I'm Not Dead Yet! The Role of the Operating System in a Kernel-Bypass Era

Irene Zhang
Microsoft Research
irene.zhang@microsoft.com

Jing Liu
University of Wisconsin - Madison
jingliu@cs.wisc.edu

Amanda Austin
University of Texas - Austin
ajaustin@cs.utexas.edu

Michael Lowell Roberts
Microsoft Research
miroberts@microsoft.com

Anirudh Badam
Microsoft Research
anirudh.badam@microsoft.com

Abstract

Researchers have long predicted the demise of the operating system [21, 26, 41]. As datacenter servers increasingly incorporate I/O devices that let applications bypass the OS kernel (e.g., RDMA [12] and DPDK [15] network devices or SPDK storage devices), this prediction may finally come true. While kernel-bypass devices do eliminate the OS kernel from the I/O path, they do not handle the kernel's most important job: offering higher-level abstractions. This paper argues for a new *high-level, device-agnostic I/O abstraction for kernel-bypass devices.* We propose the Demikernel, a new library OS architecture for kernel-bypass devices. It defines a high-level, kernel-bypass I/O abstraction and provides user-space library OSes to implement that abstraction across a range of kernel-bypass devices. The Demikernel makes applications easier to build, portable across devices, and unmodified as devices continue to evolve.

CCS Concepts • **Software and its engineering → Operating systems**.

Keywords operating systems, datacenters, kernel bypass

ACM Reference Format:
Irene Zhang, Jing Liu, Amanda Austin, Michael Lowell Roberts, and Anirudh Badam. 2019. I'm Not Dead Yet! The Role of the Operating System in a Kernel-Bypass Era. In *Workshop on Hot Topics in Operating Systems (HotOS '19), May 13–15, 2019, Bertinoro, Italy.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3317550.3321422

HotOS '19, May 13–15, 2019, Bertinoro, Italy

ACM ISBN 978-1-4503-6727-1/19/05…$15.00
https://doi.org/10.1145/3317550.3321422

1 Introduction

Over the past decade, I/O devices in datacenter servers have sped up while CPU performance has stagnated. To compensate, servers increasingly integrate *I/O accelerators* – I/O devices with attached computational units that offload work from the CPU. Classic examples include TCP offloading[14, 46] and hardware virtualization (e.g., SR-IOV [23]); in the future, compression, encryption, machine learning, and more will be offloaded (e.g., using FPGAs [9, 42, 48] and other hardware [8, 10, 43, 49]).

In contrast to classic I/O accelerators, modern datacenter accelerators commonly offer kernel bypass along with their other features, as the kernel adds significant overhead to every I/O access [5, 31, 51]. These *kernel-bypass accelerators* implement the needed OS features – multiplexing, isolation, address translation – to let applications safely access I/O without going through the kernel. While today's kernel-bypass accelerators have some limitations, eventually accelerators will eliminate the OS kernel from the fast I/O path, relegating the kernel to slow, control-path operations.

This datacenter trend raises an important question for operating system researchers: *What role does the operating system play in the upcoming kernel-bypass era?* As shown in Figure 1, kernel-bypass accelerators remove the OS kernel from the I/O data path but do not replace all of its functionality. Importantly, they lack high-level, device-agnostic abstractions offered by OS kernels, like files, sockets, and pipes.

Existing kernel-bypass applications [16, 22, 25, 35] eschew classic OS abstractions and directly interact with the hardware. For example, the many distributed RDMA storage systems [11, 16, 29, 30, 44, 60] are completely re-designed to use the RDMA NIC interface and highly optimized to the performance characteristics of specific hardware versions.

Unfortunately, customizing RDMA systems to the hardware requires enormous engineering effort (and reaps few benefits [28, 57]). The downside for hardware designers is significant too: as more applications rely on accelerators, it becomes difficult for the hardware to evolve without impacting an increasingly large number of applications.

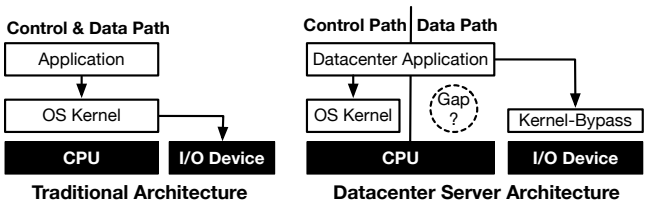

Figure 1. *Comparison of traditional server architecture and kernel-bypass server architecture.* Kernel-bypass accelerators let applications safely access I/O devices but do not replace the bypassed OS functionality. Importantly, there is no longer a high-level, device-agnostic I/O abstraction.

