I/O is faster than the OS

Nicolai Fischer
nicolai.fischer@fau.de
Friedrich-Alexander-Universität
Erlangen, Bayern

ABSTRACT

Most OS abstractions assume the CPU to be the fastest component in a given system and therefore do a significant amount of work in order to try to utilize slower peripherals as much as possible. In recent years though, single-core performance has stagnated, but the I/O-speeds are ever-increasing, and so are the number of cores per CPU. This results in application performance being held back by legacy abstractions which don't scale well enough with today's core counts and peripherals.

This challenges operating systems to provide more direct ways of interfacing with hardware. One proposal is the parakernel by Enberg et al., which aims remove the OS from the data plane by assigning partitions of hardware devices directly to applications and only multiplexing legacy devices [9].

This paper aims to provide a comprehensive overview of current problems and possible solutions within Linux, while also explaining some of the new technologies involved. Finally, it evaluates the parakernel proposal in this context.

1 INTRODUCTION

With I/O devices continuously getting faster operating systems have to perform more work in order to keep the hardware busy. But since single core performance has stagnated over the last years this task becomes increasingly difficult [11]. At the same time CPU architectures are getting increasingly complicated and less homogeneous [4].

This section outlines some of the shortcomings of current kernels related to legacy abstractions combined with modern hardware.

Section 2 then outlines some of the current solutions in Linux, followed by an overview of the parakernel architecture in section 3.

1.1 Device Queues

Traditionally most I/O devices use DMA to read and write their data. In the case of a network interface card (NIC) it writes a newly received packet into a region of memory owned by the kernel. The exact location is specified by a ring buffer of DMA descriptors. After writing is complete, the NIC generates an interrupt and the kernel network stack begins to process the received data.

This architecture worked well until network devices became too fast for one CPU core to handle. For example in data centers a 40Gbps NIC is nothing out of the ordinary anymore. One such card can receive a packet faster than the last-level cache access time on an Intel Sandy Bridge platform [14]. Even though these processors are now several generations old, newer models have not drastically decreased cache access times. On the networking front the problem has been made even worse now that 100Gbps interfaces are in use today and even faster ones will be widely available in the near future [9].

One optimization to reduce memory load is to skip the system memory entirely. Some high end network cards can write the received data directly into the last-level cache of the CPU. Therefore eliminating the comparatively slow DRAM. But this still does not change the fact, that one CPU core can no longer keep up with the network traffic [14].

Enter multi-queue NICs. Modern high performance network cards have more than one ring buffer of DMA descriptors. This allows the operating system to split the load of packet processing among multiple cores. In fact some even have so many available queues that virtually every application on a given system could have its own dedicated queue. For example an Intel X710 NIC which operates at 10Gbps has 1536 receive and transmit queues [13].

In the context of storage the numbers get even more extreme. The NVMe protocol, which is used for modern, high performance SSDs, specifies up to 65535 independent queues [18]. This would allow demanding applications to use even more than one queue.

The trend is definitely towards faster, smarter devices, which can service more requests in parallel and operating system design has to reflect that in order to fully utilize current and future hardware. Therefore, the OS has to provide applications with more direct access to hardware. This is something the parakernel aims to provide [9].

1.2 Legacy POSIX Abstractions

The Portable Operating System Interface (POSIX) standard was created more than 30 years ago to provide a common API between various UNIX and UNIX-like operating systems. This effort makes it theoretically easy to port applications between OSes and enables simpler cross platform development [12]. Since then, operating systems have evolved significantly in order to meet new and changing requirements of modern devices. Meanwhile, the POSIX standard has not evolved very much. This has some advantages, mainly that software which has been developed against the POSIX API should still work in the same way.

But as already mentioned, new kinds of hardware and new ways of interacting with hardware have emerged which are no longer covered. A recent survey of the usage of POSIX in modern environments like Ubuntu and OS X but also mobile platforms like Android was conducted by Atlidakis et al..

They found that only a relatively small portion of the vast number of POSIX functions were actively used by tested applications. Parts of the standard are not even implemented anymore, because they don't apply to current hardware or are just not useful enough. Malicious applications are known to use some of these more obscure functions, because their implementations are often not well tested and are therefore more likely to contain bugs which can be exploited.

