HELDA
HELSINGIN YLIOPISTO
HELSINGFORS UNIVERSITET
UNIVERSITY OF HELSINKI

UNIVERSITY OF HELSINKI

https://helda.helsinki.fi

Transcending POSIX: The End of an Era?

Enberg, Pekka J

2022-09

Enberg , P J , Rao , A , Crowcroft , J & Tarkoma , S 2022 , ' Transcending POSIX: The End of an Era? ' , ;login: . <
https://www.usenix.org/publications/loginonline/transcending-posix-end-era >

http://hdl.handle.net/10138/351602

acceptedVersion

Downloaded from Helda, University of Helsinki institutional repository.

This is an electronic reprint of the original article.

This reprint may differ from the original in pagination and typographic detail.

Please cite the original version.

Transcending POSIX: The End of an Era?

Pekka Enberg
University of Helsinki

Ashwin Rao
University of Helsinki

Jon Crowcroft
University of Cambridge

Sasu Tarkoma
University of Helsinki

In this article, we provide a holistic view of the Portable Operating System Interface (POSIX) abstractions by a systematic review of their historical evolution. We discuss some of the key factors that drove the evolution and identify the pitfalls that make them infeasible when building modern applications.

The POSIX standard [1] defines interfaces for a Unix-like operating system. Unix was the first operating system written by programmers for programmers, and POSIX enables developers to write portable applications that run on different Unix operating system variants and instruction set architectures. The primary use case for Unix was to multiplex storage (*the filesystem*) and provide an interactive environment for humans (*the shell*) [2, 3]. In contrast, the primary use case of many contemporary POSIX-based systems is a service running on a machine in a data center, which have orders of magnitude lower latency requirements. These services cannot expect to run faster from year to year with increasing CPU clock frequencies because the end of Dennard scaling circa 2004 implies that CPU clock frequencies are no longer increasing at the rate that was prevalent during the commoditization of Unix. Furthermore, many argue that Moore's Law is slowing down, so the software can no longer expect to get faster by increased hardware optimizations, driven by increased transistor density. As we move towards the post Moore's Law era of computing, system designers are starting to leverage devices such as fast programmable NICs, special-purpose hardware accelerators, and non-volatile main memory to address the stringent latency constraints of applications.

[4] [5]

1 POSIX Evolution

POSIX's abstractions – processes, filesystems, virtual memory, sockets, and threads – are based on the OS abstractions of the different Unix variants in development between the 1970s and 1980s, such as Research Unix, System V, BSD, SunOS, and others.

The use cases and hardware capabilities of their respective era influenced the abstractions. For example, early Unix ran on PDP-11/20, a 16-bit computer with a single CPU and up to 248 KiB of main memory [6]. As PDP-11/20 lacked memory protection, Unix did not support virtual memory, unlike contemporary OSes of the time such as Multics. Although later PDP-11 variants, such as the PDP-11/70, had a memory mapping unit (MMU) [7], virtual memory was not added to Unix until the emergence of the VAX architecture in the late 1970s [4], which became the primary architecture for Unix at the time. Similarly, Unix did not have a networking abstraction until the emergence of the Internet in the early 1980s, when 4.2BSD introduced the sockets abstraction for remote inter-process communication to abstract TCP/IP networking protocols. Likewise, Unix did not have a thread abstraction until the early 1990s, when multiprocessor machines became more mainstream [8].

Filesystem

A *filesystem* is an abstraction to access and organize bytes of data on a storage device. This abstraction and its I/O interface largely originate from Multics [9], and it was considered the most important abstraction in Unix [2, 10]. However, unlike Unix, which supported only synchronous I/O, Multics also supported asynchronous I/O [11, 12], a feature that would eventually be part of POSIX.