This paper argues for an evolution of the datacenter operating system to provide a high-level kernel-bypass I/O abstraction. Just because kernel-bypass accelerators eliminate the OS kernel does not mean that application programmers must do without the benefits of an OS. We discuss how kernel-bypass devices have changed the datacenter and how operating systems should change as well. We propose a new OS architecture, the *Demikernel*, and discuss design challenges for the Demikernel and future datacenter OSes.

2 Kernel-Bypass Accelerators in the Datacenter

Today's datacenter kernel-bypass accelerators comprise a confusing combination of technologies, protocols, and specifications, implemented by a variety of I/O devices. There is no unifying interface or set of features, other than reducing application overhead by bypassing the OS kernel on the I/O path. This section provides background on available devices and their offered features, as well as insight into the difficulties facing application programmers when using these devices.

Some kernel-bypass accelerators only implement kernel-bypass, while others offer more, ranging from additional OS functionality to arbitrary application-function offload. Table 1 gives examples in each category. Intel's Data-Plane Development Kit [15] (DPDK) or Storage-Plane Development Kit [56] (SPDK) specify basic I/O device functionality for kernel-bypass. Arrakis [51], Ix [5] and related work [27, 36] use hardware virtualization to provide the same features; for example, SR-IOV [23] provides multiplexing and the IOMMU [2] provides address translation; however, these devices provide no additional OS features. To use kernel-bypass accelerators in this category, applications must supply their own I/O stack (e.g., a complete user-level TCP stack).

RDMA NICs provide a limited networking stack and support the `verbs` interface, which offers reliable communication, and the `rdmacm` interface, which almost matches the POSIX socket interface. However, to send and receive data, applications must still supply OS buffer management and flow control. Applications have to register memory before using it for I/O, and receivers must allocate enough buffers of the right size for senders. Allocating too many buffers wastes memory while allocating too few causes communication to fail.

Table 1. *Examples of kernel-bypass accelerators.* We categorize acclerators based on their offered features. Some devices (left) only offer kernel-bypass, some (middle) add a subset of OS features (e.g., RDMA provides a limited networking stack), and others (right) add more complex features (e.g., compression, encryption, offloaded application functions).

Kernel-bypass	+OS features	+other features
DPDK/SPDK	RDMA	FPGA NICs/SSD/NVMe
Arrakis/Ix		ARM SoC NICs/SSDs/NVMe

Finally, there is a new category of programmable devices, including I/O devices with FPGAs (e.g., Catapult [9], Mellanox Innova [43], NetFPGA [48]) and I/O devices with ARM Systems-on-a-chip (e.g., Broadcom NetXtreme [8], Mellanox Bluefield [42]) that offer the potential to offload arbitrary application functions. Compression, encryption and machine learning have been proposed but almost anything is possible, albeit not always practical.

Lacking any unified OS support for today's kernel-bypass accelerator technologies, application programmers must deeply understand device features and OS internals to achieve kernel bypass. They must implement missing OS functionality that their application might need, and, worse, they must implement different OS functionality for each specific device and update any changes as devices evolve.

3 Evolving the Datacenter OS for Kernel Bypass

The role of the operating system in a kernel-bypass framework is simple: offer the benefits of the OS kernel with minimal overhead. We detail the requirements to fulfill this role.

3.1 Optimize for User-Level I/O Processing

Kernel-bypass requires all OS functionality on the I/O path be implemented in a user-level library. However, recycling existing kernel-level I/O stacks into a user-level library is both inefficient and insufficient to meet the needs of kernel-bypass applications. For example, the traditional OS I/O stack includes many mechanisms for sharing and multiplexing that add needless overhead, especially when the library OS shares the application's address space.

Further, OS kernels lack mechanisms of value to kernel-bypass applications. For example, kernel-bypass devices offer user-space address translation but expect applications to help wtih memory management by explicitly registering memory with each device. Thus, a kernel-bypass OS can simplify applications by transparently making all application memory available to I/O devices. As an added benefit, this design lets the OS make performance trade-offs without modifying every application. For example, there is a trade-off between memory usage and device registration because any registered memory

must be pinned and cannot be re-allocated or swapped. This transparent memory registration is not available in existing OS stacks. The combination of unnecessary and missing features suggests the need for a completely re-designed user-level OS I/O stack.