Furthermore, they found a distinctive lack of support for modern use cases, especially in relation to efficient hardware interaction. For example POSIX does not include any direct support for GPU related work. Instead, developers are relying on system-calls like *ioctl* that can be used to communicate with hardware in a very generic way. Userspace libraries like *OpenGL* then build on top of these to provide an actual graphics API. Atlidakis et al. even found, that these kinds of functions where some of the most frequently used one [3], which suggests that some new purpose built abstractions could improve usability and performance significantly.

One of the more recent additions to POSIX is asynchronous I/O, which seems like a step in the right direction, but adoption as been slow. This indicates that the API is not quite satisfactory [7].

1.3 Context Switches are Expensive

In recent years many assumptions about hardware which have been true for a long time have turned to no longer apply. One such assumption is, that system-calls are fast.

In 2018 two significant security vulnerabilities called Spectre [16] and Meltdown [17] changed this however.

Both of them rely on key principles of modern CPU design, namely out-of-order and speculative execution. This allows processors to change the order of the instructions of a program in order to minimize latency. The rationale behind this is, that not all instructions take the same amount of time and use the same parts of a CPU. For example if a variable is not already within a register of the CPU it has to be loaded from memory, which takes time. Instead of waiting for the data to arrive other instructions which technically should be executed afterwards are being processed in the meantime. This improves performance dramatically since the time it takes for data to arrive from memory is more than several instructions.

The problem arises when the CPU executes something out of order that should not have been otherwise. For example if an instruction raises an exception, such as a division by zero, a context switch into the operating system is performed in order to deal with said exception. But while this is happening other instructions could already have been executed and changed the processors internal state. These changes should then be wiped because they could contain sensitive data, but as the Meltdown exploit shows, the hardware fails to do this correctly. Through clever cache accesses involving side channels this data can then be extracted and it is possible to leak arbitrary data, even from within kernel space and other processes [17].

Software mitigation for these kinds of hardware based vulnerabilities is difficult and performance costs are significant, ranging from 2% to 11% in the case of Spectre and Meltdown [19].

But even with some new generations of processors having some of these issues fixed in hardware new vulnerabilities like Fallout [6] are discovered and will likely pose a continuing problem for high performance applications.

Additionally, even with hardware mitigation the cost of context switches still does not disappear and therefore reducing their amount is a new priority for all operating systems in order to maximize performance and efficiency.

2 PREVIOUS WORK

The following section outlines how the problems described in section 1 can be solved or at least partially mitigated in Linux. Specific focus is placed on new alternatives to the regular network stack.

2.1 Zero-copy Architecture

As already mentioned in section 1.1, modern peripherals can be held back by DRAM latency and throughput. It is therefore imperative to try to eliminate unnecessary memory operations. One way to do this is by using a so called zero copy architecture. This means that data from an I/O device is written directly into a buffer that can be used by the application.

Traditionally the kernel multiplexes all hardware, because most devices can not be used in parallel. For example older network cards as mentioned previously, or storage devices. The operating system collects requests from userspace, potentially reorders them according to defined priorities and then issues them to the hardware sequentially. This involves at least one copy of the data from userspace into the kernel or for receiving data in the other direction.

Linux supports the *O_DIRECT* flag with the *open* system-call, which directly uses userspace buffers owned by the application without copying to/from the kernelspace. But this option comes with some caveats like needing specific support from the file system, requiring alignment considerations and other side effects, like bypassing the regular filesystem cache. This is not necessarily a problem for applications like databases which traditionally use their own caching mechanisms and only work with fixed block sizes. For most regular applications this is not the recommended way of doing I/O [1].

The availability of I/O devices with multiple independent queues gives a new incentive to try to build new APIs which are zero copy.

