I/O Is Faster Than the CPU – Let's Partition Resources and Eliminate (Most) OS Abstractions

Pekka Enberg
University of Helsinki

Ashwin Rao
University of Helsinki

Sasu Tarkoma
University of Helsinki

Abstract

I/O is getting faster in servers that have fast programmable NICs and non-volatile main memory operating close to the speed of DRAM, but single-threaded CPU speeds have stagnated. Applications cannot take advantage of modern hardware capabilities when using interfaces built around abstractions that assume I/O to be slow. We therefore propose a structure for an OS called *parakernel*, which eliminates most OS abstractions and provides interfaces for applications to leverage the full potential of the underlying hardware. The parakernel facilitates application-level parallelism by securely partitioning the resources and multiplexing only those resources that are not partitioned.

ACM Reference Format:
Pekka Enberg, Ashwin Rao, and Sasu Tarkoma. 2019. I/O Is Faster Than the CPU – Let's Partition Resources and Eliminate (Most) OS Abstractions. In *Workshop on Hot Topics in Operating Systems (HotOS '19), May 13–15, 2019, Bertinoro, Italy.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3317550.3321426

1 Introduction

Many OS abstractions being used today were designed when I/O speeds were significantly slower than CPU speeds. OSes could, therefore, get away with abstracting the underlying hardware to make application programming easier without paying attention to the inefficiency of the interfaces [13]. However, the assumption that I/O is significantly slower than CPU is no longer valid given that 400 Gbps Ethernet is in the horizon and non-volatile memory (NVM) is approaching the speeds of DRAM [18], while the performance of single-threaded CPUs has stagnated [16, 43].

More than two decades ago Engler *et al.* [13] argued that OS abstractions and their corresponding interfaces incur a substantial performance cost. The impact of these inefficiencies is now amplified because I/O speeds are becoming faster, surpassing CPU speeds in some cases. For example, the traditional in-kernel network stacks perform too much work per packet to keep up with high packet rates of modern NICs [17]. Furthermore, OSes typically implement POSIX sockets API as a system call per socket operation, which has high overheads from context switching and CPU cache pollution. These inefficiencies have led to an influx of a variety of approaches to optimize the network stack including completely bypassing the kernel networking stack [29, 33, 42]. Similarly, there is an increasing need to bypass the kernel for I/O operations to leverage fast and parallel NVM Express (NVMe) SSDs [24, 47]. The demand to bypass the kernel for I/O operations force us to rethink OS abstractions and interfaces.

The stagnation in the performance of single-threaded CPUs has made multicore systems a norm. The performance benefits of multiple cores can be extracted only when server applications are parallelized. This need for parallelism has

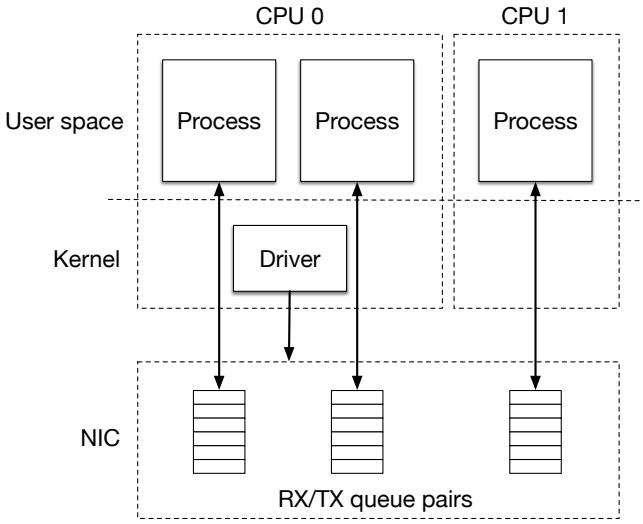

Figure 1. Parakernel architecture. *The parakernel partitions hardware resources between processes and minimizes kernel participation in data plane operations. In the case of a NIC, an in-kernel driver maps RX/TX queue pairs to process address space and configures the NIC to steer process flows to the appropriate queues. The parakernel securely multiplexes resources that are not partitioned, and it has a small trusted computing base (TCB) by eliminating most OS abstractions including obsolete POSIX abstractions.*

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

ACM ISBN 978-1-4503-6727-1/19/05…$15.00
https://doi.org/10.1145/3317550.3321426

fueled the popularity of thread-per-core application architectures and asynchronous OS interfaces. Although asynchronous OS interfaces have largely replaced their blocking counterparts, the internal synchronization in shared-memory kernels can cause asynchronous interfaces to block intermittently. This affects the response time of highly parallel server applications [11]. Parallelism requires application threads and their corresponding kernel threads to run asynchronously, and this can be achieved only by partitioning the underlying hardware resources.