3.2 Offer an Efficient I/O-Processing Abstraction

Our current I/O abstraction is inherited from a time when I/O devices were slow and applications spent much of their time waiting for I/O to complete. In contrast, datacenter I/O devices now deliver I/O requests faster than applications can process them, so applications must be highly optimized to process I/O as fast as possible. For example, Redis spends about 2µs on each read request [51]; to keep up with increasing NIC speeds, it no longer afford any added latency from the OS kernel. For these applications, the kernel's I/O abstraction is as much a barrier to performance as the kernel itself.

While user-level libraries that preserve the POSIX API (e.g., mTCP [25], F-stack[19]) are easy to use with existing applications, the legacy I/O abstraction imposes too much overhead. The POSIX abstraction requires applications copy from kernel buffers into application buffers. This copy is both inefficient (copying a 4k page takes 1µs on a 4Ghz CPU, adding 50% overhead to Redis), and unnecessary (since the data is already in the user-level address space). Second, UNIX pipes force applications to operate on streams of data; however, applications like Redis operate on atomic units of data. Redis can only process a read operation after the entire request has arrived; by the time Redis has inspected a pipe and found that its read operation is incomplete, it could have processed a request that was ready. The incompatibility of the existing POSIX interface with high-performance I/O processing calls for a new abstraction.

3.3 Implement Differing OS Functionality

Library operating systems [17, 38] already offer an architectural solution for a kernel-bypass OS; however, existing ones expect uniform OS kernel interfaces and hardware device features. For example, Arrakis [51] and Ix [5] only work with I/O devices that support standard hardware virtualization features, and even flexible library OS architectures, like the Exokernel [17] and others [38], expect a uniform kernel and device I/O abstraction that kernel-bypass devices do not offer.

A flexible library OS architecture is not only important to hide differences between kernel-bypass devices from applications but gives hardware designers the ability to move features between the hardware (the kernel-bypass accelerator) and software (the library OS). There are trade-offs to implementing functionality on the device (e.g., access to main memory is slower than from the CPU), so there will always be limitations to what OS features are practical and efficient to implement on each device. A library OS offers an ideal place to implement on the CPU those features that cannot be implemented on the device. The need for varying amounts of OS functionality is completely different from library OS architectures of the past and argues for a new design.

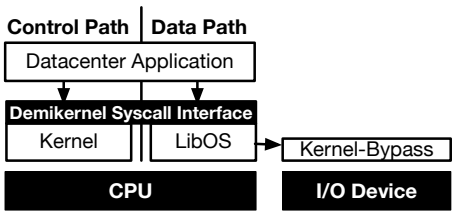

Figure 2. *The Demikernel architecture.* The protected OS kernel implements control-path operations that the application does not access on every I/O (e.g., device set up, coarse-grained resource management). Each libOS is customized to an kernel-bypass accelerator, which implements some OS functionality while the libOS provides the rest.

4 The Demikernel

The Demikernel is a new OS architecture for kernel-bypass accelerators. We describe it as a Demikernel because no single component in I/O-accelerated datacenter servers implements all OS kernel functionality. This section presents the Demikernel architecture and new kernel-bypass accelerator abstraction.

4.1 Demikernel Architecture

As shown in Figure 2, the Demikernel architecture splits traditional OS functionality into a kernel, which handles control path operations, and a library OS (libOS), which handles data-path I/O operations. Every libOS implements the Demikernel I/O abstraction across kernel-bypass accelerators and a user-level runtime library.

The *control path* includes all OS functionality not used on every I/O or used infrequently. This functionality includes allocating virtualized kernel-bypass accelerators to applications, opening network connections, creating and opening files, etc. The control path is not performance-critical, so we forsee that it will continue to be provided by legacy OS kernels (e.g., Linux), especially because it requires complex functionality that is not easily re-implemented.

The *data path* covers all OS functionality that handles I/O, including reading and writing to storage devices, networking devices and remote memory. Operations on other types of I/O that may be available in the future (e.g., writing to disaggregated memory) would also need to be included. The data path is both performance-critical and CPU-intensive because it must execute on every I/O operation. As a result, the Demikernel architecture splits this functionality across libOSes and kernel-bypass accelerators.

4.2 Demikernel I/O Queues

A key component of the Demikernel architecture is its device-independent, high-level abstraction for kernel-bypass I/O. The Demikernel abstracts kernel-bypass I/O devices as *I/O queues*, which, unlike POSIX pipes, have an atomic data unit.

Applications push data for I/O as an atomic unit into the queue and only pop data out of the queue when an entire data unit has arrived and the application can process it to completion.

