XRP: In-Kernel Storage Functions with eBPF

Yuhong Zhong[1], Haoyu Li[1], Yu Jian Wu[1], Ioannis Zarkadas[1], Jeffrey Tao[1], Evan Mesterhazy[1], Michael Makris[1], Junfeng Yang[1], Amy Tai[2], Ryan Stutsman[3], and Asaf Cidon[1]

[1]Columbia University, [2]Google, [3]University of Utah

Abstract

With the emergence of microsecond-scale NVMe storage devices, the Linux kernel storage stack overhead has become significant, almost doubling access times. We present XRP, a framework that allows applications to execute user-defined storage functions, such as index lookups or aggregations, from an eBPF hook in the NVMe driver, safely bypassing most of the kernel's storage stack. To preserve file system semantics, XRP propagates a small amount of kernel state to its NVMe driver hook where the user-registered eBPF functions are called. We show how two key-value stores, BPF-KV, a simple B^+-tree key-value store, and WiredTiger, a popular log-structured merge tree storage engine, can leverage XRP to significantly improve their throughput and latency.

1 Introduction

With the rise of new high performance memory technologies, such as 3D XPoint and low latency NAND, new NVMe storage devices can now achieve up to 7 GB/s bandwidth and latencies as low as 3 µs [11, 18, 23, 25]. At such high performance, the kernel storage stack becomes a major source of overhead impeding both application-observed latency and IOPS. For the latest 3D XPoint devices, the kernel's storage stack *doubles* the I/O latency, and it incurs an even greater overhead for throughput (§2.1). As storage devices become even faster, the kernel's relative overhead is poised to worsen.

Existing approaches to tackle this problem tend to be radical, requiring intrusive application-level changes or new hardware. Complete kernel bypass through libraries such as SPDK [77] allows applications to directly access underlying devices, but such libraries also force applications to implement their own file systems, to forgo isolation and safety, and to poll for I/O completion which wastes CPU cycles when I/O utilization is low. Others have shown that SPDK applications suffer from high average and tail latencies and severely reduced throughput when the schedulable thread count exceeds the number of available cores [49]; we confirm this in §6, showing that SPDK applications suffer a 3× throughput loss in these cases.

In contrast to these approaches, we seek a readily-deployable mechanism that can provide fast access to emerging fast storage devices that requires no specialized hardware and no significant changes to the application while working with existing kernels and file systems. To do this, we rely on BPF (Berkeley Packet Filter [62, 63]) which lets applications offload simple functions to the Linux kernel [8]. Similar to kernel bypass, by embedding application-logic deep in the kernel stack, BPF can eliminate overheads associated with kernel-user crossings and the associated context switches. Unlike kernel bypass, BPF is an OS-supported mechanism that ensures isolation, does not lead to low utilization due to busy-waiting, and allows a large number of threads or processes to share the same core, leading to better overall utilization.

The support of BPF in the Linux kernel makes it an attractive interface for allowing applications to speed up storage I/O. However, using BPF to speed up storage introduces several unique challenges. Unlike existing packet filtering and tracing use cases, where each BPF function can operate in a self-contained manner on a particular packet or system trace, a storage BPF function may need to synchronize with other concurrent application-level operations or require multiple function calls to traverse a large on-disk data structure. For example, network packet headers specify which flow they belong to, but individual storage blocks do not contain access-control information or metadata on how they fit in with the larger data structure they belong to.

To tackle these challenges, we design and implement XRP (eXpress Resubmission Path), a high-performance storage data path using Linux eBPF. XRP is inspired by XDP, the recent efficient Linux eBPF networking hook [27]. In order to maximize its performance benefit, XRP uses a hook in the NVMe driver's interrupt handler, thereby bypassing the kernel's block, file system and system call layers. This allows XRP to trigger BPF functions directly from the NVMe driver as each I/O completes, enabling quick resubmission of I/Os that traverse other blocks on the storage device.

The key challenge in XRP is that the low-level NVMe driver lacks the context that the higher levels provide. Those

layers contain information such as who owns a block (file system layer), how to interpret the block's data, and how to traverse the on-disk data structure (application layer).

Our insight is that many storage-optimized data structures that power real-world databases [10, 12, 19, 26, 40, 61, 65, 75] – such as on-disk B-trees, log-structured merge trees, and log segments – are typically implemented on a small set of large files, and they are updated orders of magnitude less frequently than they are read; we validate this in §3. Hence, we exclusively focus XRP on operations contained within one file and on data structures that have a fixed layout on disk. Consequently, the NVMe driver only requires a minimal amount of the file system mapping state, which we term the *metadata digest*; this information is small enough that it can be passed from the file system to the NVMe driver so it can safely perform I/O resubmissions. This allows XRP to safely support some of the most popular on-disk data structures.

We present a design and implementation of XRP on Linux, with support for ext4, which can easily be extended to other file systems. XRP enables the NVMe interrupt handler to resubmit storage I/Os based on user-defined BPF functions.

We augment two key-value stores with XRP: BPF-KV, a B^+-tree based key-value store that is custom-designed for supporting BPF functions, and WiredTiger's log-structured merge tree, which is used as one of MongoDB's storage engines [26]. With random 512 B object reads on BPF-KV with multiple threads using a B^+-tree that has three index levels on disk, XRP has 99%–191% higher throughput and 36%–71% lower p99 latency than io_uring. XRP also enables more efficient sharing of cores among applications than kernel bypass: it is able to provide 95% better p99 latency than SPDK with two threads sharing the same core. In addition, XRP is able to consistently improve WiredTiger's performance by up to 37% under YCSB [37].

We make the following contributions.