Server applications also need to be isolated from each other to contain misbehaving applications. Isolation is clearly at odds with having a large trusted computing base (TCB) found in shared memory monolithic kernels [5] and has fuelled the demand for virtualization techniques such as Kernel-based Virtual Machine (KVM) on Linux. These virtualization techniques incur performance and energy overheads. Furthermore, popular UNIX-based server OSes are inherently vulnerable to misbehaving applications because they are written in C which is not type-safe [3].

We believe that partitioning of resources is critical, and there is a need to provide only a small number of OS interfaces written in a type-safe language that allows applications to maximize the performance of the underlying hardware. We argue for a new OS structure which we call a *parakernel*. As shown in Figure 1, the parakernel minimizes participation for the data plane activities of the processes after hardware resources have been securely partitioned and allocated to them.

2 Motivation: Many OS abstractions are problematic

In the time since Engler *et al.* [13] highlighted the key shortcomings of OS abstractions, the hardware landscape has changed completely. Not only are multicore systems a norm, but I/O speeds are increasing while single-threaded CPU speeds have largely stagnated, which puts pressure on the OS to keep up with I/O. For example, a 40 GbE NIC can receive a cache line sized packet every 5 ns, but the last level cache (LLC) access latency is already up to 15 ns, which means a single LLC access can already prevent the OS from keeping up with arriving packets [22]. These advances expose the shortcomings of OS abstractions which were designed when I/O speeds were considerably slower than CPU speeds.

Limited I/O performance. OS abstractions simplify programming by decoupling applications from the underlying hardware. This decoupling incurs large overheads in performance, latency, and energy because the OS needs to maintain a shadow copy of hardware state and provide hardware-independent interfaces. The emergence of byte-addressable persistent non-volatile memory that is getting as fast as DRAM is also shaking up the memory hierarchy and challenging OS storage abstractions [2].

In networking, sockets are the OS abstraction that allows applications to exchange data. Sockets require OSes to maintain a shadow copy of the NIC data structures along with data structures to implement protocol state machines. At the same time, applications invoke expensive system calls to send and receive data using sockets. Traditional in-kernel network stacks built around sockets also perform too much work per-packet to keep up with the high-speed NICs [17, 42]. Approaches to bypass the in-kernel network stack are not new [38], and the current trend towards Smart NICs [14] and user space networking stacks [33] exemplify the benefits of removing the legacy abstractions from the OS.

In storage, asynchronous I/O (AIO) is an abstraction that allows applications to submit I/O requests and keep performing other tasks until the OS notifies that the I/O operation is complete. With the increase in I/O speeds, the time required to accept an I/O request and notify its completion is comparable to the time required to complete the I/O task. For instance, NVMe SSDs perform I/O faster than the OS can accept new I/O requests and notify their completion. Furthermore, the current POSIX AIO implementations in Linux are *ugly and adhoc* [48], and they have limited support from file systems [8, 25]. This has lead to new AIO interfaces that eliminate system calls and leverage polling. However, polling for completion is not suitable for large I/O transfers [49]. Also, there is ongoing work in Linux to replace the POSIX AIO interface with user space ring buffers to provide an efficient and easy-to-use asynchronous I/O interface [10]. However, implementing such an interface while maintaining some of the more arcane POSIX semantics is complicated [9].

Limited application-level parallelism. Applications are forced to use kernel threads for concurrency because synchronous interfaces, such as the `read` system call, block the caller until the I/O operation is complete. However, having a large number of threads incur high overheads due to context switches. At the same time, the threads cannot run independently when they are using shared resources. Applications are therefore increasingly designed to use asynchronous interfaces along with a thread-per-core approach to increase parallelism.