For performance, the Demikernel library OSes try to preserve the application data unit on the device if possible. Demikernel queues are not bound by hardware limitations (e.g., limited capacity queues, fixed packet sizes) and are uniform between different devices. Since I/O devices commonly use hardware queues to interact, we found that the queue abstraction is general enough to apply to a wide range of I/O accelerators. The queue abstraction also lets applications express application-specific functions that can be offloaded to the I/O device through queue filter and map functions.

4.3 Demikernel System Call Interface

Applications interact with Demikernel queues through the *Demikernel system call interface*. System calls that give application access to I/O devices, which would previously return a file descriptor, now return a queue descriptor. To reduce application changes, the Demikernel syscall interface leaves most control-path calls in place while introducing a new set of data-path calls. Figure 3 summarizes the system calls.

Control-path calls also include queue creation and modification, e.g., filtering a network I/O queue for a packet type or encrypting data in a storage I/O queue before writing to disk. These calls are considered part of the control path; though a queue filter executes on every I/O, the application calls the OS only once to set up the filter.

The most important data path operations are `push` and `pop` for sending and receiving data, respectively. These system calls take a scatter-gather array of memory pointers, which form an atomic *queue element*. A scatter-gather array pushed into a Demikernel queue always pops out as a single element. This feature provides two benefits: (1) it gives an kernel-bypass accelerator information about the granularity at which to compute (e.g., the encryption or compression unit), and (2) it ensures that applications process requests only when the entire queue element is available and can be processed to completion.

Both `push` and `pop` are non-blocking. If either operation cannot immediately complete, it returns a *qtoken*. Applications use the qtoken to fetch the completion when it is ready via `wait_*` system calls. The `wait` call blocks on a single queue token, `wait_any` provides functionality similar to `select` or `epoll`, and `wait_all` blocks until all operations complete. Section 4.4 discusses how applications process I/O using the `wait_*` calls.

The `merge` system call (line 14, Figure 3) returns a new queue that merges two queues. A `pop` from either queue results in a `pop` from the merged queue and a `push` to the merged queue results in a `push` to both queues.

The `filter` system call (line 15, Figure 3) returns a new queue with only the filtered elements from the original queue. A `pop` from the original queue results in a `pop` from the new queue only if the filter function returns true; a `push` into the new queue results in a `push` to the original queue only if the filter function is met. We currently do not restrict filter functions, but we forsee using a verified framework like Berkeley Packet Filters [40] or Floem [52]. Library OSes always implement filters directly on supported devices but default to using the CPU if necessary. Filters are useful beyond reducing CPU load: for example, they can improve cache utilization by steering I/O to CPUs based on application-specific parameters (e.g., keys in a key-value store) [32].

The `sort` system call returns a new queue with elements reordered from the original one. A `push` to the original queue automatically places the element in the sorted queue in priority order, causing a `pop` from the sorted queue to return the element with the highest priority from the original queue. Sorted queues are useful for implementing application-specific priority definitions, which the library OS strives to offload to the I/O device when possible.

The `map` system call returns a queue that applies the map function to every queue element. Applications can combine queues to create complex I/O processing pipelines, which can then be offloaded to an kernel-bypass accelerator. For example, a series of filter and map queues could be implemented on today's programmable NICs using P4 [7] or Floem [52].

4.4 Event and Thread Scheduling

Demikernel does not prescribe a particular I/O handling model (e.g., asynchronous, co-routines, polling); however, we envision Demikernel libOSes being tightly integrated with existing scheduling libraries. To support this, the Demikernel interface provides a low-level, improved `epoll` interface, which we call `wait_*` (lines 5-7 in Figure 3), that works with qtokens returned by non-blocking queue operations.

Because queues have granularity, each qtoken is unique to a single queue operation. As a result, different application threads can wait on different tokens, rather than all waiting on the same file descriptor. This abstraction solves two major issues with POSIX `epoll`: (1) `wait` directly returns the data from the operation so the application can process the returned data without making another system call, and (2) wait wakes exactly one thread on each `pop` completion, so there are never wasted wake ups for threads with no data to processes.

Applications can easily replace an application-level `epoll` loop with a call to `wait_any`. In the future, we plan to implement a libevent [39]-based Demikernel OS, which would enable applications, like memcached [20], to achieve the benefits of kernel-bypass transparently.

