all results.

References

[1] Google production workload. https://cloud.google.com/blog/topics/systems/workload-traces-for-google-warehouse-scale-computers.

[2] High performance server-side application framework. https://github.com/scylladb/seastar.

[3] Intel® 82599ES 10 Gigabit Ethernet Controller. https://ark.intel.com/content/www/us/en/ark/products/41282/intel-82599es-10-gigabit-ethernet-controller.html.

[4] Mellanox wins $200m Google, Microsoft deals. https://en.globes.co.il/en/article-1000857043.

[5] Memcached. https://memcached.org/.

[6] Nvidia messaging accelerator (vma). https://github.com/Mellanox/libvma.

[7] Oracle cloud infrastructure documentation: Flexbile shapes. https://docs.oracle.com/en-us/iaas/Content/Compute/References/computeshapes.htm#Compute_Shapes.

[8] Sizes for virtual machines in azure. https://learn.microsoft.com/en-us/azure/virtual-machines/sizes.

[9] sockperf. https://github.com/Mellanox/sockperf.

[10] Specify the maximum number of nics per virtual machine. https://ftpdocs.broadcom.com/cadocs/0/CA%20Server%20Automation%2012%206-ENU/Bookshelf_Files/HTML/Admin/specify_maximum_NICs_per_VM.html#:~:text=Administrators%20can%20specify%20the%20maximum,default%20and%20maximum%20is%2010.

[11] Aws nitro system. https://aws.amazon.com/ec2/nitro/, 2024. Accessed: 2024-05-05.

[12] Connectx-3 ethernet single and dual sfp+ port adapter card user manual. https://network.nvidia.com/pdf/user_manuals/ConnectX-3_Ethernet_Single_and_Dual_SFP+_Port_Adapter_Card_User_Manual.pdf, 2024. Accessed: 2024-05-05.

[13] Elastic fabric adapter. https://aws.amazon.com/hpc/efa/, 2024. Accessed: 2024-05-05.

[14] Faster: A fast concurrent persistent key-value store and log. https://microsoft.github.io/FASTER/, 2024. Accessed: 2024-05-05.

[15] Gve poll mode driver. https://doc.dpdk.org/guides/nics/gve.html, 2024. Accessed: 2024-05-05.

[16] High-performance computing on infiniband enabled hb-series and n-series vms. https://learn.microsoft.com/en-us/azure/virtual-machines/overview-hb-hc, 2024. Accessed: 2024-05-05.

[17] Introduce microsoft azure network adapter (mana) rdma driver. https://lwn.net/Articles/896120/, 2024. Accessed: 2024-05-05.

[18] Microsoft Azure Network Adapter (MANA) overview. https://docs.microsoft.com/en-us/azure/virtual-network/accelerated-networking-overview, 2024. Accessed: 2024-05-05.

[19] Netvsc poll mode driver. https://doc.dpdk.org/guides/nics/netvsc.html, 2024. Accessed: 2024-05-05.

[20] Sriov virtual functions (vfs). https://learn.microsoft.com/en-us/windows-hardware/drivers/network/sr-iov-virtual-functions--vfs-, 2024. Accessed: 2024-05-05.

[21] Amazon Web Services. Enhanced networking on linux - amazon elastic compute cloud. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/enhanced-networking.html, 2023. Accessed: 2023-12-07.

[22] Mahesh Balakrishnan, Jason Flinn, Chen Shen, Mihir Dharamshi, Ahmed Jafri, Xiao Shi, Santosh Ghosh, Hazem Hassan, Aaryaman Sagar, Rhed Shi, Jingming Liu, Filip Gruszczynski, Xianan Zhang, Huy Hoang, Ahmed Yossef, Francois Richard, and Yee Jiun Song. Virtual consensus in delos. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 2020.

[23] Deepak Bansal, Gerald DeGrace, Rishabh Tewari, Michal Zygmunt, James Grantham, Silvano Gai, Mario Baldi, Krishna Doddapaneni, Arun Selvarajan, Arunkumar Arumugam, Balakrishnan Raman, Avijit Gupta, Sachin Jain, Deven Jagasia, Evan Langlais, Pranjal Srivastava, Rishiraj Hazarika, Neeraj Motwani, Soumya Tiwari, Stewart Grant, Ranveer Chandra, and Srikanth Kandula. Disaggregating stateful network functions. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, 2023.

[24] Deepak Bansal, Gerald DeGrace, Rishabh Tewari, Michal Zygmunt, James Grantham, Silvano Gai, Mario Baldi, Krishna Doddapaneni, Arun Selvarajan, Arunkumar Arumugam, Balakrishnan Raman, Avijit Gupta, Sachin Jain, Deven Jagasia, Evan Langlais, Pranjal Srivastava, Rishiraj Hazarika, Neeraj Motwani, Soumya Tiwari, Stewart Grant, Ranveer Chandra, and Srikanth Kandula. Disaggregating stateful network functions. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, 2023.