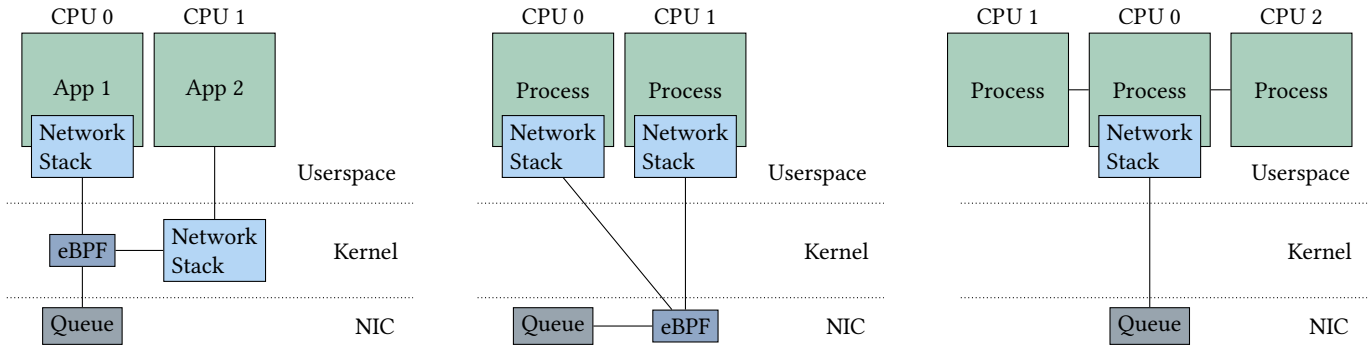
2.2 io_uring

The Linux kernel now provides three interfaces for asynchronous disk I/O: POSIX, the Linux specific *aio* and since recently *io_uring*. The latter addresses a lot of the issues and shortcomings of the previous methods. The *io_uring* subsystem uses ring buffers which are shared between the kernel and userspace. The big improvement is that an application can submit multiple I/O operations to the buffer without any system-calls. To then actually execute the requests, a system-call is necessary unless the application uses the polling mode, in which case the kernel periodically polls the queue and executes the operations without the need for a system-call.

Although *io_uring* is still very new and there are not a lot of applications using it yet, initial performance tests seem very promising [7]. Overall this approach is not too different from the way the parakernel works, with the exception, that the actual I/O operation is still done inside the kernel.

2.3 XDP and eBPF

2.3.1 Introduction. Linux offers a very flexible and capable network stack with lots of functionalities which comes at a not insignificant amount of overhead. But most applications only need a small subset of all these features. This is where XDP comes into play. XDP stands for eXpress Data Path and is a new technology that enables the Linux kernel to do its packet processing more efficiently.



(a) XDP and legacy network stack **(b) XDP with hardware offloading** **(c) Kernel bypass using DPDK**

Figure 1: Kernel participation in different network architectures

It does this by allowing users to write custom packet processing software which will then be executed inside the kernel. In order to do this safely, the code is statically analyzed before runtime and only executed within a virtual machine. The used byte code is in a format called eBPF, which is an extension to the well known BSD Packet Filter, which has been around since the early nineties. But users do not have to write their programs in this format directly, instead they can use several higher level languages like C. The software is then dynamically compiled into eBPF and then native machine code, which is why this new approach is so much faster.

An eBPF program itself is stateless, meaning it does not have any persistent memory across executions, but in order to do more complex tasks, the kernel provides CPU local and global key/value stores called BPF maps. These can store complex data types, like arrays, hashmaps, trees or even pointers to other eBPF programs. They be accessed by one eBPF program in different executions, multiple programs running in various places within the kernel or even the userspace. This enables easy interaction between programs, the kernel and the user.

But the kernel provides even more useful functions to eBPF programs, for example checksumming routines. It is necessary to provide such common functionality, otherwise many programs would have to re-implement them.

In order to remove as much of the overhead of the kernel network stack as possible, eBPF programs are executed directly after a packet arrives. Their return code then determines what happens to the packet. It can either be dropped, sent out the same interface again, passed to the regular kernel network stack or be redirected to a different network interface, a specific CPU or a new type of socket. The latter is the most interesting one, because it makes the packet available to userspace directly, without most of the overhead of the network stack. This socket uses the family *AF_XDP* and enables an userspace application to do its own packet processing [11].

Figure 1a shows two applications as they could work in Linux today. App 1 is using an *AF_XDP* socket and does its own packet processing in userspace, while App 2 uses the legacy in kernel network stack and a regular POSIX socket.

2.3.2 Hardware offload. Modern NICs offer increasingly more offload capabilities. In fact many have programmable multicore processors, ASICSs or even FPGAs onboard [10]. Since eBPF is an open standard it is possible to compile into a format which can be executed directly on a NIC, therefore removing kernel from the data plane entirely as demonstrated in figure 1b. This of course needs specific driver support, but such smart NICs are becoming increasingly common.

The parakernel will leverage this technology in order to remove itself completely from the data plane and let the NIC steer the packets directly to the correct process which will be explained in more detail in section 3.2.