4.5 Memory Management

The Demikernel interface provides semi-transparent, zero-copy I/O for applications in the form of transparent memory registration and free-protection for in-use memory buffers. Demikernel libOSes employ a memory manager, similar

```
// control path network calls
int qd = socket(...)
int err = listen(int qd, ...);
int err = bind(int qd, ...)
int qd = accept(int qd, ...)
int err = connect(int qd, ...)
int err = close(int qd);
// control path file calls
int qd = open(...);
int qd = creat(...);
```

```
// control path queue calls
int qd = queue();
int qd = merge(int qd1, int qd2);
int qd = filter(int qd, bool (*filter)(sgarray &sga));
int qd = sort(int qd1, bool (*sort)(sgarray &sga1, sgarray &sga1));
int qd = map(int qd1, void (*map)(sgarray &sga));
int status = qconnect(int qdin, int qdout);
```

```
// data path queue calls
qtoken qt = push(int qd, const sgarray &sga);
qtoken qt = pop(int qd, sgarray &sga);
ssize_t ret = wait(qtoken qt, sgarray &sga);
ssize_t ret = wait_any(qtoken *qts, size_t num_qts, qevent *qevs, size_t num_qevs, int timeout);
ssize_t ret = wait_all(qtoken *qts, size_t num_qts, qevent *qevs, size_t num_qevs, int timeout);
// identical to a push, followed by a wait on the returned qtoken
ssize_t ret = blocking_push(int qd, sgarray &sga);
// identical to a pop, followed by a wait on the returned qtoken
ssize_t ret = blocking_pop(int qd, sgarray &sga);
```

Figure 3. *Demikernel System Call Interface.* The control path interface is preserved from POSIX; however, network and storage system calls now return and take queue descriptors (qd), which are `int`s similar to file descriptors. The `...` represents unchanged arguments. We do not include all control path calls; additional ones may be added for new device features (e.g., compression). The data path interface uses queue operators in lieu of read and write operations. There are additional system calls that manipulate queues (e.g., filter, merge). Note that these calls may be implemented by the libOS to run on the CPU or be offloaded to the kernel-bypass accelerator.

to existing memory allocation libraries (e.g., jemalloc [24], Hoard [6]) but customized to support these features.

Kernel-bypass accelerators with IOMMUs typically require memory registration to perform on-device address translation. The Demikernel interface eliminates this registration from applications; instead, Demikernel libOSes register memory regions with kernel-bypass accelerators and then allocate application memory from those regions.

Zero-copy I/O requires applications to coordinate shared memory access with the I/O devices; that is, the application cannot write or free any memory currently being accessed by an I/O device. Similarly, when a device finishes processing an I/O request, it needs to notify the application that it can modify or free the buffer. Such coordination between device and application must often be done across threads or components, making it difficult for applications to accomplish on their own.

To minimize this coordination, the Demikernel interface provides free-protection for I/O memory buffers. Applications can free buffers while they are in use by a device, but the libOS will not deallocate the buffer until the device completes its I/O. The Demikernel interface does not offer write-protection for I/O buffers, which would be too expensive. Thus, applications must still wait until their I/O completes (i.e., `push` returns or `wait` on a qtoken completes) to modify buffers as they do for traditional zero-copy I/O.

We believe this trade-off is reasonable for datacenter applications because they do not often perform in-place updates, and it is easy to make changes where they do. For example, Redis allocates a new value buffer for each `put` request and changes the pointer in its data structures to the new buffer.

Our choice to integrate the memory manager in the libOS with the kernel-bypass accelerator means that applications cannot use an application-specific memory manager. However, we believe that the benefit of avoiding explicit memory registration and coordination outweighs the applications' need for custom memory allocation. We envision application-specific management can be addressed by adapting existing memory allocators for the Demikernel as part of a new libOS.

5 Future Work

While the Demikernel specifies a new OS architecture, the design of its library OSes leaves much room for future work.

5.1 Library OS Design

Each Demikernel library OSes supports a specific accelerator type (e.g., RDMA, DPDK). To do so, it implements the OS functionality missing from the specific device. For example, while DPDK requires an entire networking stack, RDMA requires a transport implementation atop the RDMA verbs interface. Design decisions are specific to each device type as well; for example, which networking stack to use for DPDK, or whether to use one- or two-sided operations for RDMA communication. Ideally, the library OSes should be built in a modular fashion and share as much code as possible, but it remains unclear how to do so when there are so many varied kernel-bypass accelerators.

5.2 Network Protocols

Network I/O poses a challenge for library OSes. While some accelerators work with all network protocols (e.g., DPDK can support any networking stack), others require both ends of the connection to support a particular protocol (e.g., RDMA

NICs require that both ends communicate via Infiniband or RoCE [4]). Further, to support Demikernel queues, Demikernel libOSes need a unit for breaking up I/O. The libOS could insert the needed framing itself (e.g., atop a TCP stream); however, the other end must be able to correctly parse the framing and recreate the scatter-gather array. Alternatively, the libOS could use framing available in an existing protocol (e.g., HTTPS, REST), but this approach trades-off limit libOS generality.