[25] Tom Barbette, Georgios P. Katsikas, Gerald Q. Maguire, and Dejan Kostić. Rss++: Load and state-aware receive side scaling. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, CoNEXT '19, 2019.

[26] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the killer microseconds. In *Communications of the ACM*, volume 60, pages 48–54. ACM, 2017.

[27] 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)*. USENIX Association, October 2014.

[28] Marc Brooker, Tao Chen, and Fan Ping. Millions of tiny databases. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, 2020.

[29] Matthew Burke, Sowmya Dharanipragada, Shannon Joyner, Adriana Szekeres, Jacob Nelson, Irene Zhang, and Dan R. K. Ports. Prism: Rethinking the rdma interface for distributed systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21. Association for Computing Machinery, 2021.

[30] Caddy Web Server. Caddy: Powerful, enterprise-ready, open source web server with automatic https written in go. https://github.com/caddyserver/caddy, 2024. Accessed: 2024-05-03.

[31] Qizhe Cai, Midhul Vuppalapati, Jaehyun Hwang, Christos Kozyrakis, and Rachit Agarwal. Towards μs tail latency and terabit ethernet: Disaggregating the host network stack. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, 2022.

[32] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. Faster: A concurrent key-value store with in-place updates. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18. Association for Computing Machinery, 2018.

[33] Jingrong Chen, Yongji Wu, Shihan Lin, Yechen Xu, Xinhao Kong, Thomas Anderson, Matthew Lentz, Xiaowei Yang, and Danyang Zhuo. Remote procedure call as a managed system service. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, 2023.

[34] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Dale Woodford, Yasushi Saito, Christopher Taylor, Michal Szymaniak, and Ruth Wang. Spanner: Google's globally-distributed database. In *OSDI*, 2012.

[35] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 153–167, 2017.

[36] Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Arefin, Anshuman Gupta, Brian Fahs, Dima Rubinstein, Enrique Cauich Zermeno, Erik Rubow, James Alexander Docauer, et al. Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization. In *Networked Systems Design and Implementation (NSDI)*. USENIX, 2018.

[37] Jeffrey Dean and Luiz André Barroso. The tail at scale. In *Communications of the ACM*. ACM, 2013.

[38] Christina Delimitrou and Christos Kozyrakis. ibench: Quantifying interference for datacenter applications. In *2013 IEEE International Symposium on Workload Characterization (IISWC)*, 2013.

[39] DPDK. Ena poll mode driver. https://doc.dpdk.org/guides/nics/ena.html, 2024. Accessed: 2024-05-05.

[40] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, 2014.

[41] Tamer Eldeeb, Xincheng Xie, Philip A. Bernstein, Asaf Cidon, and Junfeng Yang. Chardonnay: Fast and General Datacenter Transactions for On-Disk Databases. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, 2023.

[42] Mostafa Elhemali, Niall Gallagher, Nick Gordon, Joseph Idziorek, Richard Krog, Colin Lazier, Erben Mo, Akhilesh Mritunjai, Somasundaram Perianayagam, Tim Rath, Swami Sivasubramanian, James Christopher Sorenson III, Sroaj Sosothikul, Doug Terry, and Akshat Vig. Amazon DynamoDB: A scalable, predictably performant, and fully managed NoSQL database service. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 1037–1048, Carlsbad, CA, July 2022. USENIX Association.

[43] Wei Bai et al. Empowering Azure Storage with RDMA. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, 2023.

[44] etcd Authors. etcd: A distributed, reliable key-value store for the most critical data of a distributed system. https://etcd.io/, 2023.

[45] Daniel Firestone. VFP: A virtual switch platform for host SDN in the public cloud. In *Networked Systems Design and Implementation (NSDI)*. USENIX, 2017.

[46] 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 *Networked Systems Design and Implementation (NSDI)*. USENIX, 2018.

[47] Joshua Fried, Gohar Irfan Chaudhry, Enrique Saurez, Esha Choukse, Inigo Goiri, Sameh Elnikety, Rodrigo Fonseca, and Adam Belay. Making kernel bypass practical for the cloud with junction. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 55–73, Santa Clara, CA, April 2024. USENIX Association.

[48] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In *Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX, 2020.

[49] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19.

[50] Google Cloud. Using google virtual nic - compute engine documentation. https://cloud.google.com/compute/docs/networking/using-gvnic, 2023. Accessed: 2023-12-07.

[51] Google Cloud Platform. compute-virtual-ethernet-linux. https://github.com/GoogleCloudPlatform/compute-virtual-ethernet-linux/tree/v1.4.1, 2024. Version 1.4.1.

[52] 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*, pages 202–215, 2016.

[53] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. Re-Architecting Datacenter Networks and Stacks for Low Latency and High Performance. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2017.