1. **New Datapath**. XRP is the first datapath that enables the use of BPF to offload storage functions to the kernel.
2. **Performance**. XRP improves the throughput of a B-tree lookup by up to 2.5× compared to normal system calls.
3. **Utilization**. XRP provides latencies that approach kernel bypass, but unlike kernel bypass, it allows cores to be efficiently shared by the same threads and processes.
4. **Extensibility**. XRP supports different storage use cases, including different data structures and storage operations (e.g., index traversals, range queries, aggregations).

2 Background and Motivation

In this section we show why the Linux kernel is becoming a primary bottleneck with fast NVMe devices, and provide a primer on BPF.

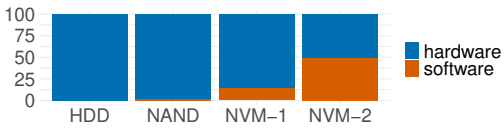


Figure 1: Kernel's latency overhead with 512 B random reads. HDD is Seagate Exos X16, NAND is Intel Optane 750 TLC NAND, NVM-1 is first generation Intel Optane SSD (900P), and NVM-2 is second generation Intel Optane SSD (P5800X).

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 P5800X.

2.1 Software is Now the Storage Bottleneck

New media like 3D Xpoint [1] and low-latency NAND [25], have led to new NVMe storage devices that exhibit single-digit µs latencies and millions of IOPS [11, 18, 23, 25]. The kernel storage stack is becoming a major performance bottleneck when accessing these devices. Figure 1 shows the percentage of time spent in the Linux stack when issuing a 512 B random read I/O on different storage devices. While the software overhead for the first generation of fast NVMe devices (first generation Intel Optane or Z-NAND) was non-negligible (~15%), with the latest generation of devices (Intel Optane SSD P5800X) the software overhead accounts for about half of the latency of each read request. The kernel's relative overhead will only get worse as storage devices become even faster.

Where is the time going? Table 1 shows the time spent in the different storage layers when issuing a random 512 B read with O_DIRECT on Optane P5800X. The experimental setup, which is used throughout the paper, is a server with 6-core i5-8500 3 GHz with 16 GB of memory, using Ubuntu 20.04, and Linux 5.8.0. We also disable processor C-states and turbo boost and use the maximum performance governor. The experiment shows that the most expensive layer is the file system (ext4), followed by the block layer (bio) and the kernel crossing, and that the total software overhead accounts for 48.6% of the average latency.

Why not just bypass the kernel? One approach to eliminate kernel overhead is to bypass it altogether [7, 60, 77, 78], leaving just the cost to post a request to the NVMe driver and the device's latency. However, kernel bypass is no panacea: each user is entrusted with full access to the device; they must also construct their own user space file systems [68, 69]. This means that there is no mechanism to enforce fine-grained isolation or to share capacity among different applications accessing the same device. In addition, there is no efficient way

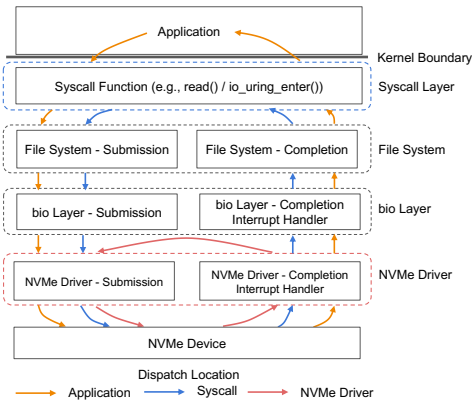


Figure 2: Dispatch paths for the application and two kernel hooks.

for user space applications to receive interrupts on I/O completions, so applications must directly poll on device completion queues to obtain high performance. Consequently, when I/O is not the bottleneck, cores cannot be shared among processes, which results in significant under-utilization and wasted CPU. Furthermore, when more than one polling thread shares the same processor, the CPU contention between them coupled with the lack of synchronization lead all polling threads to experience degraded tail latency and significantly lower overall throughput. Recent work has highlighted this issue [49] and we reproduce it in §6.2.

2.2 BPF Primer

BPF (Berkeley Packet Filter) is an interface that allows users to offload a simple function to be executed by the kernel. Linux's framework for BPF is called eBPF (extended BPF) [22]. Linux eBPF is commonly used for filtering packets (e.g., TCPdump) [5,6,27,47], load balancing and packet forwarding [5,17,24,55], tracing [2,4,45], packet steering [42], network scheduling [48,53] and network security checks [14]. Functions are verified by the kernel at install-time to ensure they are safe; for example, they are checked to make sure they do not contain too many instructions, unbounded loops, or accesses to out-of-bounds memory addresses [28]. After verification, which typically takes a few seconds or less, the eBPF functions can be called normally.

2.3 The Potential Benefit of BPF

BPF can be a mechanism for avoiding data movement between the kernel and user space in cases when a logical lookup requires a sequence of "auxiliary" I/O requests that generate intermediate data not needed directly by the application, such as in pointer-chasing workloads. For example, to traverse a B-tree index, a look-up at each level traverses the kernel's entire storage stack only to be thrown away by the application once it obtains the pointer to the next child node in the tree. Instead of a sequence of system calls from user space, each of the intermediate pointer lookups could be executed

	Latency	Speedup	Throughput	Speedup
User Space	78 µs	1×	109K IOPS	1×
Syscall Layer	68 µs	1.15×	130K IOPS	1.2×
NVMe Driver	40 µs	1.95×	276K IOPS	2.5×

Table 2: Average latency and throughput improvement with respect to user space when resubmitting I/O from the given layer; for kernel layers, resubmission is executed with a BPF function. Results shown for lookups on an on-disk B-tree of depth 10 [80].

by a BPF function, which would parse the B-tree node to find the pointer to the relevant child node. The kernel would then submit the I/O to fetch the next node. Chaining a sequence of such BPF functions could avoid the cost of traversing kernel layers and moving data to user space.