5.3 File Systems and Storage

A similar challenge for libraryOSes appears in efficient access to storage. If Demikernel library OSes used a custom disk layout for performance, any application would have to find a compatible libOS to read stored data. Existing disk layouts (e.g., ext4) may impose unnecessary overhead since each Demikernel libOS supports only a single application, which may not require an entire UNIX file system. Future work could include design of an accelerator-specific storage layout.

6 Related Work

The Demikernel takes inspiration from the significant past research into operating systems and kernel-bypass accelerators.

Operating Systems. The library operating system approach has been used previously to improve performance while maintaining flexibility. Previous library operating systems [17, 38, 53] were customized to applications in order to provide high performance and flexibility. Recent work, such as Arrakis [51] and IX [5], focus on providing low-latency access to I/O devices; both split functionality between a control plane and a data plane. Demikernel also splits functionality in this way; however, it targets a range of kernel-bypass accelerators, not hardware virtualization devices. Additionally, Demikernel provides a new high-level kernel-bypass interface that Ix and Arrakis do not.

User-level OS extensions [18, 61] let applications customize parts of the OS for their needs. User-space file systems and networking stacks have also been heavily researched. Device drivers, whether in the OS kernel [13, 58] or at user level [37], hide differences between hardware interfaces; however, they do not implement OS functionality. For example, Mellanox and Intel provide DPDK device drivers for their respective devices, but these drivers implement no OS functionality, not even a networking stack.

I/O Accelerated Systems. Many user-level networking stacks replace missing functionality in DPDK devices and maintain the POSIX interface. We explored mTCP [25] but found it to be too expensive; for example, its latency was higher than the Linux kernel's. Other options [3, 19, 50, 55] suffer from similar inefficiencies imposed by the POSIX interface. Netmap [54] and Stackmap [62] both offer user-level interfaces to NICs, but they are much lower level than the one proposed in this paper.

ReFlex [35] and PASTE [22] provide fast remote access to SSDs and NVMMEs, respectively, by optimizing processing between the network and storage I/O. The Demikernel provides a more general interface to make these applications easier to build.

Past and recent efforts have explored moving functionality between the CPU, NIC and network. Moving TCP is an old idea [14, 46] that has become newly popular. Narayan, et al. [47] make a case for leaving TCP on the CPU but moving congestion control off of the data path, while TAS [33] proposes moving parts of the protocol to a dedicated CPU. Mittal, et al. [45] explore moving reliable delivery from the network into the NIC, although a CPU-based option would also be possible. DPI [1] proposes an interface similar to the Demikernel syscall interface but uses flows instead of queues and considers network I/O but not storage.

I/O Accelerated Applications. Many applications have been customized to leverage low-latency network access. Systems such as FaRM [16], FASST [30], and more [11, 29, 34, 44, 59, 60] use RDMA for low-latency access to remote memory. In contrast, the Demikernel targets applications that want the benefits of kernel-bypass and are willing to sacrifice access to hardware-specific features for portability and future-proofing.

7 Conclusion

Today's widely-available kernel-bypass accelerators let applications bypass the operating systems stack on the I/O path. Unfortunately, they do not replace the OS kernel's functionality, leaving a crucial gap in the stack: no high-level I/O abstraction. To fill this gap, we proposed the Demikernel, a new OS architecture for kernel-bypass datacenter servers. The Demikernel defines a new kernel-bypass I/O abstraction and customizes library OSes, not applications, to specific kernel-bypass accelerators. More research into the challenges of designing these library OSes remains for future work.

Acknowledgments

We would like to thank all who provided valuable feedback and help, including Dan Ports, Anna Kornfeld Simpson, Adriana Szekeres, Amar Phanishayee, Adam Belay, Sandy Kaplan, Natacha Crooks and the anonymous reviewers on the HotOS PC.

References

[1] G. Alonso, C. Binnig, I. Pandis, K. Salem, J. Skrzypczak, R. Stutsman, L. Thostrup, T. Wang, Z. Wang, and T. Ziegler. DPI: the data processing interface for modern networks. In *Proc. of CIDR*, 2019.

[2] AMD. *AMD I/O Virtualization Technology (IOMMU) Specification*, December 2016. https://support.amd.com/TechDocs/48882_IOMMU.pdf.

