BPF for storage: an exokernel-inspired approach

Yu Jian Wu[1], Hongyi Wang[1], Yuhong Zhong[1],
Asaf Cidon[1], Ryan Stutsman[2], Amy Tai[3], and Junfeng Yang[1]

[1]Columbia University, [2]University of Utah, [3]VMware Research

Abstract

The overhead of the kernel storage path accounts for half of the access latency for new NVMe storage devices. We explore using BPF to reduce this overhead, by injecting user-defined functions deep in the kernel's I/O processing stack. When issuing a series of dependent I/O requests, this approach can increase IOPS by over 2.5× and cut latency by half, by bypassing kernel layers and avoiding user-kernel boundary crossings. However, we must avoid losing important properties when bypassing the file system and block layer such as the safety guarantees of the file system and translation between physical blocks addresses and file offsets. We sketch potential solutions to these problems, inspired by exokernel file systems from the late 90s, whose time, we believe, has finally come!

> "As a dog returns to his vomit, so a fool repeats his folly."
>
> *Attributed to King Solomon*

1 Introduction

Storage devices have historically lagged behind networking devices in achieving high bandwidth and low latency. While 100 Gb/s bandwidth is now common for network interface cards (NICs) and the physical layer, storage devices are only beginning to support 2-7 GB/s bandwidth and 4-5 µs latencies [8, 13, 19, 20]. With such devices, the software stack is now a substantial overhead on every storage request. In our experiments this can account for about half of I/O operation latency, and the impact on throughput can be even more significant.

Kernel-bypass frameworks (e.g. SPDK [44]) and near-storage processing reduce kernel overheads. However, there are clear drawbacks to both such as significant, often bespoke, application-level changes [40, 41], lack of isolation, wasteful busy waiting when I/O usage isn't high, and the need for specialized hardware in the case of computational storage [10, 16, 42]. Therefore, we want a standard OS-supported mechanism that can reduce the software overhead for fast storage devices.

To address this, we turn to the networking community, which has long had high-bandwidth devices. Linux's eBPF [6] provides an interface for applications to embed simple functions directly in the kernel. When used to intercept I/O, these functions can perform processing that is traditionally done in the application and can avoid having to copy data and incurring context switches when going back and forth between the kernel and user space. Linux eBPF is widely used for packet processing and filtering [5, 30], security [9] and tracing [29].

BPF's ubiquity in the kernel and its wide acceptance make it a natural scheme for application-specific kernel-side extensions in layers outside of the networking stack. BPF could be used to chain dependent I/Os, eliminating costly traversals of the kernel storage stack and transitions to/from userspace. For example, it could be used to traverse a disk-based data structure like a B-tree where one block references another. Embedding these BPF functions deep enough in the kernel has the potential to eliminate nearly all of the software overhead of issuing I/Os like kernel-bypass, but, unlike kernel-bypass, it does not require polling and wasted CPU time.

To realize the performance gains of BPF we identify four substantial open research challenges which are unique to the storage use case. First, for ease of adoption, our architecture must support Linux with standard file systems and applications with minimal modifications to either. It should also be efficient and bypass as many software layers as feasible. Second, storage pushes BPF beyond current, simple packet processing uses. Packets are self-describing, so BPF can operate on them mostly in isolation. Traversing a structure on-disk is stateful and frequently requires consulting outside state. Storage BPF functions will need to understand applications' on-disk formats and access outside state in the application or kernel, for example, to synchronize concurrent accesses or to access in-memory structure metadata. Third, we

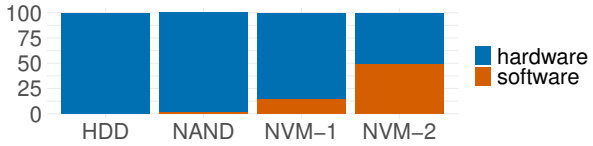


Figure 1: Kernel's latency overhead with 512 B random reads. HDD is Seagate Exos X16, NAND is Intel Optane 750 TLC NAND, NVM-1 is first generation Intel Optane SSD (900P), and NVM-2 is second generation Intel Optane SSD (P5800X)

need to ensure that BPF storage functions cannot violate file system security guarantees while still allowing sharing of disk capacity among applications. Storage blocks themselves typically do not record ownership or access control attributes, in contrast to network packets whose headers specify the flows to which the packets belong. Hence, we need an efficient scheme for enforcing access control that doesn't induce the full cost of the kernel's file system and block layers. Fourth, we need to enable concurrency. Applications support concurrent accesses via fine-grained synchronization (e.g. lock coupling [28]) to avoid read-write interference with high throughput; synchronization from BPF functions may be needed.