In the thread-per-core approach, the number of application threads is equal to the number of available cores, and each thread is expected to run independently on the core assigned to it. This approach partitions and pins the available CPU cores among the application threads. Applications also need to configure interrupt affinity properly, and mistakes in the configuration limit application-level parallelism [31]. This approach shows that partitioning of resources is essential for parallelism because it allows threads to run independently.

Large attack surface. In-kernel OS abstractions run in privileged CPU mode where they have unrestricted access. The more complex the OS abstractions are, the larger the attack

surface of the OS becomes. A large attack surface is a serious concern in monolithic kernels because applications share the same OS kernel and there is a large set of system call and device drivers bugs to exploit [5].

High-level type-safe programming languages have the potential to eliminate many classes of programming bugs that could be exploited [3, 30]. Similarly, formal verification of OS kernels provides a fundamentally safe trusted computing base (TCB) [27, 39]. Although this approach can reduce bugs in the implementation of OS abstractions, they do not address the root of the problem: a large TCB. It is critical that the TCB be as small as possible and that the interfaces can be formally verified. Eliminating unused and inefficient OS abstraction reduces the attack surface and thus helps improving safety.

Lack of predictable tail latency and energy-efficiency. Energy-efficiency and low latency are critical because servers are being deployed at the network edge on battery-powered mobile devices to offer latency-critical services [7]. However, OS abstractions are often designed for throughput and not for energy-efficiency and low latency. For example, to satisfy a single network request, the application needs to call the `epoll_wait`, `recvmsg`, and `sendmsg` system calls on Linux, which has high overheads. These overheads are amplified when the kernel processes the packet on a CPU other than the one used by the application. The inefficiency of OS abstractions and interfaces is a waste of CPU cycles, which hurts system tail latency [11] and is not energy efficient.

3 Solution

In this section, we propose an OS structure aimed at facilitating application-level parallelism by securely partitioning resources and providing interfaces for a small set of abstractions. We call this structure a *parakernel*. The parakernel minimizes participation in the data plane activities of the applications after the resources have been partitioned and allocated to them. Applications have complete control over the allocated resources, and the resources that are not partitioned are multiplexed by the parakernel.

Securely partition hardware resources for parallelism. Resource sharing in multicore systems requires synchronization between CPU cores. This road-block for application-level parallelism can be mitigated by partitioning the resources between CPU cores to allow each core to run independently. The multikernel exemplifies this by partitioning the hardware resources such as DRAM between CPU cores [4]. This shared-nothing model complements the thread-per-core approach of MICA [32] and Seastar [44], where applications run a single kernel thread per core after partitioning hardware resources among the threads. Partitioning resources at the application-level require applications to discover the hardware topology, such as DRAM NUMA locality and use existing OS interfaces to allocate resources for specific CPU cores. In contrast, the lack of OS-level partitioning in shared memory systems requires various workarounds to avoid OS-level synchronizations. Secure partitioning of hardware resources is also critical for isolation, and processes sharing the same hardware device should not be able to affect the operation of other processes.

DRAM partitioning is straight-forward because the parakernel can leverage the MMU to map the part of the DRAM that belongs to a process partition to its virtual address space. In NUMA architectures, DRAM can be partitioned based on the processor socket topology to provide local access from CPU core to memory. Furthermore, with recently introduced sub-NUMA clustering, DRAM can also be partitioned within the NUMA domain to take advantage of LLC and memory controllers that are local to a cluster of CPU cores belonging to the same sub-NUMA domain.

In contrast, secure partitioning of I/O devices is more complicated because the I/O devices only have device-level protection via IOMMU, but they lack process-level protection. For example, NICs use ring buffers of DMA descriptors internally to determine where packet data is written. These DMA descriptors contain physical memory addresses, which is why it is critical that the OS ensures that the addresses are valid for secure partitioning. The parakernel requires the application to register memory buffers as a control plane operation where the OS can validate the memory addresses and program the NIC ring buffers.

Current hardware interfaces of network and storage devices support multiple queues which can work independently. For example, the Intel X710 Ethernet Controller supports up to 1536 RX/TX queue pairs [20], and the NVMe interface is specified to support up to 65535 queues [40]. Many OSes partition hardware as one queue per CPU core for improved parallelism. However, the number of queues is large enough to support an allocation of more than one queue per CPU core. The parakernel, therefore, allocates queues on a per-process basis. Specifically, the parakernel allocates an RX/TX queue pair per-process for networking. For storage, the parakernel allocates one or more NVMe queues per-process depending on the I/O parallelism required by the application.