[3] Ans(accelerated network stack) on dpdk, dpdk native tcp/ip stack. https://github.com/ansyun/dpdk-ans.

[4] I. T. Association. Infiniband architectural specification, Sept 2014. https://www.infinibandta.org/ibta-specification/.

[5] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: a protected dataplane operating system for high

throughput and low latency. In *Proc. of OSDI*, 2014.

[6] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *ACM SIGARCH Computer Architecture News*, volume 28, pages 117–128. ACM, 2000.

[7] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, et al. P4: Programming protocol-independent packet processors. *Proc. of SIGCOMM*, 2014.

[8] Broadcom. High-performance datacenter SoC with integrated NetXtreme ethernet controller. https://www.broadcom.com/products/ethernet-connectivity/controllers/bcm58800.

[9] Project Catapult. https://www.microsoft.com/en-us/research/project/project-catapult/.

[10] Cavium. LiquidIO 2 SmarNICs. https://www.cavium.com/liquidio-II-server-adapters.html.

[11] H. Chen, R. Chen, X. Wei, J. Shi, Y. Chen, Z. Wang, B. Zang, and H. Guan. Fast in-memory transaction processing using rdma and htm. *ACM Transactions on Computer Systems*, 35(1):3, 2017.

[12] R. Consortium. A rdma protocol specification, October 2002. http://rdmaconsortium.org/.

[13] J. Corbet, A. Rubini, and G. Kroah-Hartman. *Linux Device Drivers: Where the Kernel Meets the Hardware*. " O'Reilly Media, Inc.", 2005.

[14] A. Currid. Tcp offload to the rescue. *ACM Queue*, 2004.

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

[16] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson. Farm: Fast remote memory. In *Proc. of NSDI*, pages 401–414, 2014.

[17] D. R. Engler, M. F. Kaashoek, et al. Exokernel: An operating system architecture for application-level resource management. In *Proc. of SOSP*, 1995.

[18] J. Evans. A scalable concurrent malloc (3) implementation for FreeBSD. In *Proc. of the BSDCan Conference*, 2006.

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

[20] B. Fitzpatrick. Distributed caching with memcached. *Linux Journal*, 2004.

[21] H. Gilmore. The Cloud as a Tectonic Shift in IT: The Death of Operating Systems (as We Know Them). Cloudbees, July 2012. https://www.cloudbees.com/blog/cloud-tectonic-shift-it-death-operating-systems-we-know-them.

[22] M. Honda, G. Lettieri, L. Eggert, and D. Santry. PASTE: a network programming interface for non-volatile main memory. In *Proc. of NSDI*, 2018.

[23] T. Hudek. Overview of single root i/o virtualization (sr-iov). Technical report, Microsoft, April 2017. https://docs.microsoft.com/en-us/windows-hardware/drivers/network/overview-of-single-root-i-o-virtualization--sr-iov-.

[24] http://jemalloc.net/.

[25] E. Jeong, S. Woo, M. A. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mTCP: a highly scalable user-level tcp stack for multicore systems. In *Proc. of NSDI*, 2014.

[26] M. Kabay and G. Merrill. Is the operating system dead? NetworkWorld, July 2011. https://www.networkworld.com/article/2178825/wireless/is-the-operating-system-dead-.html.

[27] K. Kaffes, T. Chong, J. T. Humphries, A. Belay, D. Mazières, and C. Kozyrakis. Shinjuku: Preemptive scheduling for μsecond-scale tail latency. In *Proc. of NSDI*, pages 345–360, 2019.

[28] A. Kalia, M. Kaminsky, and D. Andersen. Datacenter RPCs can be general and fast. In *nsdi*, 2019.

[29] A. Kalia, M. Kaminsky, and D. G. Andersen. Using rdma efficiently for key-value services. *ACM SIGCOMM Computer Communication Review*, 44(4):295–306, 2015.

[30] A. Kalia, M. Kaminsky, and D. G. Andersen. Fasst: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram rpcs. In *Proc. of OSDI*, 2016.

[31] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks. Profiling a warehouse-scale computer. In *ACM SIGARCH Computer Architecture News*. ACM, 2015.

[32] A. Kaufmann, S. Peter, N. K. Sharma, T. Anderson, and A. Krishnamurthy. High performance packet processing with flexnic. In *Proc. of ASPLOS*, 2016.

[33] A. Kaufmann, T. Stamler, S. Peter, N. K. Sharma, T. Anderson, and A. Krishnamurthy. TAS: TCP acceleration as a service. In *Proc. of EuroSys*, 2019.