[54] HashiCorp. Hashicorp raft: Golang implementation of the raft consensus algorithm. https://github.com/hashicorp/raft, 2023.

[55] HashiCorp. Vault: A tool for managing secrets, 2024. Accessed: 2024-05-03.

[56] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, page 11, USA, 2010. USENIX Association.

[57] Darby Huye, Yuri Shkuro, and Raja R. Sambasivan. Lifting the veil on Meta's microservice architecture: Analyses of topology and request workflows. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. USENIX Association, 2023.

[58] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mtcp: a highly scalable user-level {TCP} stack for multicore systems. In *Networked Systems Design and Implementation (NSDI)*. USENIX, 2014.

[59] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for μsecond-scale tail latency. In *Networked Systems Design and Implementation (NSDI)*. USENIX, 2019.

[60] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter RPCs can be general and fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, 2019.

[61] Magnus Karlsson and Björn Töpel. The path to dpdk speeds for af xdp. In *Linux Plumbers Conference*, volume 37, page 38, 2018.

[62] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr Sharma, Arvind Krishnamurthy, and Thomas Anderson. TAS: TCP acceleration as an OS service. In *European Conference on Computer Systems (EuroSys)*. ACM, 2019.

[63] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. R2P2: Making RPCs first-class datacenter citizens. In *Annual Technical Conference (ATC)*. USENIX, 2019.

[64] Xinhao Kong, Jingrong Chen, Wei Bai, Yechen Xu, Mahmoud Elhaddad, Shachar Raindel, Jitendra Padhye, Alvin R. Lebeck, and Danyang Zhuo. Understanding RDMA microarchitecture resources for performance isolation. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, 2023.

[65] Teemu Koponen, Keith Amidon, Peter Balland, Martin Casado, Anupam Chanda, Bryan Fulton, Igor Ganichev, Jesse Gross, Paul Ingram, Ethan Jackson, Andrew Lambeth, Romain Lenglet, Shih-Hao Li, Amar Padmanabhan, Justin Pettit, Ben Pfaff, Rajiv Ramanathan, Scott Shenker, Alan Shieh, Jeremy Stribling, Pankaj Thakkar, Dan Wendlandt, Alexander Yip, and Ronghua Zhang. Network virtualization in multi-tenant datacenters. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX Association, 2014.

[66] Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pages 51–58, December 2001.

[67] Bojie Li, Tianyi Cui, Zibo Wang, Wei Bai, and Lintao Zhang. Socksdirect: Datacenter sockets can be fast and compatible. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, 2019.

[68] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. Kv-direct: High-performance in-memory key-value store with programmable nic. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, 2017.

[69] Ilias Marinos, Robert N.M. Watson, and Mark Handley. Network stack specialization for performance. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, 2014.

[70] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C Evans, Steve Gribble, et al. Snap: a microkernel approach to host networking. In *Symposium on Operating Systems Principles (SOSP)*. ACM, 2019.

[71] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A Receiver-Driven Low-Latency Transport Protocol Using Network Priorities. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2018.

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

[73] Brian M. Oki and Barbara H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, PODC '88. Association for Computing Machinery, 1988.

[74] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14. USENIX Association, 2014.

[75] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *Networked Systems Design and Implementation (NSDI)*. USENIX, 2019.

[76] Boris Pismenny, Haggai Eran, Aviad Yehezkel, Liran Liss, Adam Morrison, and Dan Tsafrir. Autonomous NIC Offloads. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21. Association for Computing Machinery, 2021.

[77] Steve Pope and David Riddoch. Introduction to openonload—building application transparency and protocol conformance into application acceleration middleware. Technical report, Technical report, Solarflare Communication, April 2011. http://www …, 2011.

[78] George Prekas, Marios Kogias, and Edouard Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Symposium on Operating Systems Principles (SOSP)*. ACM, 2017.

[79] Deepti Raghavan, Shreya Ravi, Gina Yuan, Pratiksha Thaker, Sanjari Srivastava, Micah Murray, Pedro Henrique Penna, Amy Ousterhout, Philip Levis, Matei Zaharia, and Irene Zhang. Cornflakes: Zero-copy serialization for microsecond-scale networking. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, 2023.

[80] Matheus Stolet, Liam Arzola, Simon Peter, and Antoine Kaufmann. Virtuoso: High Resource Utilization and us-scale Performance Isolation in a Shared Virtual Machine TCP Network Stack, 2023.

[81] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. CockroachDB: The Resilient Geo-Distributed SQL Database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD 2020)*, SIGMOD '20. Association for Computing Machinery, 2020.

[82] Björn Töpel. Af_xdp: Introducing zero-copy support. *LWN.net*, 2018. Accessed on 2024-05-06.

[83] Juncheng Yang, Yao Yue, and K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *Operating Systems Design and Implementation (OSDI)*. USENIX, 2020.

[84] 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, 2021.