For networking, the parakernel leverages NIC request steering to mitigate synchronization between processes. The parakernel partitions NIC RX queues to processes and leverages hardware-based steering to forward packets to the appropriate process. Leveraging packet filtering to implement efficient user space network stacks is not new, and in-kernel packet filters are widely used by applications such as packet monitors in UNIX-based systems [36, 37]. Recently, Linux introduced XDP, a high-speed networking interface which is based on the extended Berkeley Packet Filter (eBPF) [17]. XDP lets applications run eBPF programs as part of in-kernel packet processing in a VM, which enables applications to

perform up to L7 protocol (e.g., HTTP or Memcached) processing. Furthermore, some programmable NICs are able to execute eBPF programs directly on the NIC hardware [23]. This approach eliminates the OS from the network receive path as packets arrive in memory buffers, which applications can directly consume. The parakernel can leverage this hardware eBPF capability to steer packets based on the protocol information present in their headers and payloads.

Eliminate obsolete POSIX abstractions. POSIX is outdated because applications are increasingly using non-standard techniques to provide asynchronous I/O and also to talk to the kernel [1]. For instance, sockets are too heavyweight for high-speed networks [19, 42]. The parakernel provides a networking interface which steers packets to per-process memory buffers by leveraging NIC hardware capabilities and implements the network stack in user space.

The parakernel provides asynchronous interfaces and eliminates all blocking operations from OS interfaces. Blocking OS interfaces are detrimental because they require applications to leverage kernel threads for concurrency. This limits application-level parallelism because context switching between kernel threads is expensive, and the application must synchronize data access among the threads. With an asynchronous model, kernel threads are unnecessary and the parakernel replaces them with processes for parallelism and application-controlled primitives, such as coroutines or fibers, for concurrency.

Virtual memory is an abstraction that gives processes an illusion of a memory address space that is larger than DRAM. The OS is allowed to page out or swap physical memory of a process to slower secondary storage temporarily. However, many latency-sensitive applications circumvent this policy by pinning their physical memory with the `mlock` system call because page replacement has a large negative impact on tail latency.

The virtual memory abstraction also allows using memory mapping via the `mmap` system call as an I/O interface. However, `mmap` for I/O access is is effectively a blocking interface [26]. When accessed data is not in the page cache, a page fault occurs and the accessing thread blocks until the I/O operation is complete. Page cache eviction is managed by the kernel, which makes it difficult for applications to ensure that relevant data is present. It is therefore preferable that the application manages data caching because it has more information on what data to retain and what to evict. The parakernel therefore does not implement a page cache, and instead allows applications to access I/O devices directly.

The parakernel provides an interface for applications to allocate node-local anonymous memory, which is never swapped out. This interface matches what is currently done by many low-latency server applications. They first use the `mbind` system call to restrict memory allocation to the current node. Then they use the `mmap` system call to allocate physical memory. Finally, they use the `mlock` system call to prevent the OS from swapping the physical memory.

OS abstractions in the parakernel. The parakernel provides abstractions for processes, descriptor queues and memory buffers used in high-speed I/O, and for resources which are not partitioned.

Applications first request the parakernel for an isolated partition of a hardware resource via a system call and then interact directly with the hardware without interference from the OS. For example, an application requests the parakernel for a queue in a multi-queue SmartNIC via a system call. The application has full control over this NIC queue, and the parakernel leverages techniques such as eBPF to steer received packets to their appropriate queues. In the unlikely absence of traffic steering support by the NIC, the parakernel exposes a virtual queue to the applications.

Legacy hardware is full of examples of resources which cannot be partitioned. For example, a single-queue NICs require an OS abstraction that multiplexes the hardware if more than one process receives or transmits network packets. We envision that the parakernel will need to provide such abstractions only to support legacy hardware. This is because the current trends are to provide resources which can be assigned or mapped to CPU cores.