Other popular on-disk data structures, such as log-structured merge trees (LSM trees) [65], also have such auxiliary pointer lookups which can be accelerated using BPF functions. Other types of operations that would benefit from such an approach include range queries, iterators, and other types of aggregations (e.g., obtain the maximum or average value in a range of key-value pairs). In all of these cases, only a single result or a small subset of the objects that might be accessed by the storage system ultimately need to be returned to the application.

The BPF function that resubmits (dispatches) I/O in auxiliary I/O workloads could be placed at any layer of the kernel. Figure 2 shows the I/O paths for both normal user space dispatch and for two possible locations of BPF resubmission hooks: in the syscall layer and in the NVMe driver. Zhong et al. [80] compared the performance improvement from a resubmission hook in both locations on workloads with auxiliary I/O by measuring the speedup of lookup queries on an on-disk B-tree of depth 10. The baseline for comparison is reading I/O through the `read` system call. Table 2 summarizes the results.

Best Case Acceleration. Dispatching the I/O requests from the NVMe driver provides a significant latency reduction (up to 49%) and corresponding speedup (up to 2.5×), since it bypasses almost the entire kernel software stack. On the other hand, as expected, issuing the BPF functions from the syscall dispatch layer only provides a maximum speedup of 1.25×, since the requests only benefit from eliminating kernel boundary crossings, which only account for 5-6% of the kernel overhead (Table 1). After reaching CPU saturation, the computation savings of reissuing the submissions from the NVMe driver translate into throughput improvements of 1.8x-2.5x, depending on the number of threads in the workload [80].

Placing an eBPF hook *anywhere* in the kernel may improve throughput between 1.2–2.5×. However, pushing the I/O dispatching as close as possible to the storage device dramatically improves the performance of a traversal. Hence *to obtain the highest possible speedup, XRP's resubmission hook should reside in the NVMe driver.*

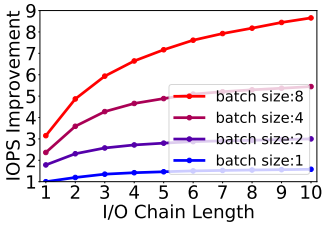


Figure 3: Single-threaded lookups with io_uring syscall, using NVMe driver hook.

What about io_uring? io_uring is a new Linux system call framework [9] that allows processes to submit batches of asynchronous I/O requests, which amortizes user-kernel crossings. However, each I/O submitted with io_uring still passes through all the layers shown in Table 1, so each individual I/O still incurs the full storage stack overhead. In fact, BPF I/O resubmissions are largely complementary to io_uring: io_uring can efficiently submit batches of I/Os that trigger different I/O chains managed by BPF in the kernel.

Figure 3 shows throughput improvements when using io_uring with a BPF hook in the NVMe driver. I/O Chain Length denotes the number of I/Os resubmitted. For example, a lookup on a B-tree of depth 3 would have an I/O Chain Length of 2. Figure 3 shows that BPF can increase throughput with respect to io_uring by up to 3× for small batch sizes and up to 9× as batch sizes increase.

In summary, BPF can benefit both legacy read and io_uring system calls. By placing the hook in the kernel NVMe driver, BPF may increase throughput of legacy I/O by up to 2.5× and of io_uring by up to 9×.

3 Design Challenges and Principles

As shown in the previous section, I/O resubmission must occur as close to the device as possible in order to reap the greatest benefits. In the NVMe software stack, this is the NVMe interrupt handler. However, executing the resubmissions from within the NVMe interrupt handler, which lacks the context of the file system layer, introduces two major challenges.

Challenge 1: address translation and security. The NVMe driver has no access to file system metadata. In the example of an index traversal, XRP issues a read I/O to a particular block and executes a BPF function that would extract the offset of the next block it would like to query. However, this offset is meaningless to the NVMe layer, since it cannot tell which physical block the offset corresponds to without having access to the file's metadata and extents. Even if the application developer made the effort to embed physical block addresses to avoid the translation of the file system offset, which would be burdensome, the BPF function could access *any* block on the device, including blocks that belong to a file that the user does not have permissions to access.

Challenge 2: concurrency and caching. It is challenging to enable concurrent reads and writes issued from the file system with XRP. A write issued from the file system will only be reflected in the page cache, which is not visible to XRP. In addition, any writes that modify the layout of the data structure (e.g., modify the pointers to the next block) that are issued concurrently to read requests could lead XRP to accidentally fetch the wrong data. Both of these could be addressed by locking, but accessing locks from within the NVMe interrupt handler may be expensive.

Observation: most on-disk data structures are stable. Both of these challenges would make it difficult to implement arbitrary concurrent BPF storage functions. However, we make the observation that the file extents of many storage engines (e.g., LSM trees and B-trees) remain relatively stable. Some data structures simply do not modify on-disk storage structures in-place. For example, once an LSM tree writes its index files (called SSTables) to disk, they are immutable until they are deleted [19, 26, 40]. Similarly, even though some on-disk B-tree index implementations support in-place updates, their file extents remain stable for long periods of time. We verify this in a 24-hour YCSB [37] (40% reads, 40% updates, 20% inserts, Zipfian 0.7) experiment on MariaDB running TokuDB [12], which uses a fractal tree (an on-disk B-tree variant) as its lookup index. 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, making it possible to safely cache these extents. We also make the observation that in all of these storage engines, the indices are stored on a small number of large files, and each index does not span multiple files.

Design principles. These observations and experiments inform the following design principles.