Year	Abstraction	Example Interfaces	Version
'69	Filesystem	`open`, `read`, `write`	V0
'69	Processes	`fork`	V0
'71	Processes	`exec`	V1
'71	Virtual memory	`break`[1]	V1
'73	Pipes	`pipe`	V3
'73	Signals	`signal`	V4
'79	Signals	`kill`	V7
'79	Virtual memory	`vfork`[2]	3BSD
'83	Networking	`socket`, `recv`, `send`	4.2BSD
'83	I/O multiplexing	`select`	4.2BSD
'83	Virtual memory	`mmap`[3]	4.2BSD
'83	IPC	`msgget`, `semget`, `shmget`	SRV1
'87	I/O multiplexing	`poll`	SRV3
'88	Virtual memory	`mmap`	SunOS 4.0
'93	Async. I/O	`aio_submit`	POSIX.1b
'95	Threads	`pthread_create`	POSIX.1c

Table 1: **Timeline of POSIX abstractions and interfaces.** *The abstractions were introduced in different variants of Unix between the 1970s and 1990s. Filesystem and processes are fundamental interfaces that were already present in V0. Virtual memory was introduced in 3BSD in the late 1970s and completed in 4.2BSD and SunOS 4.0 in the 1980s. Networking support was added to 4.2BSD in the 1980s. Asynchronous I/O and threads were introduced in the POSIX standard in the 1990s.*

[1] *The* `break` *system call was later renamed to* `brk` *and another variant* `sbrk` *was added. Both of them are now deprecated.*

[2] *3BSD added support for paging-based virtual memory. They added the* `vfork` *system call to avoid implementing copy-on-write for* `fork` *[4].*

[3] *Although* `mmap` *was designed in 1983, the proposed design was fully implemented in 1986 [5].*

The filesystem abstraction also includes files, directories, special files [2], and hard and symbolic links [13]. A file in a filesystem is a sequence of bytes that the OS does not interpret in any way [2]. This enables OSes to represent hardware devices as special files, and the interfaces to operate on files have become the defacto interfaces for I/O devices.

The filesystem abstraction enables easy integration of I/O devices. However, it can become a bottleneck for fast I/O devices [14, 15, 16, 17].

Processes

A *process* is an abstraction for the execution of an application in a system. Specifically, the application is represented as an *image* that abstracts its execution environment comprising of, among others, the program code (*text*), processor register values, and open files [2]. This image is stored in the filesystem, and the OS ensures that the executing part of the process image resides in memory. The process abstraction has been around since early Unix [2], and it has become vital for time-sharing of computing and I/O resources.

This abstraction has its roots in *multi-programming* which was a technique developed in the mid-1950's to improve hardware utilization while performing I/O [18]. Early Unix running on PDP-7 supported only two processes, one for each terminal attached to the machine [19]; later versions of Unix that were designed to run on the PDP-11 could keep multiple processes in memory.

A process is a processor-centric abstraction that is extremely useful for applications built with the assumption that the execution of the process image is done only on the CPUs. However, the prevalence of hardware devices such as graphics processing units (GPUs), tensor processing units (TPUs), and various other special-purpose accelerators for offloading computation are challenging this assumption.

Virtual memory

Virtual memory is an abstraction that creates an illusion of a memory space that is as large as the storage space [20]. It emerged from the need to automatically take advantage of the speed of the main memory and the cheap storage capacity. The concept of virtual memory dates back to the early 1960s: page-based virtual memory was first introduced in 1962 in the Atlas Supervisor [21], and Multics also supported virtual memory [22].

Virtual memory was added to Unix in the late 1970s, almost a decade after its inception. At its inception, the Unix process address space was divided into three segments: program text (*code*) segment that was shared between all processes but not writable, process data segment that was read/write but private, and stack segment. The `sbrk` system call could grow and shrink the process data segment. However, motivated by the need to run programs that required more storage than the main memory capacity at the time (*e.g.*, Lisp), the MMU in the VAX-11 architecture made paging-based virtual memory possible [4, 23].

This abstraction decouples two related two concepts: *address space*, i.e., the identifiers to address memory, and *memory space*, i.e., the physical locations to store data. Historically, this decoupling had three main objectives: (1) promote *machine independence* with an address space that is independent of the physical memory space, (2) promote *modularity* by allowing programmers to compose programs from independent modules that are linked together at execution time, and (3) make it possible to run *large programs* that would not fit in the physical memory (*e.g.*, Lisp programs). Other benefits of virtual memory include running programs of arbitrary size, running partially loaded programs, and changing memory configuration without recompiling programs. Virtual memory is considered to be a fundamental operating system abstraction, but current hardware and application trends are challenging its core assumptions.