Our approach is inspired by the work on exokernel file system designs. User-defined kernel extensions were the cornerstone of XN for the Xok exokernel. It supported mutually distrusting "libfs"es via code downloaded from user processes to the kernel [32]. In XN, these *untrusted deterministic functions* were interpreted to give the kernel a user-defined understanding of file system metadata. Then, the application and kernel design was clean slate, and absolute flexibility in file system layout was the goal. While we require a similar mechanism to allow users to enable the kernel to understand their data layout, our goals are different: we want a design that allows applications to define custom BPF-compatible data structures and functions for traversing and filtering on-disk data, works with Linux's existing interfaces and file systems, and substantially prunes the amount of kernel code executed per-I/O to drive millions of IOPS.

2 Software is the Storage Bottleneck

The past few years have seen new memory technologies emerge in SSDs attached to high bandwidth PCIe using NVMe. This has led to storage devices that now rival the performance of fast network devices [1] with a few microseconds of access latency and gigabytes per second of bandwidth [8, 13, 19, 20]. Hence, just as the kernel networking stack emerged as a CPU bottleneck for fast network cards [22, 34, 37, 43], the kernel storage stack is now becoming a bottleneck for these new devices.

Figure 1 shows this; it breaks down the fraction of read I/O latency that can be attributed to the device hardware and the system software for increasingly fast storage devices. The results show the kernel's added software

kernel crossing	351 ns	5.6%
read syscall	199 ns	3.2%
ext4	2006 ns	32.0%
bio	379 ns	6.0%
NVMe driver	113 ns	1.8%
storage device	3224 ns	51.4%
total	6.27 µs	100.0%

Table 1: Average latency breakdown of a 512 B random `read()` syscall using Intel Optane SSD gen 2.

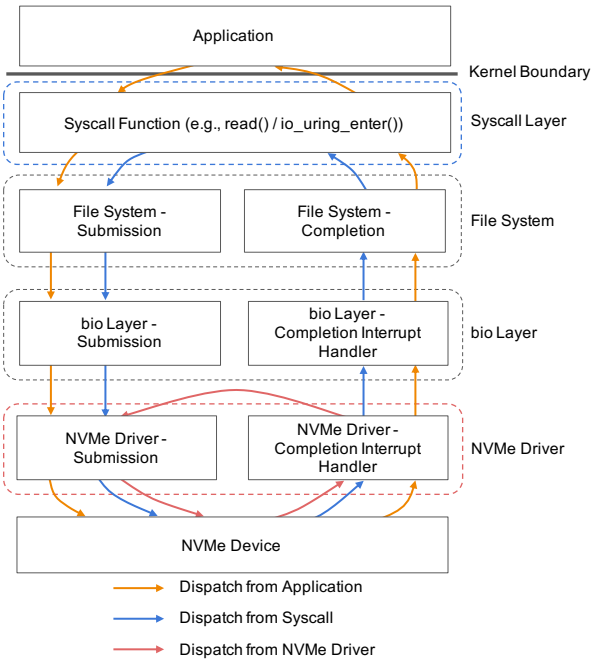
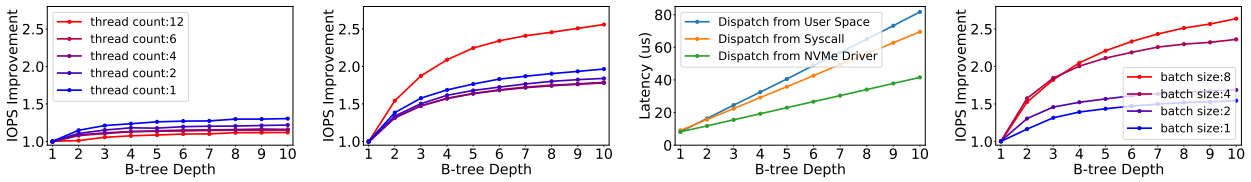