- **One file at a time.** We initially restrict XRP to only issue chained resubmissions on a single file. This greatly simplifies address translation and access control, and it minimizes the metadata that we need to push down to the NVMe driver (the *metadata digest*, §4.1.3).
- **Stable data structures.** XRP targets data structures, whose layout (i.e. pointers) remain immutable for a long period of time (i.e. seconds or more). Such data structures include the indices used in many popular commercial storage engines, such as RocksDB [40], LevelDB [19], TokuDB [19] and WiredTiger [26]. Since the cost of implementing locks in the NVMe layer is high, we also initially do not plan to support operations that require locks during the traversal or iteration of data structures.
- **User-managed caches.** XRP does not interface with the page cache, so XRP functions cannot safely be run concurrently if blocks are buffered in the kernel page cache. This constraint is acceptable since popular storage engines often implement their own user space caches [12, 26, 35, 40]; Commonly they do this to fine-tune their caching and

prefetching policies and to cache data in an application-meaningful way (e.g., cache key-value pairs or database rows instead of physical blocks).

- **Slow path fallback.** XRP is best-effort; if a traversal fails for some reason (e.g., the extent mappings become stale), the application must retry or fall back to dispatching the I/O requests using user space system calls.

4 XRP Design and Implementation

This section presents XRP's design and implementation with Linux eBPF and ext4. We describe the kernel modifications that enable XRP's resubmission logic in the interrupt handler, and how applications are modified to use XRP. We also discuss XRP's synchronization and scheduling limitations.

4.1 Resubmission Logic

The core of XRP augments the NVMe interrupt handler with resubmission logic that consists of a BPF hook, a file system translation step, and the construction and resubmission of the next NVMe request at the new physical offsets (Figure 4). Our modifications to the Linux kernel consist of ~900 lines of code: ~500 lines for the BPF hook and the changes to the NVMe driver, ~400 lines for the file system translation step.

When an NVMe request completes, the device generates an interrupt that causes the kernel to context switch into the interrupt handler. For each NVMe request that is completed in the interrupt context, XRP calls its associated BPF function, the address of which is stored in a field in the NVMe request struct (`bpf_func_0` in Figure 4). After calling the BPF function, XRP invokes the metadata digest, which is usually a digest of file system state that enables XRP to translate the logical address of the next resubmission. Finally, XRP prepares the next NVMe command resubmission by setting the corresponding fields in the NVMe request, and it appends the request to the NVMe submission queue (SQ) for that core.

For a particular NVMe request, the resubmission logic is called as many times as necessary for subsequent completions as determined by the specific BPF function registered with the NVMe request. For example, for traversing a tree-like data structure, the BPF function would resubmit I/O requests for branch nodes and end resubmission whenever a leaf node is found. In our current prototype there is no hard limit on the number of resubmissions before the completion returns control to the application; such a limit would be necessary to prevent unbounded execution. BPF function contexts are per-request, while the metadata digest is shared across *all* invocations of the interrupt handler across all cores. Safe concurrent access to the metadata digest relies on read-copy-update (RCU) (§4.1.3).

4.1.1 BPF Hook

XRP introduces a new BPF type (`BPF_PROG_TYPE_XRP`) with the signature shown in Listing 1 – any BPF function that matches the signature can be called from the hook. § 5

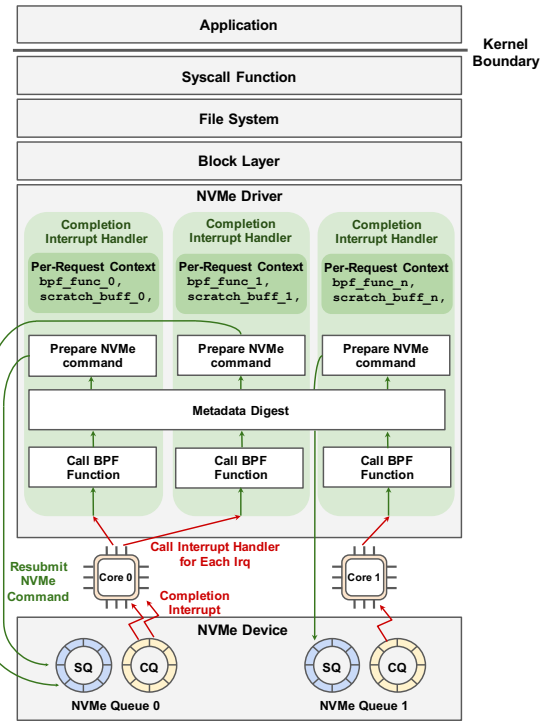


Figure 4: XRP architecture.

```
struct bpf_xrp {
  // Fields inspected outside BPF
  char *data;
  int done;
  uint64_t next_addr[16];
  uint64_t size[16];
  // Field for BPF function use only
  char *scratch;
}
uint32_t BPF_PROG_TYPE_XRP(struct bpf_xrp *ctxt);
```

Listing 1: Signature of BPF programs that can be loaded by XRP.

presents several concrete BPF functions matching this signature that are used in our applications. For example, for on-disk data structure traversal, the BPF function typically contains logic to extract the next offset to fetch from the block.

`BPF_PROG_TYPE_XRP` programs require a context with five fields, categorized into fields that are inspected or modified by the BPF caller (resubmission logic in the interrupt handler), and fields that should be private to the BPF function. Fields that are accessed externally include `data`, which buffers data read from the disk (e.g., a B-tree page waiting to be parsed by the BPF function). `done` is a boolean that notifies the resubmission logic whether to return to the user or continue resubmitting I/O requests. `next_addr` and `size` are arrays of logical addresses and their corresponding sizes that indicate the next logical addresses for resubmission.