Inter-process communication (IPC)

Abstractions for inter-process communication enable one or more processes to interact with each other. Early versions of Unix supported signals and pipes [2]. Signals enabled programmers to programmatically handle hardware faults, and this mechanism was generalized to allow a process to notify other processes. For instance, a shell process can use signals to stop processes. Pipes are special files that allow processes to exchange data with each other. Pipes do not allow arbitrary processes to exchange data, because a pipe between two processes must be set up by their common ancestor.

With the limitations of pipes and signals, sockets were added to BSD to provide a uniform IPC mechanism for both local and remote processes, i.e., processes running on different host machines. Sockets have become the standard way of networking, however they are not as widely used as platform-specific IPC mechanisms for local IPC [24].

The `mmap` interface for shared memory was envisioned as a IPC mechanism [25, 26], but never quite caught on. Additional IPC mechanisms (semaphores, IPC-specific interface for shared memory, and message queues) were added in POSIX.1b, released in 1993, but have since then been largely replaced by vendor-specific IPC mechanisms [24].

Threads and Asynchronous I/O

Threads and asynchronous I/O are the latecomer abstractions in POSIX for addressing the demands for parallelism and concurrency.

The traditional UNIX process offered a single thread of execution. This inability to support concurrent threads of execution makes a single UNIX process unfit for exploiting the parallelism offered by multiple computing cores. One way to exploit the parallelism is to fork multiple processes but this requires the forked processes to communicate with each other using IPC mechanisms, which are in turn inefficient.

The POSIX asynchronous I/O (AIO) interface was designed to address this growing demand for a non-blocking I/O interface that can be leveraged to improve concurrency. This interface enables processes to invoke I/O operations that are performed asynchronously. However, it can block under various circumstances, and it requires at least two system calls per I/O: one to submit a request, and the other to wait for its completion.

In POSIX, threads emerged in the early 1990s from the need to support parallelism of multicore hardware and enable application-level concurrency [8, 27]. Unlike processes, threads run in the same address space.

	Caching	Sync	Copies	Complexity
read/write	kernel	yes	yes	low
mmap	kernel	yes	no	medium
DIO	user	yes	no	medium
AIO/DIO	user	no	no	high
io_uring	kernel/user	no	yes/no	high

Table 2: **I/O access methods in Linux.**

POSIX threads can be implemented in different ways: *1-on-1*: Every thread runs in their own kernel thread; *N-on-1*: All threads run in a single kernel thread; and *N-on-M*: *N* threads runs in *M* kernel threads [27,28,29]. Managing parallelism in user space is essential for high performance [27]. However, mainstream POSIX OSes settled on the 1-on-1 threading model, citing simplicity of implementation [30, 31]. Regardless, application architectures that use a large amount of threads, such as the staged event-driven architecture (SEDA) [32], are inefficient because of thread overheads [33]. Many high-performance applications are therefore adopting a thread-per-core model where the number of threads equals the number of processing cores, and providing their own interfaces for concurrency [34, 35].

2 Transcending POSIX

Offloading Computation

The POSIX process is a CPU-centric abstraction because CPUs were the central and primary computation resource for many decades of Unix evolution. However, offloading computation from CPUs to domain-specific coprocessors and accelerators such as GPUs for graphics and parallel computing, and NICs for offloading packet processing has become mainstream [36]. The CPU is, therefore, increasingly a coordinator for orchestrating computation across these resources, and the computing power of CPUs is increasingly being used by applications solely to orchestrate the computation across a variety of hardware resources.

However, POSIX does not have the machinery to address coprocessors or accelerators. Consequently, all compute elements that are not the CPU, are treated as I/O devices. The applications therefore need to upload the code and data to the accelerator using an userspace API that integrates with the operating system kernel via an opaque system call such as `fcntl()`. For example, there are APIs such as OpenCL and CUDA for GPGPUs and Vulkan and others for graphics programming [37]. These APIs have to handle things like memory and resource management because POSIX doesn't natively support this type of hardware.