Figure 2: Dispatch paths for the application and the two kernel hooks.

overhead was already measurable (10-15% latency overhead) with the first generation of fast NVMe devices (e.g. first generation Optane SSD or Samsung's Z-NAND); *in the new generation of devices software accounts for about half of the latency of each read I/O.*

The Overhead Source. To breakdown this software overhead, we measured the average latency of the different software layers when issuing a random 512 B `read()` system call with O_DIRECT on an Intel Optane SSD Gen 2 prototype (P5800X) on a 6-core i5-8500 3 GHz server with 16 GB of memory, Ubuntu 20.04, and Linux 5.8.0. We use this setup throughout the paper. We disable processor C-states and turbo boost and use the maximum performance governor. Table 1 shows that the layers that add the most latency are the file system (ext4 here), followed by the transition from user space into kernel space.

Kernel-bypass allows applications directly submit requests to devices, effectively eliminating all of these costs



(a) Lookups with read syscall, using syscall dispatcher layer hook.

(b) Lookups with read syscall, using NVMe driver layer hook.

(c) One-threaded lookup with read syscall, using syscall layer hook and NVMe driver layer hook.

(d) Single-threaded lookups with io_uring syscall, using NVMe driver hook.

Figure 3: Search throughput improvement on B-tree with varying depth when reissuing lookups from different kernel layers.

except the costs to post NVMe requests ("NVMe driver") and the device latency itself [22, 33, 44, 45]. However, eliminating all of these layers comes at a high cost. The kernel can only delegate whole devices to processes, and applications must implement their own file systems on raw devices [40, 41]. Even when they do, they still cannot safely share files or capacity between distrusting processes. Finally, lack of efficient application-level interrupt dispatching means these applications must resort to polling to be efficient at high load; this makes it impossible for them to efficiently share cores with other processes, resulting in wasteful busy polling when I/O utilization isn't high enough.

There have been recent efforts to streamline kernel I/O submission costs via a new system call called *io_uring* [7]. It provides batched and asynchronous I/O submission/completion path that amortizes the costs of kernel boundary crossings and can avoid expensive scheduler interactions to block kernel threads. However, each submitted I/O still must pass through all of the kernel layers shown in Table 1. So, each I/O still incurs significant software overhead when accessing fast storage devices (we quantify this in §3).

3 BPF to the Rescue?

With the rise of fast networks, Berkeley Packet Filter (BPF) has gained in popularity for efficient packet processing since it eliminates the need to copy each packet into userspace; instead applications can safely operate on packets in the kernel. Common networking use cases include filtering packets [4, 5, 21, 30], network tracing [2, 3, 29], load balancing [4, 12], packet steering [27] and network security checks [9]. It has also been used as a way to avoid multiple network crossings when accessing disaggregated storage [35]. Linux supports BPF via the eBPF extension since Linux 3.15 [18]. The user-defined functions can be executed either using an interpreter or a just-in-time (JIT) compiler.

BPF for Storage. We envision similar use of BPF for storage by removing the need to traverse the kernel's storage stack and move data back and forth between the kernel and user space when issuing dependent storage requests. Many storage applications consist of many "auxiliary" I/O requests, such as index lookups. A key characteristic of these requests is that they occupy I/O bandwidth and CPU time to fetch data that is ultimately *not* returned to the user. For example, a search on a B-tree index is a series of pointer lookups that lead to the final I/O request for the user's data page. Each of these lookups makes a roundtrip from the application through the kernel's storage stack, only for the application to throw the data away after simple processing. Other, similar use cases include database iterators that scan tables sequentially until an attribute satisfies a condition [15] or graph databases that execute depth-first searches [24].