2.3.3 Application level partitioning. But it is possible to take this concept even further. One could actually use XDP to do load balancing between multiple processes that serve the same application. This is possible because an eBPF program can decode the request headers of a packet and for a known partitioning scheme therefore steer it to the correct CPU core. Enberg et al. are currently working on a Memcache compatible Key-Value Store to demonstrate the viability of this approach [10].

2.4 Kernel Bypass with SPDK and DPDK

One obvious way of reducing context switches and eliminate the slow kernel network stack is to circumvent the kernel entirely. This can be accomplished with so called kernel bypass frameworks like the Storage Performance Development Kit (SPDK) [2] and the Data Plane Development Kit (DPDK) for networking [8]. They can be used by high performance applications to get exclusive access to I/O devices. By implementing a minimal network stack in userspace applications can drastically improve performance by eliminating context switches, per-packet memory allocation and locking, tasks usually performed by the kernel [10].

Also, they do no longer need to use system-call intensive POSIX sockets but instead get the packets directly written into buffers within the memory of the application. Since one CPU core is still often not fast enough, applications usually dedicate several cores to packet processing, each with access to its own hardware queue.

Figure 1c illustrates that the kernel is not taking part in any of the data plane operations.

But bypassing the kernel comes with some significant drawbacks. By design the application gets exclusive access to the device, which means in case of a network card that the server likely needs a secondary interface for managing purposes, increasing the amount of switches and other network infrastructure. In case of storage, the system needs a dedicated boot disk and independent data drives. This might not be a huge issue, but is something to be aware of.

A much bigger concern is that applications need to implement their own drivers, since the kernel is no longer involved. This increases complexity and ties the software closer to the used hardware, potentially requiring more work in order to run the application on newer hardware.

Debugging and security auditing is more difficult, since many well tested tools and methodologies can not be used without the kernel interfacing with the hardware.

Even though kernel bypass is currently one of the fastest ways of doing I/O, hardware offloaded XDP has some potential performance benefits over it. Mainly that the CPU cores assigned to packet processing still need to direct the data to the right cores which will then serve the actual requests. Depending on memory topology and structure of the application this could require copying the data or accessing it using NUMA. With XDP done by the network card it is possible to always write the data directly into the local memory of the core which will then serve the request [10].

3 PARAKERNEL

After outlining current problems with modern hardware and old abstractions in section 1 and explaining some solutions which are available today within Linux in section 2, this section will now introduce the parakernel architecture. Conceptualized last year by Enberg et al., this new design aims to solve all the mentioned issues and provide a more scalable and performant kernel design, for current and future hardware [9].

3.1 Multikernel Architecture

Enberg et al. describe their parakernel design as comparable to multikernels with the exception that the parakernel tries not to take part in any data plane operations [9].

In a monolithic kernel the kernel address space is shared between all cores. So if a process tries to acquire a shared resource, the kernel uses locking to ensure that no other process is currently using the same resource. This synchronization becomes increasingly expensive with growing CPU core counts and system memory. For example the memory allocation code has to ensure that the same page of memory does not get allocated by two different processes.

But modern CPU architectures introduce even more complications. Many servers have more than one physical processor, resulting in multiple memory controllers, in the simplest case on per CPU. With nonuniform memory access (NUMA) each CPU can access the entire memory, no matter if it is directly connect to itself or not. Accessing non-local memory introduces more latency and also potential bandwidth problems since the CPU interconnects are limited in their throughput.

Recent CPUs take this even further by being composed of multiple clusters of processing cores. While some clusters have a memory controller, others may not and only be able to access memory through the CPU internal interconnects. This further complicates memory allocation and scheduling policies.

Such networked topologies are similar to distributed systems which is where the inspiration for the multikernel comes from. Rather than seeing all state as implicitly shared, it is explicitly replicated. This is done by message passing, which can be implemented very efficiently and further optimized by batching messages and pipelining [4].

Essentially each CPU core is running its own OS instance and only working with its partitioned hardware slice. For DRAM this task can easily be accomplished by the MMU. I/O devices are a bit more complicated and this is where the parakernel differentiates itself. Namely, by leveraging modern hardware capabilities and partitioning these previously shared devices [9].