[34] D. Kim, A. Memaripour, A. Badam, Y. Zhu, H. H. Liu, J. Padhye, S. Raindel, S. Swanson, V. Sekar, and S. Seshan. Hyperloop: group-based nic-offloading to accelerate replicated transactions in multi-tenant storage systems. In *Proc. of SIGCOMM*, pages 297–312. ACM, 2018.

[35] A. Klimovic, H. Litz, and C. Kozyrakis. Reflex: Remote flash ≈ local flash. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 345–359, New York, NY, USA, 2017. ACM.

[36] Y. Kwon, H. Fingler, T. Hunt, S. Peter, E. Witchel, and T. Anderson. Strata: A cross media file system. In *Proc. of SOSP*, pages 460–477. ACM, 2017.

[37] B. Leslie, P. Chubb, N. Fitzroy-Dale, S. Götz, C. Gray, L. Macpherson, D. Potts, Y.-T. Shen, K. Elphinstone, and G. Heiser. User-level device drivers: Achieved performance. *Journal of Computer Science and Technology*, 20(5):654–664, 2005.

[38] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Select Areas in Communications*, 14(7):1280–1297, 9 1996.

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

[40] S. McCanne and V. Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *USENIX winter*, volume 46, 1993.

[41] S. McCarty. The operating system is dead. Long live the operating system? InfoWorld, May 2018. https://www.infoworld.com/article/3269605/operating-systems/the-operating-system-is-dead-long-live-the-operating-system.html.

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

[43] Mellanox. Innova Flex Smart NIC. http://www.mellanox.com/page/products_dyn?product_family=276&mtag=programmable_adapter_cards_innova2flex.

[44] C. Mitchell, Y. Geng, and J. Li. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *Proc. of USENIX ATC*, 2013.

[45] R. Mittal, A. Shpiner, A. Panda, E. Zahavi, A. Krishnamurthy, S. Ratnasamy, and S. Shenker. Revisiting network support for rdma. In *Proc. of SIGCOMM*, 2018.

[46] J. C. Mogul. Tcp offload is a dumb idea whose time has come. In *Proc. of HotNets*, pages 25–30, 2003.

[47] A. Narayan, F. Cangialosi, P. Goyal, S. Narayana, M. Alizadeh, and H. Balakrishnan. The case for moving congestion control out of the datapath. In *Proc. of HotNets*. ACM, 2017.

[48] NetFPGA: A line-rate, flexible, and open platform for research, and classroom experimentation. https://netfpga.org/site/#/.

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

[50] Openonload. https://www.openonload.org/.

[51] S. Peter, J. Li, I. Zhang, D. R. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The operating system is the control plane. *ACM Transactions on Computer Systems*, 2016.

[52] P. M. Phothilimthana, M. Liu, A. Kaufmann, S. Peter, R. Bodik, and T. Anderson. Floem: a programming system for NIC-accelerated network applications. In *Proc. of OSDI*, pages 663–679, 2018.

[53] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt. Rethinking the library os from the top down. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 291–304, New York, NY, USA, 2011. ACM.

[54] L. Rizzo. Netmap: a novel framework for fast packet i/o. In *Proc. of USENIX Security*, pages 101–112, 2012.

[55] SolarFlare. https://www.solarflare.com/ultra-low-latency.

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

[57] M. Su, M. Zhang, K. Chen, Z. Guo, and Y. Wu. Rfp: When rpc is faster than server-bypass with rdma. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 1–15. ACM, 2017.

[58] M. M. Swift, S. Martin, H. M. Levy, and S. J. Eggers. Nooks: An architecture for reliable device drivers. In *Proc. of EuroSys*, 2002.

[59] K. Taranov, G. Alonso, and T. Hoefler. Fast and strongly-consistent per-item resilience in key-value stores. In *Proceedings of the Thirteenth EuroSys Conference*, page 39. ACM, 2018.

[60] X. Wei, Z. Dong, R. Chen, and H. Chen. Deconstructing rdma-enabled distributed transactions: Hybrid is better! In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, Carlsbad, CA, 2018. USENIX Association.

[61] B. B. Welch and J. K. Ousterhout. Pseudo devices: User-level extensions to the sprite file system. Technical report, UC Berkeley, 1988.

[62] K. Yasukata, M. Honda, D. Santry, and L. Eggert. Stackmap: Low-latency networking with the os stack and dedicated nics. In *Proc. of USENIX ATC*, pages 43–56, 2016.