The Benefits. We design a benchmark that executes lookups on an on-disk B^+-tree (which we call a B-tree for simplicity), a common data structure used to index databases [17, 31]. For simplicity, our experiments assume that the leaves of the index contain user data rather than pointers [36]. We assume each node of the B-tree sits on a separate disk page, which means for a B-tree of depth d, a lookup requires reading d pages from disk. The core operation of a B-tree lookup is parsing the current page to find the offset of the next disk page, and issuing a read for the next page, a "pointer lookup". Traditionally, a B-tree lookup requires d successive pointer lookups from userspace. To improve on the baseline, we reissue successive pointer lookups from one of two hooks in the kernel stack: the syscall dispatch layer (which mainly eliminates kernel boundary crossings) or the NVMe driver's interrupt handler on the completion path (which eliminates nearly all of the software layers from the resubmission). Figure 2 shows the dispatch paths for the two hooks along with the normal user space dispatch path. These two hooks are proxies for the eBPF hooks that we would ultimately use to offload user-defined functions.

Figures 3a and 3b show the throughput speedup of both hooks relative to the baseline application traversal, and Figure 3c shows the latency of both hooks while varying the depth of the B-tree. When lookups are reissued from the syscall dispatch layer, the maximum speedup is 1.25×. The improvement is modest because each lookup still incurs the file system and block layer overhead; the speedup comes exclusively from eliminating kernel boundary crossings. As storage devices approach 1 µs latencies, we expect greater speedups from this dispatch

hook. On the other hand, reissuing from the NVMe driver makes subsequent I/O requests significantly less computationally expensive, by bypassing nearly the entire software stack. Doing so achieves speed ups up to 2.5×, and reduces latency by up to 49%. Relative throughput improvement actually goes down when adding more threads, because the baseline application also benefits from more threads until it reaches CPU saturation at 6 threads. Once the baseline hits CPU saturation, the computational savings due to reissuing at the driver becomes much more apparent. The throughput improvement from reissuing in the driver continues to scale with deeper trees, because each level of the tree compounds the number of requests that are issued cheaply.

What about io_uring? The previous experiments use Linux's standard, synchronous `read` system call. Here, we repeat these experiments using the more efficient and batched io_uring submission path to drive B-tree lookups from a single thread. Like before, we reissue lookups from within the NVMe driver and plot the throughput improvement against an application that simply batches I/Os using unmodified io_uring calls. Figure 3d shows the throughput speedup due to reissuing from within the driver relative to the application baseline.

As expected, increasing the batch size (number of system calls batched in each io_uring call), increases the speedup, since a higher batch size increases the number of requests that can be reissued at the driver. For example, for a batch size of 1 only 1 request (per B-tree level) can be reissued inexpensively, whereas for a batch size of 8, each B-tree level saves on 8 concurrent requests. Therefore, placing the hooks close to the device benefits both standard, synchronous `read` calls and more efficient io_uring calls. With deep trees, BPF coupled with io_uring delivers $> 2.5\times$ higher throughput; even three dependent lookups give 1.3–1.5× speedups.

4 A Design for Storage BPF

Our experiments have given us reason to be optimistic about BPF's potential to accelerate operations with fast storage devices; however, to realize these gains, I/O resubmissions must happen as early as possible, ideally within the kernel NVMe interrupt handler itself. This creates significant challenges in using BPF to accelerate storage lookups for a practical system such as a key-value store.

We envision building a library that provides a higher level-interface than BPF and new BPF hooks in the Linux kernel as early in the storage I/O completion path as possible, similar to XDP [21]. This library would contain BPF functions to accelerate access and operations on popular data structures, such as B-trees and log-structured merge trees (LSM).

Within the kernel, these BPF functions that would be triggered in the NVMe driver interrupt handler on each block I/O completion. By giving these functions access to raw buffers holding block data, they could extract file offsets from blocks fetched from storage and immediately reissue an I/O to those offsets; they could also filter, project, and aggregate block data by building up buffers that they later return to the application. By pushing application-defined structures into the kernel these functions can traverse persistent data structures with limited application involvement. Unlike XN, where functions were tasked with implementing full systems, these storage BFP functions would mainly be used to define the layout of a storage blocks that make up application data structures.

We outline some of the key design considerations and challenges for our preliminary design, which we believe we can realize without substantial re-architecture of the Linux kernel.