Asynchronous I/O

Asynchronous I/O has its roots in Multics [11,12]. However, POSIX I/O calls have their roots in Unix whose I/O interface was synchronous. Consequently, the POSIX `read`/`write` system calls are synchronous and they result in copying data from the kernel page cache. Synchronous interfaces are a bottleneck for fast I/O and they require applications to use threads for application-level concurrency and parallelism. The `mmap` interface is faster than the traditional `read`/`write` because it avoids the system call overhead and copies between the kernel and user space. However, I/O with `mmap` is synchronous and has more complex error handling. For instance, on a disk full a `write` would return with an error code, while `mmap`-based I/O would require handling a signal. In contrast, Direct I/O (DIO) allows applications to use the same `read` and `write` system calls while bypassing the page cache. However the buffer management and caching is performed in user space. The Asynchronous I/O (AIO) interface provides a new set of system calls that allow the userspace application to submit I/O asynchronously with the `io_submit` system call and poll for I/O completion with the `io_getevents` system call. However, Linux's AIO implementation has some problems: it copies up to 104 B of descriptor and completion metadata in each system call, and the system calls tend to block at times [38].

Linux's `io_uring` interface aims to address these shortcomings, and provide a true asynchronous I/O interface [38]. It was first introduced in the Linux kernel version 5.1, and it uses two lockless single-producer single-consumer (SPSC) queues for communication between the kernel and user space [39]. One queue is

for I/O submission, and it is written to by the application and read by the kernel, while the other queue is for I/O completion, and this queue is written by the kernel and read by the application. Depending on the use case, the application can configure an `io_uring` instance to operate either as interrupt-driven, polled, or kernel-polled. The `io_uring` interface allows a thread to submit I/O requests and keep performing other tasks until the OS notifies it that the I/O operation is complete

Bypassing POSIX I/O

The POSIX I/O model assumes that the kernel performs the I/O, which transports the data to user space for further processing. However, the model does not scale very well for high arrival rates, so one of the early examples for bypassing the POSIX I/O interfaces is the BSD Packet filter (BPF). BPF facilitates user-level packet capture by filtering packets inside a pseudo-machine that runs in the kernel [40]. Packet capture applications first command the kernel to make copies of the packets that arrive at the network interface card (NIC): one copy traverses the network protocol stack, while the other traverses the pseudo-machine. The BPF pseudo-machine executes packet filtering code compiled from a high-level description language before sending the filtered packets to the user space. The Extended Berkeley Packet Filter (eBPF) builds on BPF and allows applications to execute sandboxed programs either in an in-kernel virtual machine or on hardware capable of running the programs [41]. This enables applications to offload the I/O activities such as network protocol processing and implementing the filesystem in user space. Specifically, eBPF enables applications to completely bypass the POSIX abstractions for I/O and implement them in user space. eBPF complements existing kernel-bypass approaches such as DPDK and SPDK that enables applications to by-pass the kernel for networking an storage I/O [42, 43].

Beyond the Machine Abstraction

POSIX provides abstractions for writing applications in a portable manner across Unix-like operating system variants and machine architectures. However, contemporary applications rarely run on a single machine. They increasingly use remote procedure calls (RPC), HTTP and REST APIs, distributed key-value stores, and databases, all implemented with a high-level language such as JavaScript or Python, running on managed runtimes. These managed runtimes and frameworks expose interfaces that hide the details of their underlying POSIX abstractions and interfaces. Furthermore, they also allow applications to be written in a programming language other than *C*, the language of Unix and POSIX. Consequently, for many developers of contemporary systems and services, POSIX is largely obsolete because its abstractions are low-level and tied to a single machine.

Nevertheless, the cloud and serverless platforms are now facing a problem that operating systems had before POSIX: their APIs are fragmented and platform-specific, making it hard to write portable applications. Furthermore, these APIs are still largely CPU-centric, which makes it hard to efficiently utilize special-purpose accelerators and disaggregated hardware without resorting to custom solutions. For example, JavaScript is arguably in a similar position today as POSIX was in the past: it decouples the application logic from the underlying operating system and machine architecture. However, the JavaScript runtime is still CPU-centric, which makes it hard to offload parts of a JavaScript application to run on accelerators on the NIC or storage devices. Specifically, *we need a language to express application logic that enables compilers and language run times to efficiently exploit the capabilities of the plethora of hardware resources emerging across different parts of the hardware stack*. At the same time, *it would be an an interesting thought experiment to ponder how different would the hardware design of these devices be without the CPU-centrism in POSIX*.