In order to support data structures with fanout, multiple

next_addr values can be supplied. By default we limit fanout to 16; on-disk data structures align their components to small multiples of device pages, so we haven't encountered a need for higher fanout per completion. For example, chained hash table buckets are likely implemented as a chain of individual physical pages and the elements of an on-disk linked list are likely implemented at the granularity of physical pages. Setting a corresponding size field to zero issues no I/O.

scratch is a scratch space that is private to the user and the BPF function. It can be used to pass the parameters from the user to the BPF function. Also, the BPF function can use it to store intermediate data in between I/O resubmissions and to return data to the user. For example, in the first BPF invocation, the application can store a search key in the scratch buffer so that the BPF function can compare it with the keys in the disk block in order to find the next offset. When the I/O chain reaches the leaf node of the B-tree, the BPF function then places the key-value pair in the scratch buffer to return it back to the application. For simplicity, we assume that the size of the scratch buffer is always 4 KB. We find that a 4 KB scratch buffer is sufficient to support a BPF function for a production key-value store (§5). BPF functions can also use BPF maps to store more data if their intermediate data cannot fit into the scratch buffer. Each BPF context is private to one NVMe request, so no locking is needed when working with BPF context state. Letting the user supply a scratch buffer (instead of using BPF map) avoids the overhead of processes and functions having to call bpf_map_lookup_elem to access the scratch buffer.

4.1.2 BPF Verifier

The BPF verifier ensures memory safety by tracking the semantics of the value stored in each register [13]. A valid value can either be a scalar or a pointer. SCALAR_TYPE represents a value that cannot be dereferenced. The verifier defines various pointer types; most of them include extra constraints beyond the no out-of-bound access requirement. For example, PTR_TO_CTX is the type for the pointer to a BPF context. It can only be dereferenced using a constant offset so the verifier can identify which context field a memory operation accesses. Each BPF function type also defines a callback function is_valid_access() to perform additional checks on context accesses and to return the value type of the context field. PTR_TO_MEM describes a pointer referring to a fixed-size memory region. It supports dereferencing using a variable offset as long as the access is always within bounds. The data and scratch fields of the BPF_PROG_TYPE_XRP context are PTR_TO_MEM and the rest are SCALAR_TYPE. We augment the verifier to allow the BPF_PROG_TYPE_XRP's is_valid_access() callback to pass the size of the data buffer or scratch buffer to the verifier so that it can perform the boundary check. We discussed our proposed modification to the verifier with the Linux eBPF maintainers, and they think it is sensible.

```
void update_mapping(struct inode *inode)
void lookup_mapping(struct inode *inode,
                    off_t offset, size_t len,
                    struct mapping *result)
```

Listing 2: Metadata digest: XRP exposes an interface to share logical-to-physical-block mappings between the file system and the IRQ handler.

4.1.3 The Metadata Digest

In the conventional storage stack, the logical block offsets in on-disk data structures are translated by the file system in order to identify the next physical page to read. This translation step also enforces access control and security, preventing reading in regions that are not mapped to the open file. In XRP, the next logical address for a lookup is given by the next_addr field after the BPF call. However, translating this logical address to a physical address is challenging since the interrupt handler has no notion of a file and does not perform physical address translation.

To solve this, we implement the metadata digest, a thin interface between the file system and the interrupt handler that lets the file system share its logical-to-physical-block mappings with the interrupt handler, enabling safe eBPF-based on-disk resubmissions. The metadata digest consists of two functions (Listing 2). The update function is called within the file system when the logical-to-physical mapping is updated. The lookup function is called within the interrupt handler; it returns the mapping for a given offset and length. The lookup function also enforces access control by preventing BPF functions from requesting resubmissions for blocks outside of the open file. The inode address of the open file is passed to the interrupt handler in order to query the metadata digest. If an invalid logical address is detected, XRP returns to user space immediately with an error code. The application can then fall back to use the normal pread() to attempt its request again.

These two functions are specific to each file system, and even for a particular file system, there may be multiple ways to implement the metadata digest, presenting a tradeoff between ease of implementation and performance. For example, in our implementation for ext4, the metadata digest consists of a cached version of the extent status tree, which stores the physical-to-logical block mappings. This cached tree is accessed by the update and lookup function of the interface, and it uses read-copy-update (RCU) for concurrency control. RCU enables the lookup function to be lockless and fast (96 ns on average).

To keep the cached tree up-to-date with the main extents in ext4, the update function is called in two places in ext4: whenever extents are inserted or removed from the main extent tree. To prevent a race condition where an extent is modified while there is an inflight read on it, we maintain a version number for each extent to track its changes. After data is read, but before it is passed to the BPF function, a second metadata

digest lookup is performed. If the corresponding extent no longer exists or its version number has changed, XRP will abort the operation. Since application-level synchronization usually prevents concurrent modifications and lookups on the same region of a file at the same time, version mismatches should only occur if the application is buggy or malicious.

An alternative, simpler implementation of the metadata digest for ext4 could simply pass through to existing update and access functions of the extent tree in ext4. In this case, the update function would be a no-op, because ext4 already keeps its extent tree up-to-date. However, such an implementation would be much slower on the lookup path, because the extent lookup function in ext4 acquires a spinlock, which would be prohibitively expensive in the interrupt handler.

For now, XRP only supports the ext4 file system, but the metadata digest can be easily implemented for other file systems. For example, in f2fs [59], logical-to-physical-block mappings are stored in the node address table (NAT). Similar to the ext4 implementation, an implementation of its metadata digest could cache a local copy of the NAT, which would be consulted in `lookup_mapping`. Then `update_mapping` would need to be called anywhere in f2fs where the NAT is updated.

4.1.4 Resubmitting NVMe Requests

After looking up the physical block offsets, XRP prepares the next NVMe request. Because this logic occurs in the interrupt handler, to avoid the (slow) `kmalloc` calls needed to prepare NVMe requests, XRP reuses the existing NVMe request struct of the just-completed request. XRP simply updates the physical sector and block addresses of the existing NVMe request to the new offsets derived from the mapping lookup.