The parakernel has a small TCB because it eliminates obsolete in-kernel OS abstractions, and minimizes participation in the data plane operations of resources that can be partitioned. Its reduced attack surface improves safety and isolation, and when implemented in a high-level type-safe implementation language, we believe the parakernel can be a robust OS to deploy server applications.

4 Discussion

Why not eliminate all abstractions? Engler *et al.* [13] make a case for an exokernel that eliminates all OS abstractions and securely multiplexes the hardware resources. The parakernel does not multiplex resources that are partitioned because it gives applications full control of the hardware partition allocated to it. Partitioning of I/O resources and minimizing OS participation from the data plane is vital to allow applications to maximize the performance obtained from the underlying hardware. This also enables the parakernel to complement the current thread-per-core model where the application partition the resources to allow application threads run independently.

Do we need to maintain compatibility with POSIX? Applications are increasingly implemented on top of managed runtimes such as Node.js [15], or are using taller interfaces to workaround the limitation of POSIX [1]. It is easier to port these runtimes and interfaces because they use only a limited set of POSIX interfaces. Furthermore, I/O intensive applications are increasingly using hardware specific

development kits such as DPDK [33] and SPDK [47]. User space libraries are another approach for providing backward compatibility with POSIX interfaces such as sockets. These interfaces can be implemented as a library the application links to, instead of calling system calls per socket operation directly. This approach is already used today by some kernel-bypass solutions such as OpenOnload [46]. They implement the socket interface over their custom kernel interface, and they allow applications to use the `LD_PRELOAD` to override the default implementations [35].

Why not use kernel-bypass techniques on Linux? Applications can pin their threads to CPU cores and leverage kernel-bypass frameworks such as DPDK for networking and SPDK for storage. These frameworks offer high performance, but they also have serious shortcomings. For example, they require user-space device drivers which increases complexity, and the device assignment makes it difficult to share the device between applications. In contrast, Remote Direct Memory Access (RDMA) is a hardware-based kernel-bypass technique which allows remote access of the memory of another machine. With RDMA, it is possible to eliminate the OS from data plane operations altogether. The main downside of RDMA is that applications need to use RDMA primitives which can be difficult to use [21]. Linux also supports XDP which is a high-performance networking interface that allows applications to run packet processing eBPF code inside a sandboxed in-kernel virtual machine. Some programmable NICs are capable of executing the eBPF code [23], which allows applications to bypass the OS altogether.

These kernel bypass techniques make a strong case to eliminate obsolete POSIX abstractions. However, they fail to address the problems of a large TCB and the limited support for asynchronous operations in Linux. While these techniques can bypass the kernel for specific I/O operations, the Linux kernel continues to control the other hardware and will perform its tasks in the background. This can cause variable delays in request processing and may not be suitable for latency sensitive applications.

The parakernel is designed assuming devices such as the programmable NICs running eBPF code will soon be a part of commodity server hardware. It partitions the NIC queues to applications and minimizes participation in the data plane operations. Furthermore, the parakernel complements kernel-bypass techniques with a small TCB that eliminates obsolete POSIX abstractions.

How is the parakernel different from other kernel architectures? The primary difference of the parakernel from the past approaches is that it tries to identify the OS abstractions that must be kept when the OS minimizes participation in the data plane operations.

Microkernels implement most OS services, including device drivers, in user space on top of a minimal kernel core [12]. Applications access OS services via message passing, and microkernels can be extremely secure because they can have a small TCB [5]. However, microkernels often have the same OS abstractions as monolithic kernels, although implemented in user space. Furthermore, microkernels can limit the networking performance by decoupling the device drivers, the networking stack, and the application [19].

The splitkernel implements OS services by managing hardware resources as network-attached components [45]. The splitkernel model improves the utilization of hardware resources in a cluster of machines but can incur slowdown due to lack of locality of hardware access.

Multikernels implement OS services separately on each core, and each CPU core runs an independent instance of the OS [4]. Message passing is used for communication between CPU cores, and similarly to microkernels, many OS services such as memory management and network stack run in user space. The main difference between multikernels and parakernels is that the latter aims to eliminate all OS abstractions from the data path.

The unikernel model leverages hypervisor for isolation, and it eliminates the separation of kernel and user space at the guest-level. The application and the kernel code run in the privileged CPU mode, thus eliminating the costs of context switches [34]. The main limitation of the unikernel model is that it assumes a hypervisor, and it does not provide application multi-tenancy.