3.2 Partition Devices

Modern devices are able to execute more than one operation in parallel, by using multiple queues as explained in section 1.1. The parakernel design allows applications to directly interface with these queues rather than to pass the data through kernel buffers. Since DMA uses physical memory addresses it is crucial for stability and security that the memory locations are in fact valid.

This needs to be enforced by the kernel. An application wanting to use the network has to register dedicated buffers with the kernel through a system-call. The parakernel then verifies if the request itself is valid, for example that a requested port is not already in use or the buffer is of the correct size and possibly alignment. If everything is correct the hardware then gets partitioned accordingly and the application can now use its slice without further interaction with the kernel.

Of course the hardware needs to be able to direct data into the right queues. Until recently, most network cards only supported relatively basic rules about where to steer traffic. But fully programmable NICs are becoming increasingly common and can be used make this architecture possible. Devices will be partitioned by using eBPF programs to direct the network traffic, similar to Linux with hardware offloaded XDP as explained in section 2.3.

The parakernel design aims to remove the kernel from data plane operations, but this is only possible if the hardware is able to be partitioned. Legacy devices like SATA drives or hardware timers can not be shared between applications safely. In these cases the parakernel multiplexes the devices just like Linux is operating today [9].

3.3 Eliminate Legacy Abstractions

Many applications today use blocking system-calls and therefore multiple kernel threads in order to increase CPU utilization and performance. But this comes with significant overhead due to context switches as explained in section 1.3.

In order to reduce these as much as possible the parakernel provides no API for kernel threads. Therefore, applications have to be implemented as potentially multiple processes all running on their dedicated CPU core. In order to increase concurrency, software

has to use an asynchronous design pattern like fibers or coroutines. Consequently, all system-calls are asynchronous as well.

This API design is optimal for the thread-per-core pattern in which an application consists out the same number of threads, as processor cores. By utilizing thread pinning the amount of scheduling and context switches can be minimized, which results in a more consistent tail-latency. In the parakernel model, this has to be implemented with processes, because threads are not available. This pattern is mostly used by high performance and latency sensitive applications like key/value stores.

The parakernel will not provide POSIX compatibility directly. Instead, userspace libraries can implement a POSIX layer on top of the provided kernel API to provide backwards compatibility with legacy applications.

Memory management is tailored towards low latency server applications. This includes easy access to memory which can not be swapped out and eliminating implicitly blocking memory mapped I/O. Enberg et al. argue that applications are better off doing their own caching, than the kernel guessing what data will be needed next [9].

3.4 Security

Monolithic kernels have always been considered to be potentially less secure than micro kernels, simply because they contain a larger trusted computing base (TCB) and are therefore more likely to contain bugs. Even though it is theoretically possible to verify an entire kernel, the amount of work required and the special design process necessary to do this are so complicated and labor intensive that even verifying a micro kernel can take years. It is therefore not practical and might not even be possible to verify a bigger kernel [15].

Regardless of this unfortunate reality, the initial assumption still holds. The smaller the kernel is and the less complicated operations it has to perform, the easier it is for developers to fully grasp all aspects of it and therefore potentially make fewer mistakes implementing it. Even in the event of a bug, finding its cause is simpler and therefore a smaller TCB is still potentially more secure than a larger one [5].

The parakernel leverages this principle and aims to be more secure, because of its smaller TCB [9].

4 CONCLUSION

With growing core counts and increasingly complex memory hierarchies, the overhead caused by system wide synchronization continually increases. By dividing the hardware into smaller, more independent partitions, a significant amount of otherwise necessary synchronization can be eliminated entirely. This architecture will allow the parakernel to run more efficiently on current and future multi-core systems.

At the same time, the parakernel eliminates legacy abstractions, which waste memory bandwidth, because they are not implemented as a zero-copy architecture. Furthermore they waste CPU cycles, because they require many system-calls, which cause context switches. And finally, legacy abstractions are built around the concept, that the operating system has to multiplex all hardware, but with current devices this is no longer necessary. Network cards and SSDs can be partitioned between applications securely, without further interaction with the kernel. This eliminates the operating system from the data plane and therefore saves context switches, memory copies and reduces latency.

In order to save as many context switches as possible, the parakernel embraces a process-per-core pattern for software design. This means there is no API for kernel threads, instead applications use asynchronous primitives, like coroutines or fibers. Consequently all system-calls are non-blocking as well, which increases the difficulty of writing applications compared to a more classic pattern with a thread pool and blocking I/O. But with good support from libraries, managed runtimes and modern languages this inconvenience can be minimized [9]. Furthermore many high performance applications are already using an asynchronous thread-per-core model today [10].