While `struct bpf_xrp` supports a maximum fanout of 16, in the current implementation a resubmitted I/O request can only fetch as many physical segments as the initial NVMe request. For example, if an initial NVMe request only fetches a single block, then all subsequent resubmissions for that request can only fetch a single physical segment. During a resubmission chain, if the BPF call returns multiple valid addresses in `next_addr`, XRP will abort the request. This limitation can be worked around by allocating and setting up 16 dummy NVMe commands in the first NVMe request so that subsequent resubmissions can express fanout if necessary.

4.2 Synchronization Limitations

BPF currently only supports a limited spinlock for synchronization. The verifier only allows BPF programs to acquire one lock at a time, and they must release the lock before returning. Also, user space applications do not have direct access to these BPF spinlocks. Instead, they must invoke the `bpf()` syscall; the syscall can read or write the lock-protected structure while holding the lock for the duration of that operation. Hence, complex modifications that require synchronizing across multiple reads and writes cannot be accomplished in user space.

```
int bpf_xrp(int cmd, union bpf_attr *attr,
            unsigned int size);
int read_xrp(int fd, void *buf, size_t count,
             off_t offset, int bpf_fd,
             void *scratch);
```

Listing 3: The XRP application interface consists of a syscall to register BPF functions with the kernel and a read syscall that requests that a BPF function be used. `bpf_xrp` has the same signature as the `bpf` syscall. `bpf_xrp` returns a file descriptor for the registered function, which must be passed to `read_xrp`. `read_xrp` adds two arguments to the standard `pread` [20] syscall: this file descriptor and a pointer to a 4 KB scratch space that is passed to the BPF context.

4.3 Interaction with Linux Schedulers

Process scheduler. Interestingly, we observed that a microsecond-scale storage device like Optane SSD interferes with Linux's CFS when multiple processes share the same core, *even when all I/O is issued from user space*. For example, in the case where an I/O-heavy and compute-heavy process share the same core, the I/O interrupts generated by the I/O-heavy process will be handled in the timeslice of the compute-heavy process. This may cause the compute-heavy process to be starved of CPU; in the worst case in our experiments, the compute-heavy process only received about 34% of what would be a "fair" allocation of CPU time. We experimentally verified this does not occur when using a slower storage device, which generates interrupts much less frequently. While XRP exarcerbates this problem by generating chains of interrupts, this issue is not specific to eBPF, and can also be caused by network-driven interrupts [54]. We leave this problem for future work.

I/O scheduler. XRP bypasses Linux's I/O scheduler, which sits at the block layer. However, the noop scheduler is already the default I/O scheduler for NVMe devices, and the NVMe standard supports arbitration at hardware queues if fairness is a requirement [16].

5 Case Studies

To use XRP, applications use the interface shown in Listing 3. Applications call `bpf_xrp` to register a BPF function of type `BPF_PROG_TYPE_XRP` to be offloaded in the driver and call `read_xrp` to apply a specific BPF function to the read request. Applications can register multiple BPF functions with XRP. For example, a database can register a function for filtering and calculating aggregations from values on-disk and a function for `GET` point lookups. XRP allows the application to load multiple BPF functions into the kernel and to specify the BPF function to use in each `read_xrp` syscall. We present two case studies on how applications should be modified to use XRP.

5.1 BPF-KV

We built a simple key-value store, called BPF-KV, with which we can evaluate XRP against other baselines: Linux's syn-

chronous and asynchronous system calls and kernel bypass (SPDK [77]). BPF-KV is designed to store a large number of small objects and to provide good read performance even under uniform access patterns. BPF-KV uses a B$^+$-tree index to find the location of objects, and the objects themselves are stored in an unsorted log. For simplicity, BPF-KV uses fixed-sized keys (8 B) and values (64 B). The index and the log are both stored in one large file. The index nodes use a simple page format with a header followed by keys followed by values. Leaf nodes contain a file offset pointing to the next leaf node, enabling efficient index traversal for range queries and aggregation. Object sizes are fixed, so updates occur in-place in the unsorted log. Newly inserted items are appended to the log; their index is initially stored in an in-memory hash table. Once the hash table fills, BPF-KV merges it with the on-disk B$^+$-tree file.

Caching. BPF-KV implements a user space DRAM cache for index blocks and objects. To reduce the number of I/Os it needs to issue for lookups, BPF-KV caches the top k levels of the B$^+$-tree index. With a sufficiently large number of objects, it is not possible to fit the entire index in the cache. Consider the case where BPF-KV is used to store 10 billion 64 B objects. In BPF-KV's index, each node is 512 B (matching the access granularity of the Optane SSD); hence, the tree has a fanout of 31 (i.e. each internal node can store pointers to 31 children). Therefore, 10 billion objects would require an index with 8 levels. Fitting 6 index levels in DRAM is expensive and would require 14 GB, while fitting 7 levels or more becomes prohibitively expensive (437 GB of DRAM or more). So, to support a large number of keys, BPF-KV would require at the minimum 3-4 I/Os from storage for each lookup, including a final I/O to fetch the actual key-value pair from disk. Also note that having a hard memory budget for caching the index is common in many real-world key-value stores (e.g., RocksDB [41], DocumentDB [73], SplinterDB [36], TokuDB [12]), since the index cache often competes with other parts of the system that need memory, such as filters and the object cache.

BPF-KV also maintains a least recently used (LRU) object cache of the most popular key-value pairs. Before looking up an object on disk, BPF-KV first checks whether it is stored in the object cache. If not, it checks whether it is indexed in the in-memory hash table. If the item is not found in the in-memory hash table, it looks up the object by accessing the first k cached levels of the index. Once it encounters an index node that is not cached, it completes the index and the final lookup on disk.