3 Concluding Remarks

POSIX has become the standard for operating systems abstractions and interfaces over the decades. Two drivers for the design of the abstractions are the hardware constraints and the use cases of the time. Today, the speed balance between I/O and compute is shifting in favor of I/O, which is partly why coprocessors and special-purpose accelerators are becoming more mainstream. Therefore, we argue that the POSIX era is over, and future designs need to transcend POSIX and re-think the abstractions and interfaces at a higher level. We also argue that the operating system interface has to change to support these higher level abstractions.

Acknowledgements

The authors would like to thank Kirk McKusick, Babak Falsafi, Matt Fleming and the anonymous reviewers of OSDI '20, EuroSys '21, HotOS XVIII, ApSys '21, and ASPLOS '22 for their insightful comments that made this article better.

References

[1] The Open Group. The Open Group Base Specifications Issue 7, 2018 edition, 2018.

[2] Dennis M. Ritchie and Ken Thompson. The UNIX Time-sharing System. *Communications of the ACM*, 17(7):365–375, July 1974.

[3] Peter H. Salus. *A Quarter Century of UNIX*. ACM Press/Addison-Wesley Publishing Co., 1994.

[4] Özalp Babaoglu and William Joy. Converting a Swap-Based System to Do Paging in an Architecture Lacking Page-Referenced Bits. In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, SOSP '81, pages 78–86, 1981.

[5] Robert A. Gingell, Joseph P. Moran, and William A. Shannon. Virtual Memory Architecture in SunOS. In *Proceedings of USENIX Summer Conference*, pages 81–94, 1987.

[6] Computer History Wiki: PDP-11/20. `http://gunkies.org/wiki/PDP-11/20`. [Online; accessed 2019-09-21].

[7] Computer History Wiki: PDP-11/70. `http://gunkies.org/wiki/PDP-11/70`. [Online; accessed 2019-09-21].

[8] M. L. Powell, Steve R. Kleiman, Steve Barton, Devang Shah, Dan Stein, and Mary Weeks. SunOS Multi-thread Architecture. In *Proceedings of the Usenix Winter 1991 Conference, Dallas, TX, USA, January 1991*, pages 65–80, 1991.

[9] Corbató, F. J. and Vyssotsky, V. A. Introduction and overview of the multics system. In *Proceedings of the November 30–December 1, 1965, Fall Joint Computer Conference, Part I*, AFIPS '65 (Fall, part I), pages 185–196, 1965.

[10] Daniel Cooke, Joseph Urban, and Scott Hamilton. Unix and beyond: An interview with Ken Thompson. *Computer*, (5):58–64, 1999.

[11] J. F. Ossanna, L. E. Mikus, and S. D. Dunten. Communications and Input/Output Switching in a Multiplex Computing System. In *Proceedings of the November 30–December 1, 1965, Fall Joint Computer Conference, Part I*, AFIPS '65 (Fall, part I), pages 231–241, 1965.

[12] R. J. Feiertag and E. I. Organick. The Multics Input/Output System. In *Proceedings of the Third ACM Symposium on Operating Systems Principles*, SOSP '71, pages 35–41, 1971.

[13] William Joy, Eric Cooper, Robert Fabry, Samuel Leffler, Kirk McKusick, and David Mosher. 4.2BSD System Manual. Technical report, University of California, Berkeley, 1983.

[14] Ying Wang, Dejun Jiang, and Jin Xiong. Caching or not: Rethinking virtual file system for Non-Volatile main memory. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, Boston, MA, July 2018. USENIX Association.

[15] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. MegaPipe: A New Programming Interface for Scalable Network I/O. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 135–148, 2012.

[16] Xiaofeng Lin, Yu Chen, Xiaodong Li, Junjie Mao, Jiaquan He, Wei Xu, and Yuanchun Shi. Scalable kernel tcp design and implementation for short-lived connections. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, page 339–352, New York, NY, USA, 2016. Association for Computing Machinery.

[17] Bojie Li, Tianyi Cui, Zibo Wang, Wei Bai, and Lintao Zhang. Socksdirect: Datacenter sockets can be fast and compatible. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 90–103, New York, NY, USA, 2019. Association for Computing Machinery.

[18] E.F. Codd. Multiprogramming. *Advances in Computers*, 3:77–153, 1962.