Regardless of all these new and improved APIs, POSIX compatibility is still a desirable feature, since many applications are currently relying on it. This can be achieved by implementing POSIX in userspace libraries, which is already an established way of doing this [9].

Overall, the parakernel has a smaller API, and therefore TCB, than a traditional monolithic kernel like Linux. Combined with an implementation in a high-level language like Rust it can therefore be more secure [9].

Ultimately Linux can already do a lot of the things, if explicitly requested, that the parakernel does by design, like CPU local memory, thread pinning and even asynchronous I/O with io_uring and hardware offloaded XDP [10].

So the questions comes down to, 'How much quicker and more scalable can a kernel which is designed from the ground up with these modern problems in mind, be in comparison to a monolithic kernel like Linux.' The team around Enberg et al. is currently working on a prototype OS, which will ultimately have to answer that question [9].

REFERENCES

[1] 2020. open(2) - Linux manual page. https://man7.org/linux/man-pages/man2/open.2.html

[2] 2020. Storage Performance Development Kit. https://spdk.io/

[3] 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)*. Association for Computing Machinery, London, United Kingdom, 1–17. https://doi.org/10.1145/2901318.2901350

[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)*. Association for Computing Machinery, Big Sky, Montana, 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)*. Association for Computing Machinery, Jeju Island, Republic of Korea, 1–7. https://doi.org/10.1145/3265723.3265733

[6] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. 2019. Fallout: Leaking Data on Meltdown-resistant CPUs. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19)*. Association for Computing Machinery, London, United Kingdom, 769–784. https://doi.org/10.1145/3319535.3363219

[7] Jonathan Corbet. 2020. The rapid growth of io_uring [LWN.net]. https://lwn.net/Articles/810414/

[8] DPDK Project. 2020. Data Plane Development Kit. https://www.dpdk.org/ Library Catalog: www.dpdk.org.

[9] 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 *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '19)*. Association for Computing Machinery, Bertinoro, Italy, 81–87. https://doi.org/10.1145/3317550.3321426

[10] Pekka Enberg, Ashwin Rao, and Sasu Tarkoma. 2019. Partition-Aware Packet Steering Using XDP and eBPF for Improving Application-Level Parallelism. In *Proceedings of the 1st ACM CoNEXT Workshop on Emerging in-Network Computing Paradigms (ENCP '19)*. Association for Computing Machinery, Orlando, FL, USA, 27–33. https://doi.org/10.1145/3359993.3366766

[11] 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)*. Association for Computing Machinery, Heraklion, Greece, 54–66. https://doi.org/10.1145/3281411.3281443

[12] IEEE and The Open Group. 2001. POSIX.1 Backgrounder. http://www.opengroup.org/austin/papers/backgrounder.html

[13] Intel Corporation. 2019. 710-series-datasheet-v-3-6-5-08-28-2019-final.pdf. https://cdrdv2.intel.com/v1/dl/getContent/332464

[14] 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)*. Association for Computing Machinery, Atlanta, Georgia, USA, 67–81. https://doi.org/10.1145/2872362.2872367

[15] 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)*. Association for Computing Machinery, Big Sky, Montana, USA, 207–220. https://doi.org/10.1145/1629575.1629596

[16] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, San Francisco, CA, USA, 1–19. https://doi.org/10.1109/SP.2019.00002

[17] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, Mike Hamburg, and Raoul Strackx. 2020. Meltdown: reading kernel memory from user space. *Commun. ACM* 63, 6 (May 2020), 46–56. https://doi.org/10.1145/3357033

[18] NVM Express, Inc. 2019. NVM-Express-1_4-2019.06.10-Ratified.pdf. https://nvmexpress.org/wp-content/uploads/NVM-Express-1_4-2019.06.10-Ratified.pdf

[19] Nikolay A. Simakov, Martins D. Innus, Matthew D. Jones, Joseph P. White, Steven M. Gallo, Robert L. DeLeon, and Thomas R. Furlani. 2018. Effect of Meltdown and Spectre Patches on the Performance of HPC Applications. *arXiv:1801.04329 [cs]* (Jan. 2018). http://arxiv.org/abs/1801.04329 arXiv: 1801.04329.