How is the parakernel different from virtualization? Hypervisors can partition a physical machine to run multiple isolated applications. The Disco hypervisor was motivated by the observation that commodity OSes did not scale well on multicore systems [6]. While the parakernel has a similar goal of improving multicore scaling, the main difference to hypervisors is that hypervisors provide a virtual machine abstraction, whereas the parakernel provides a process abstraction.

Similarly, hardware virtualization solutions also support physical resource partitioning. SR-IOV is a hardware virtualization solution which allows partitioning of physical PCI functions into independent virtual PCI functions [29]. Arrakis leverages SR-IOV to eliminate the OS from data plane operations [41]. In the Arrakis model, there is a control plane OS that manages the physical hardware and multiple data plane OSes that manage virtual devices with user space device drivers. Although the goals of Arrakis and the parakernel are similar, the main difference is that the parakernel is responsible for managing all the hardware and it also has all the device drivers. The parakernel also allows the OS to leverage NIC steering to make server-level partitioning transparent to clients.

Can the parakernel be applied for serverless computing? One emerging paradigm where the parakernel can provide tangible benefits is serverless computing. The serverless

paradigm challenges OS abstractions because even light-weight virtualization techniques such as containers are too heavy-weight [28]. The parakernel complements serverless runtimes by keeping the OS out of data plane operations. This allows light-weight abstractions to be implemented for supporting serverless applications. Furthermore, the parakernel eliminates many complex OS abstractions, which improves isolation and safety of serverless applications.

5 Conclusion

The current works to bypass the kernel clearly highlight that OS abstractions are barriers that restrict the I/O performance. These works build on a well-known fact that exporting physical resources to the applications has the potential to solve a plethora of problems plaguing current OS architectures. We present an OS structure called parakernel which partitions the resources that can be partitioned and multiplexes only those resources that are not partitioned. This allows the parakernel to have a small trusted computing base which can be implemented in a high-level language. The parakernel facilitates application level parallelism and complements the thread-per-core design of popular server applications.

A prototype parakernel written in Rust is currently under development.

Acknowledgments

We want to thank the EuroDW 2018 workshop and Gernot Heiser for helpful comments on an earlier version of this work. We would also like to thank Lars Eggert for his feedback on the first draft of this paper.

References

[1] Vaggelis Atlidakis, Jeremy Andrus, Roxana Geambasu, Dimitris Mitropoulos, and Jason Nieh. 2016. POSIX Abstractions in Modern Operating Systems: The Old, the New, and the Missing. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*. ACM, New York, NY, USA, Article 19, 17 pages. https://doi.org/10.1145/2901318.2901350

[2] Katelin Bailey, Luis Ceze, Steven D. Gribble, and Henry M. Levy. 2011. Operating System Implications of Fast, Cheap, Non-volatile Memory. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems (HotOS'13)*. USENIX Association, Berkeley, CA, USA, 2–2. http://dl.acm.org/citation.cfm?id=1991596.1991599

[3] Abhiram Balasubramanian, Marek S. Baranowski, Anton Burtsev, Aurojit Panda, Zvonimir Rakamarić, and Leonid Ryzhyk. 2017. System Programming in Rust: Beyond Safety. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS '17)*. ACM, New York, NY, USA, 156–161. https://doi.org/10.1145/3102980.3103006

[4] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. 2009. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP '09)*. ACM, New York, NY, USA, 29–44. https://doi.org/10.1145/1629575.1629579

[5] Simon Biggs, Damon Lee, and Gernot Heiser. 2018. The Jury Is In: Monolithic OS Design Is Flawed: Microkernel-based Designs Improve Security. In *Proceedings of the 9th Asia-Pacific Workshop on Systems (APSys '18)*. ACM, New York, NY, USA, Article 16, 7 pages. https://doi.org/10.1145/3265723.3265733

[6] Edouard Bugnion, Scott Devine, and Mendel Rosenblum. 1997. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP '97)*. ACM, New York, NY, USA, 143–156. https://doi.org/10.1145/268998.266672

[7] Microsoft News Center. 2018. DJI and Microsoft partner to bring advanced drone technology to the enterprise. https://news.microsoft.com/2018/05/07/dji-and-microsoft-partner-to-bring-advanced-drone-technology-to-the-enterprise/. [Online; accessed 2019-01-08].