[19] Dennis Ritchie. The Evolution of the Unix Time-Sharing System. In *Lecture Notes in Computer Science #79: Language Design and Programming Methodology*, pages 25–36. Springer-Verlag, 1980.

[20] Peter J. Denning. Virtual Memory. *ACM Computing Surveys*, 2(3):153–189, 1970.

[21] T. Kilburn, R. B. Payne, and D. J. Howarth. The Atlas Supervisor. In *Proceedings of the December 12-14, 1961, Eastern Joint Computer Conference: Computers - Key to Total Systems Control*, AFIPS '61 (Eastern), pages 279–294, 1961.

[22] A. Bensoussan, C. T. Clingen, and R. C. Daley. The Multics Virtual Memory: Concepts and Design. *Commun. ACM*, 15(5):308–318, May 1972.

[23] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1996.

[24] Vaggelis Atlidakis, Jeremy Andrus, Roxana Geambasu, Dimitris Mitropoulos, and Jason Nieh. 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, pages 1–17, 2016.

[25] Samuel J. Leffler, William N. Joy, and Robert S. Fabry. 4.2BSD Networking Implementation Notes (Revised July, 1983). Technical Report UCB/CSD-83-146, EECS Department, University of California, Berkeley, Jul 1983.

[26] M McKusick and M Karels. A New Virtual Memory Implementation for Berkeley UNIX. In *Proceedings of the European UNIX Users Group Meeting*, pages 451–460, 1986.

[27] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler Activations: Effective Kernel Support for the User-level Management of Parallelism. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, SOSP '91, pages 95–109, 1991.

[28] Robert A. Alfieri. An Efficient Kernel-Based Implementation of POSIX Threads. In *USENIX Summer 1994 Technical Conference, Boston, Massachusetts, USA, June 6-10, 1994, Conference Proceeding*, pages 59–72, 1994.

[29] Frank Mueller. A Library Implementation of POSIX Threads under UNIX. In *Proceedings of the Usenix Winter 1993 Technical Conference, San Diego, California, USA, January 1993*, pages 29–42, 1993.

[30] Sun Microsystems. Multithreading in the Solaris Operating Environment. `https://web.archive.org/web/20090327002504/http://www.sun.com/software/whitepapers/solaris9/multithread.pdf`, 2002. [Online; accessed 2019-07-30].

[31] Ulrich Drepper and Ingo Molnar. The Native POSIX Thread Library for Linux. `https://www.cs.utexas.edu/~witchel/372/lectures/POSIX_Linux_Threading.pdf`, 2003. [Online; accessed 2019-07-30].

[32] Matt Welsh, David Culler, and Eric Brewer. SEDA: An Architecture for Well-conditioned, Scalable Internet Services. *ACM SIGOPS Operating Systems Review*, 35(5):230–243, October 2001.

[33] Matt Welsh. A Retrospective on SEDA. http://matt-welsh.blogspot.com/2010/07/retrospective-on-seda.html, 2010. [Online; accessed 2019-07-30].

[34] Seastar. http://www.seastar-project.org/. [Online; accessed 2019-07-30].

[35] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. 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, pages 429–444, 2014.

[36] Jeffrey C. Mogul. 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, pages 5–5, Berkeley, CA, USA, 2003. USENIX Association.

[37] Nicola Capodieci, Roberto Cavicchioli, and Andrea Marongiu. A taxonomy of modern gpgpu programming methods: On the benefits of a unified specification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41(6):1649–1662, 2022.

[38] Jonathan Corbet. Ringing in a new asynchronous I/O API. https://lwn.net/Articles/776703/, 2019. [Online; accessed 2020-05-26].

[39] Jonathan Corbet. Ringing in a new asynchronous i/o api. https://lwn.net/Articles/776703/. [Online; accessed 2022-07-30].

[40] Steven McCanne and Van Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *USENIX Winter 1993 Conference*, San Diego, CA, January 1993. USENIX Association.

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

[42] Data plane development kit. https://www.dpdk.org/. [Online; accessed 2022-07-30].

[43] Z. Yang, J. R. Harris, B. Walker, D. Verkamp, C. Liu, C. Chang, G. Cao, J. Stern, V. Verma, and L. E. Paul. SPDK: A Development Kit to Build High Performance Storage Applications. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 154–161, 2017.