To find an object without XRP, BPF-KV traverses the B-tree until the desired value is found using an I/O request per level. For example, if the index contains 7 levels and the first 3 are cached and read from DRAM, then the traversal will issue 4 I/Os to navigate the rest of the tree, followed by a final I/O to fetch the object from the log.

```
struct node {
  uint64_t num; uint64_t type;
  uint64_t key[31]; uint64_t ptr[31];
};
uint32_t bpfkv_bpf(struct bpf_xrp *ctxt) {
  uint64_t key = *((uint64_t*)ctxt->scratch);
  struct node *n = (struct node *)ctxt->data;
  if (n->type == LEAF_NODE) {
    ctxt->done = true;
    return 0;
  }
  for (int i = 0; i < n->num; i++)
    if (key < n->key[i]) break;
  ctxt->done = false;
  ctxt->next_addr[0] = n->ptr[i - 1];
  ctxt->size[0] = 512;
  return 0;
}
```

Listing 4: BPF function for BPF-KV.

BPF function. Listing 4 shows the BPF function used in BPF-KV to lookup a key-value pair. We omit the code to handle the final lookup in the log for simplicity. `struct node` defines the layout of B$^+$-tree index nodes whose size is 512 B. The BPF function `bpfkv_bpf` first extracts the target key stored in the scratch buffer, and then it linearly searches the slots in the current node to find the next node to read.

Interface modifications. We replace `read` calls with `read_xrp`. Before calling into `read_xrp`, BPF-KV first allocates a buffer for the scratch space and calculates the offset at which to start the lookup.

Range queries. BPF-KV supports range queries returning a variable number of objects. We implement a BPF function that runs as a state machine, allowing the operation to be suspended and resumed when objects are returned to the application for processing. The BPF function state, including the beginning and end of the range, and the retrieved objects, are stored in the scratch space (up to 32 72-byte key-value pairs). On the initial invocation, the function traverses to the leaf node that contains the starting key. Once the first key in the range is found, the function stores the leaf node in the scratch space and requests the block containing the corresponding value. On the next BPF invocation, the function stores the value in the scratch space and it continues the index scan on the cached leaf node. When the leaf node has been read completely, the function submits a request for the next leaf node using the node's next-leaf file offset. The function returns to the application in three cases: 1) the function reaches a key past the end of the range; 2) the function reaches the end of the index; 3) the function fills the scratch space with values read from the log. In the last case, the application can process the values and re-invoke the BPF function with the

range query state, allowing the range query to resume from where it left off.

Aggregations. BPF-KV also supports aggregation operations, such as `SUM`, `MAX` and `MIN`. We implement these operations on top of the BPF range query function by setting a bit that causes the function to perform the corresponding aggregation instead of returning the individual values. Since aggregation queries return a single answer, storing values in the scratch space does not limit the number of I/O resubmits the BPF function can request.

5.2 WiredTiger

WiredTiger is a popular key-value store that is the default backend for MongoDB [26]. We use it as a case study since it is a relatively simple and open key-value store that is used in production. WiredTiger provides an option to use an LSM tree where data is split into different levels; each level contains a single file. Each file uses a B-tree index with the key-value pairs embedded in the tree's leaf nodes. The files are read-only; updates and inserts are written into a buffer in memory. When the buffer is full, the data is written out in a new file. We configure the B-tree page size to be the same as our Optane SSD's block size (512 B). Our modification to WiredTiger is around 500 lines of code, which mainly consist of buffer allocation, extending function signatures and wrapping the XRP syscall. XRP helps accelerate reads that are serviced from disk, and it does not affect updates or inserts, which are always absorbed by WiredTiger's in-memory buffer.

BPF function. To use XRP, WiredTiger installs a BPF function similar to the one shown in Listing 4. The difference is in order to find the next look-up address from the current page, the BPF function contains a port of WiredTiger's B-tree page parsing code. This parsing logic replaces the `for` loop in Listing 4.

The WiredTiger BPF function also makes several modifications to make the BPF program compile correctly and pass the BPF verifier. The modifications mainly consist of adding bounds on loops to avoid infinite loops, masking pointers to eliminate out-of-bound access, and initializing local variables to prevent access to uninitialized registers. We also use the BPF function-by-function verification feature [3] to break a complex function into several simple sub-functions. This allows BPF functions to be verified independently, so the functions that have been verified do not need another round of verification when being called by other functions. The function-by-function verification feature also supports more complex BPF programs without exceeding the verifier's restrictions on function length.

Caching. WiredTiger maintains a least recently used (LRU) cache for its B-tree internal pages and leaf pages. When looking up a new key-value pair, WiredTiger caches the entire lookup path including the leaf page in the cache. In order to comply with WiredTiger caching semantics, the BPF function described in the previous section also returns all traversed pages so that WiredTiger can cache them. The BPF function stores traversed pages in the scratch buffer of its context. When the scratch buffer is exhausted, the BPF function will stop resubmitting requests and return to user space immediately. After WiredTiger adds those pages into its cache, it will call `read_xrp` again to continue the lookup starting at the previous page. Since we set the size of the scratch buffer to 4 KB, a BPF function can store up to 6 traversed 512 B pages in the scratch buffer, which leaves room for necessary metadata such as the search key.

Average Lookup Latency (µs)				
# Ops	SPDK	io_uring	`read()`	XRP
1	5.2	13.3	14.5	10.2
2	7.8	19.9	21.2	13.4
3	11.3	27.5	28.6	17.0
4	14.2	34.5	35.3	20.4
5	17.2	41.5	42.3	23.8
6	20.2	48.4	49.3	27.5

Table 3: Average latency of a random key lookup with BPF-KV as a function of the depth of the B^+-tree stored on-disk. # ops is the number of index I/Os per lookup.