Installation & Execution. To accelerate dependent accesses, our library installs a BPF function using a special `ioctl`. Once installed, the application I/Os issued using that file descriptor are "tagged"; submissions to the NVMe layer propagate this tag. The kernel I/O completion path, which is triggered in the NVMe device interrupt handler, checks for this tag. For each tagged submission/completion, our NVMe interrupt handler hook passes the read block buffer into the BPF function.

When triggered, the function can perform a few actions. For example, it can extract a file offset from the block; then, it can "recycle" the NVMe submission descriptor and I/O buffer by calling a helper function that retargets the descriptor to the new offset and reissues it to the NVMe device submission queue. Hence, one I/O completion can determine the next I/O that should be submitted with no extra allocations, CPU cost, or delay. This lets functions perform rapid traversals of structures without application-level involvement.

The function can also copy or aggregate data from the block buffer into its own buffers. This lets the function perform selection, projection, or aggregation to build results to return to the application. When the function completes it can indicate which buffer should be returned to the application. For cases where the function started a new I/O and isn't ready to return results to the application yet (for example, if it hasn't found the right block yet), it can return no buffer, preventing the I/O completion from being raised to the application.

Translation & Security. In Linux the NVMe driver doesn't have access to file system metadata. If an I/O completes for a block at offset o_1 in a file, a BPF function might extract file offset o_2 as the next I/O to issue. However, o_2 is meaningless to the NVMe context, since it cannot tell which physical block this corresponds to without access to the file's metadata and extents. Blocks could embed physical block addresses to avoid the need

to consult the extents, but without imposing limits on these addresses, BPF functions could access any block on the device. Hence, a key challenge is imbuing the NVMe layer with enough information to efficiently and safely map file offsets to the file's corresponding physical block offsets without restricting the file system's ability to remap blocks as it chooses.

For simplicity and security in our design, each function only uses the file offsets in the file to which the `ioctl` attached the function. This ensures functions cannot access data that does not belong to the file. To do this without slow file system layer calls and without constraining the file system's block allocation policy, we plan to only trigger this block recycling *when the extents for a file do not change*. We make the observation that many data center applications do not modify persistent structures on block storage in place. For example, once an LSM-tree writes SSTable files to disk, they are immutable and their extents are stable [26]. Similarly, the index file extents remain nearly stable in on-disk B-tree implementations; In a 24 hour YCSB [25] (40% reads, 40% updates, 20% inserts, Zipfian 0.7) experiment on MariaDB running TokuDB [14], we found the index file's extents only changed every 159 seconds on average with only 5 extent changes in 24 hours unmapping any blocks. Note that in these index implementations, each index is stored on a single file, and does not span multiple files, which further helps simplify our design.

We exploit the relative stability of file extents via a soft state cache of the extents at the NVMe layer. When the `ioctl` first installs the function on the file storing the data structure, its extents are propagated to the NVMe layer. If any block is unmapped from any of the file's extents, a new hook in the file system triggers an invalidation call to the NVMe layer. Ongoing recycled I/Os are then discarded, and an error is returned to the application layer, which must rerun the `ioctl` to reset the NVMe layer extents before it can reissue tagged I/Os. This is a heavy-handed but simple approach. It leaves the file system almost entirely decoupled from the NVMe layer, and it places no restrictions on the file system block allocation policies. Of course, these invalidations need to be rare for the cache to be effective, but we expect this is true in most of the applications we target.

I/O Granularity Mismatches. When the BIO layer "splits" an I/O, e.g. across two discontiguous extents, it will generate multiple NVMe operations that complete at different times. We expect these cases to be rare enough that we can perform that I/O as a normal BIO and return the buffer and completion to the application. There, it can run the BPF function itself and restart the I/O chain with the kernel starting at the next "hop". This avoids extra complexity in the kernel. Similarly, if application needs to generate more than one I/O in response to a single I/O completion, we propagate the completion up to the BIO layer which allocates and submits the multiple I/Os to the NVMe layer. This avoids returning to userspace.

Caching. As the caching of indices is often managed by the application [14, 23, 26], we assume the BPF traversal will not interact with the buffer cache directly and that applications manage caching and synchronizing with traversals. Cache eviction and management is increasingly done at the granularity of application-meaningful objects (e.g. individual data records) instead of whole pages. Our scheme fits well into this model, where BPF functions can return specific objects to the application rather than pages, to which it can apply its own caching policies.