[8] Jonathan Corbet. 2017. Toward non-blocking asynchronous I/O. https://lwn.net/Articles/724198/.

[9] Jonathan Corbet. 2019. io_uring, SCM_RIGHTS, and reference-count cycles. https://lwn.net/Articles/779472/. [Online; accessed 2019-04-15].

[10] Jonathan Corbet. 2019. Ringing in a new asynchronous I/O API. https://lwn.net/Articles/776703/. [Online; accessed 2019-01-16].

[11] Jeffrey Dean and Luiz André Barroso. 2013. The Tail at Scale. *Commun. ACM* 56, 2 (Feb. 2013), 74–80. https://doi.org/10.1145/2408776.2408794

[12] Kevin Elphinstone and Gernot Heiser. 2013. From L3 to seL4 What Have We Learnt in 20 Years of L4 Microkernels?. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM, New York, NY, USA, 133–150. https://doi.org/10.1145/2517349.2522720

[13] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. 1995. Exokernel: An Operating System Architecture for Application-level Resource Management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP '95)*. ACM, New York, NY, USA, 251–266. https://doi.org/10.1145/224056.224076

[14] 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. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 51–66. https://www.usenix.org/conference/nsdi18/presentation/firestone

[15] Node.js Foundation. 2009. Node.js. https://nodejs.org/. [Online; accessed 2018-12-08].

[16] John L. Hennessy and David A. Patterson. 2017. *Computer Architecture, Sixth Edition: A Quantitative Approach* (6th ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[17] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. 2018. The eXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT '18)*. ACM, New York, NY, USA, 54–66. https://doi.org/10.1145/3281411.3281443

[18] Michio Honda, Giuseppe Lettieri, Lars Eggert, and Douglas Santry. 2018. PASTE: A Network Programming Interface for Non-Volatile Main Memory. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 17–33. https://www.usenix.org/conference/nsdi18/presentation/honda

[19] Tomas Hruby, Teodor Crivat, Herbert Bos, and Andrew S. Tanenbaum. 2014. On Sockets and System Calls: Minimizing Context Switches

for the Socket API. In *Proceedings of the 2014 International Conference on Timely Results in Operating Systems (TRIOS'14)*. USENIX Association, Berkeley, CA, USA. http://dl.acm.org/citation.cfm?id=2750315.2750323

[20] Intel Corporation. 2018. Intel Ethernet Controller X710/XXV710/XL710 Datasheet. https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/xl710-10-40-controller-datasheet.pdf. [Online; accessed 2019-01-14].

[21] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. Design Guidelines for High Performance RDMA Systems. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '16)*. USENIX Association, Berkeley, CA, USA, 437–450. http://dl.acm.org/citation.cfm?id=3026959.3027000

[22] Antoine Kaufmann, Simon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurthy. 2016. High Performance Packet Processing with FlexNIC. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. ACM, New York, NY, USA, 67–81. https://doi.org/10.1145/2872362.2872367

[23] Jakub Kicinski and Nicolaas Viljoen. 2016. eBPF Hardware Offload to SmartNICs: cls bpf and XDP. *Proceedings of netdev* 1 (2016).

[24] Hyeong-Jun Kim, Young-Sik Lee, and Jin-Soo Kim. 2016. NVMeDirect: A User-space I/O Framework for Application-specific Optimization on NVMe SSDs. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*. USENIX Association, Denver, CO. https://www.usenix.org/conference/hotstorage16/workshop-program/presentation/kim

[25] Avi Kivity. 2016. Qualifying Filesystems for Seastar and ScyllaDB. https://www.scylladb.com/2016/02/09/qualifying-filesystems/. [Online; accessed 2019-04-16].

[26] Avi Kivity. 2017. Different I/O Access Methods for Linux, What We Chose for Scylla, and Why. https://www.scylladb.com/2017/10/05/io-access-methods-scylla/. [Online; accessed 2019-04-06].