Concurrency and Fairness. A write issued through the file system might only be reflected in the buffer cache and would not be visible to the BPF traversal. This could be addressed by locking, but managing application-level locks from within the NVMe driver could be expensive. Therefore, data structures that require fine-grained locking (e.g. lock coupling in B+trees [28]) require careful design to support BPF traversal.

To avoid read/write conflicts, we initially plan to target data structures that remain immutable (at least for a long period of time). Fortunately, many data structures have this property, including LSM SSTable files that remain immutable [26, 39], and on-disk B-trees that are not updated dynamically in-place, but rather in a batch process [14]. In addition, due to the difficulty of acquiring locks, we plan initially to only support read-only BPF traversals.

BPF issued requests do not go through the file system or block layer, so there is no easy place to enforce fairness or QoS guarantees among processes. However, the default block layer scheduler in Linux is the noop scheduler for NVMe devices, and the NVMe specification supports command arbitration at hardware queues if fairness is a requirement [11]. Another challenge is that the NVMe layer may reissue an infinite number of I/O requests. The eBPF verifier prevents loops with unknown bounds [38], but we would also need to prevent unbounded I/O loops at our NVMe hook.

For fairness purposes and to prevent unbounded traversals, we plan to implement a counter per process in the NVMe layer that will track the number of chained submissions, and set a bound on this counter. The counter's values can periodically be passed to the BIO layer to account the number of requests.

5 Conclusions

BPF has the potential to significantly speed up dependent lookups to fast storage devices. However, it creates several key challenges, arising due to the loss of context when operating deep in the kernel's storage stack. In this paper, we focused primarily on enabling the ini-

tial use case of index traversals. Notwithstanding, even for current fast NVMe devices (and more so for future ones), chaining a small number of requests using BPF provides significant gains. We envision a BPF for storage library could help developers offload many other standard storage operations to the kernel, such as compaction, compression, deduplication and scans. We also believe that the interactions of BPF with the cache and scheduler policies create exciting future research opportunities.

6 Acknowledgments

We would like to thank Frank Hady and Andrew Ruffin for their generous support, and to Adam Manzanares, Cyril Guyot, Qing Li and Filip Blagojevic for their guidance throughout the project and feedback on the paper.

References

[1] 200Gb/s ConnectX-6 Ethernet Single/Dual-Port Adapter IC | NVIDIA. https://www.mellanox.com/products/ethernet-adapter-ic/connectx-6-en-ic.

[2] bcc. https://github.com/iovisor/bcc.

[3] bpftrace. https://github.com/iovisor/bpftrace.

[4] Cilium. https://github.com/cilium/cilium.

[5] Cloudflare architecture and how BPF eats the world. https://blog.cloudflare.com/cloudflare-architecture-and-how-bpf-eats-the-world/.

[6] eBPF. https://ebpf.io/.

[7] Efficient io with io_uring. https://kernel.dk/io_uring.pdf.

[8] Intel® Optane™ SSD DC P5800X Series. https://ark.intel.com/content/www/us/en/ark/products/201859/intel-optane-ssd-dc-p5800x-series-1-6tb-2-5in-pcie-x4-3d-xpoint.html.

[9] MAC and Audit policy using eBPF. https://lkml.org/lkml/2020/3/28/479.

[10] Ngd systems newport platform. https://www.ngdsystems.com/technology/computational-storage.

[11] NVMe base specification. https://nvmexpress.org/wp-content/uploads/NVM-Express-1_4b-2020.09.21-Ratified.pdf.

[12] Open-sourcing katran, a scalable network load balancer. https://engineering.fb.com/2018/05/22/open-source/open-sourcing-katran-a-scalable-network-load-balancer/.

[13] Optimizing Software for the Next Gen Intel Optane SSD P5800X. https://www.intel.com/content/www/us/en/events/memory-and-storage.html?videoId=6215534787001.

[14] Percona tokudb. https://www.percona.com/software/mysql-database/percona-tokudb.

[15] RocksDB iterator. https://github.com/facebook/rocksdb/wiki/Iterator.

[16] SmartSSD computational storage drive. https://www.xilinx.com/applications/data-center/computational-storage/smartssd.html.

[17] SQL server index architecture and design guide. https://docs.microsoft.com/en-us/sql/relational-databases/sql-server-index-design-guide.

[18] A thorough introduction to eBPF. https://lwn.net/Articles/740157/.

[19] Toshiba memory introduces XL-FLASH storage class memory solution. https://business.kioxia.com/en-us/news/2019/memory-20190805-1.html.

[20] Ultra-Low Latency with Samsung Z-NAND SSD. https://www.samsung.com/semiconductor/global.semi.static/Ultra-Low_Latency_with_Samsung_Z-NAND_SSD-0.pdf.

[21] XDP. https://www.iovisor.org/technology/xdp.

[22] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages

49–65, Broomfield, CO, October 2014. USENIX Association.

[23] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. FASTER: A concurrent key-value store with in-place updates. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 275–290, New York, NY, USA, 2018. Association for Computing Machinery.

[24] J Chao. Graph databases for beginners: Graph search algorithm basics. https://neo4j.com/blog/graph-search-algorithm-basics/.

[25] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[26] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. Optimizing space amplification in rocksdb. In *CIDR*, volume 3, page 3, 2017.

[27] Pekka Enberg, Ashwin Rao, and Sasu Tarkoma. 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, page 27–33, New York, NY, USA, 2019. Association for Computing Machinery.

[28] Goetz Graefe. A Survey of B-Tree Locking Techniques. *ACM Transactions on Database Systems*, 35(3), July 2010.

[29] Brendan Gregg. *BPF Performance Tools*. Addison-Wesley Professional, 2019.

[30] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. 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*, pages 54–66, 2018.

[31] Stratos Idreos, Niv Dayan, Wilson Qin, Mali Akmanalp, Sophie Hilgard, Andrew Ross, James Lennon, Varun Jain, Harshita Gupta, David Li, et al. Design continuums and the path toward self-designing key-value stores that know and learn. In *CIDR*, 2019.

[32] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Hector M. Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application Performance and Flexibility on Exokernel Systems. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, page 52–65, New York, NY, USA, 1997. Association for Computing Machinery.

[33] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for μsecond-scale tail latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 345–360, 2019.

[34] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter RPCs can be general and fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 1–16, Boston, MA, February 2019. USENIX Association.

[35] Kornilios Kourtis, Animesh Trivedi, and Nikolas Ioannou. Safe and Efficient Remote Application Code Execution on Disaggregated NVM Storage with eBPF. *arXiv preprint arXiv:2002.11528*, 2020.

[36] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Wisckey: Separating keys from values in ssd-conscious storage. *ACM Transactions on Storage (TOS)*, 13(1):1–28, 2017.

[37] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Mike Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Mike Ryan, Erik Rubow, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: a microkernel approach to host networking. In *In ACM SIGOPS 27th Symposium on Operating Systems Principles*, New York, NY, USA, 2019.

[38] Luke Nelson, Jacob Van Geffen, Emina Torlak, and Xi Wang. Specification and verification in the field: Applying formal methods to BPF just-in-time compilers in the linux kernel. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 41–61. USENIX Association, November 2020.

[39] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.

[40] Weikang Qiao, Jieqiong Du, Zhenman Fang, Michael Lo, Mau-Chung Frank Chang, and Jason Cong. High-throughput lossless compression on tightly coupled CPU-FPGA platforms. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 37–44. IEEE, 2018.

[41] Zhenyuan Ruan, Tong He, and Jason Cong. INSIDER: Designing in-storage computing system for emerging high-performance drive. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 379–394, 2019.

[42] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. Willow: A user-programmable SSD. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 67–80, Broomfield, CO, October 2014. USENIX Association.

[43] Shin-Yeh Tsai and Yiying Zhang. Lite kernel RDMA support for datacenter applications. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 306–324, 2017.

[44] Ziye Yang, James R Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse 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. IEEE, 2017.

[45] Irene Zhang, Jing Liu, Amanda Austin, Michael Lowell Roberts, and Anirudh Badam. I'm not dead yet! the role of the operating system in a kernel-bypass era. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 73–80, 2019.