[27] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP '09)*. ACM, New York, NY, USA, 207–220. https://doi.org/10.1145/1629575.1629596

[28] Ricardo Koller and Dan Williams. 2017. Will Serverless End the Dominance of Linux in the Cloud?. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS '17)*. ACM, New York, NY, USA, 169–173. https://doi.org/10.1145/3102980.3103008

[29] Patrick Kutch. 2011. PCI-SIG SR-IOV Primer: An Introduction to SR-IOV Technology.

[30] Amit Levy, Bradford Campbell, Branden Ghena, Pat Pannuto, Prabal Dutta, and Philip Levis. 2017. The Case for Writing a Kernel in Rust. In *Proceedings of the 8th Asia-Pacific Workshop on Systems (APSys '17)*. ACM, New York, NY, USA, Article 1, 7 pages. https://doi.org/10.1145/3124680.3124717

[31] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. 2014. Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC '14)*. ACM, New York, NY, USA, Article 9, 14 pages. https://doi.org/10.1145/2670979.2670988

[32] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. 2014. MICA: A Holistic Approach to Fast In-memory Key-value Storage. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI'14)*. USENIX Association, Berkeley, CA, USA, 429–444. http://dl.acm.org/citation.cfm?id=2616448.2616488

[33] Linux Foundation. 2010. Data Plane Development Kit. https://www.dpdk.org. [Online; accessed 2019-01-08].

[34] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. 2013. Unikernels: Library Operating Systems for the Cloud. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. ACM, New York, NY, USA, 461–472. https://doi.org/10.1145/2451116.2451167

[35] Marek Majkowski. 2015. Kernel bypass. https://blog.cloudflare.com/kernel-bypass/. [Online; accessed 2019-01-16].

[36] Steven McCanne and Van Jacobson. 1993. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proceedings of the USENIX Winter 1993 Conference (USENIX'93)*. USENIX Association, Berkeley, CA, USA, 2–2. http://dl.acm.org/citation.cfm?id=1267303.1267305

[37] J. Mogul, R. Rashid, and M. Accetta. 1987. The Packer Filter: An Efficient Mechanism for User-level Network Code. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles (SOSP '87)*. ACM, New York, NY, USA, 39–51. https://doi.org/10.1145/41457.37505

[38] Jeffrey C. Mogul. 2003. TCP Offload is a Dumb Idea Whose Time Has Come. In *Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9 (HOTOS'03)*. USENIX Association, Berkeley, CA, USA, 5–5. http://dl.acm.org/citation.cfm?id=1251054.1251059

[39] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. 2017. Hyperkernel: Push-Button Verification of an OS Kernel. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. ACM, New York, NY, USA, 252–269. https://doi.org/10.1145/3132747.3132748

[40] NVM Express, Inc. 2018. NVM Express Base Specification. https://nvmexpress.org/wp-content/uploads/NVM-Express-1$_3$c-2018.05.24-Ratified.pdf. [Online; accessed 2019-01-14].

[41] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. 2014. Arrakis: The Operating System is the Control Plane. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, Berkeley, CA, USA, 1–16. http://dl.acm.org/citation.cfm?id=2685048.2685050

[42] Luigi Rizzo. 2012. netmap: A Novel Framework for Fast Packet I/O. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. USENIX Association, Boston, MA, 101–112. https://www.usenix.org/conference/atc12/technical-sessions/presentation/rizzo

[43] Karl Rupp. 2018. 42 Years of Microprocessor Trend Data. https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/. [Online; accessed 2018-02-25].

[44] Seastar. 2015. http://www.seastar-project.org/. [Online; accessed 2019-01-14].

[45] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. 2018. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 69–87. https://www.usenix.org/conference/osdi18/presentation/shan

[46] Solarflare Communications, Inc. 2008. OpenOnload. https://www.openonload.org/. [Online; accessed 2019-01-16].

[47] Storage Performance Development Kit. 2015. https://spdk.io/. [Online; accessed 2019-01-15].

[48] Linus Torvalds. 2016. Re: [PATCH 09/13] aio: add support for async openat(). https://lwn.net/Articles/671657/.

[49] Jisoo Yang, Dave B. Minturn, and Frank Hady. 2012. When Poll is Better Than Interrupt. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12)*. USENIX Association, Berkeley, CA, USA, 3–3. http://dl.acm.org/citation.cfm?id=2208461.2208464