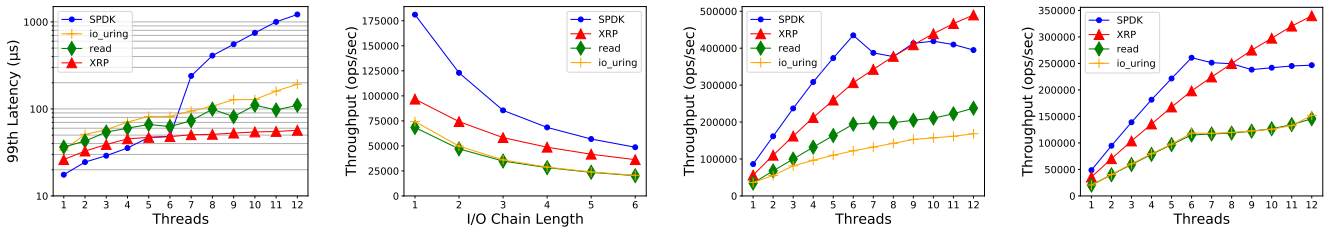
Interface modifications. To integrate WiredTiger with XRP, we replace normal read calls with `read_xrp`. `read_xrp` is called when the next page is not in the cache and needs to be read from disk. The eviction policy of WiredTiger enforces that only the pages without any cached children pages can be evicted, so any uncached page will not have cached descendants. Therefore, it is safe to call `read_xrp` to read all of the remaining path from disk without checking the application-level cache again. If `read_xrp` fails for any reason, WiredTiger falls back to the normal lookup path. We allocate a data and scratch buffer for each WiredTiger session to avoid the overhead of allocating and freeing buffers for every request. WiredTiger sessions synchronously process one request at a time, which avoids concurrency issues.

6 Evaluation

In this section we seek to answer the following questions:

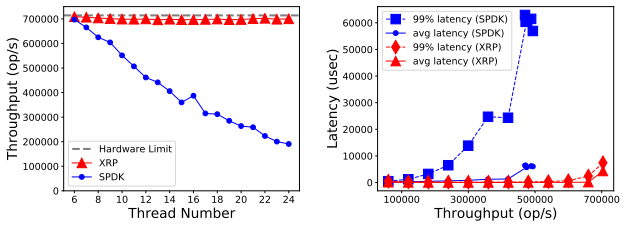
1. What are the overheads of using BPF for storage (§6.1)?
2. How does XRP scale to multiple threads (§6.2)?
3. What types of operations can XRP support (§6.3)?
4. Can XRP accelerate a real-world key-value store (§6.4)?

Experimental setup. All experiments are conducted on a 6-core i5-8500 3 GHz server with 16 GB of memory, using Ubuntu 20.04, and Linux 5.12.0 with an Intel Optane 5800X prototype. All experiments use O_DIRECT, turn off hyper-threading, disable processor C-states and turbo boost, and use the maximum performance governor. We use WiredTiger 4.4.0 in the experiments.



(a) (Log scale) 99th-percentile latency, index depth 3. **(b)** Single-threaded throughput for varying index depth. **(c)** Multi-threaded throughput, index depth 3. **(d)** Multi-threaded throughput, index depth 6.

Figure 5: Performance of XRP and SPDK against `read()` with BPF-KV with random key lookups and closed-loop load generator.



(a) Scalability of XRP and SPDK. **(b)** Latency-throughput graph of XRP and SPDK with 12 threads.

Figure 6: XRP vs. SPDK with open-loop load generator.

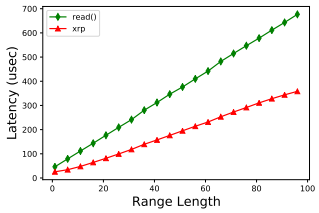


Figure 7: Average read latency of BPF-KV with XRP vs. `read()` when performing a range query over a varying number of objects.

Baselines. We compare the following configurations: (a) XRP, (b) SPDK (a popular kernel-bypass library), (c) standard `read()` system calls, and (d) standard io_uring system calls.

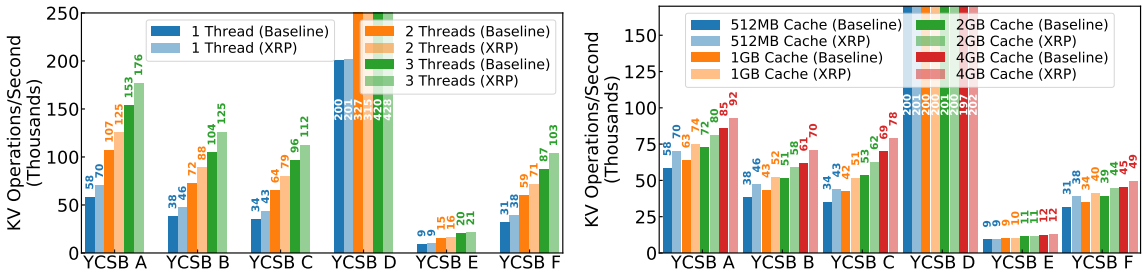
6.1 BPF-KV

Latency. To answer the first evaluation question, we measure the performance of BPF-KV on a benchmark that performs a million read operations with keys drawn randomly with uniform probability. The experiment varies the number of levels of the tree that are stored on-disk. In this subsection, we disable caching of data objects and index nodes to focus on the overhead of looking up on-disk items. The measured average latency is shown in Table 3. The leftmost column represents the number of chained I/Os that are required to lookup the key in the index (including the final data lookup). For example, if the number of operations is 4, then BPF-KV is configured with an on-disk tree of depth 4, and it also needs to issue one more I/O to fetch the key-value pair from the log.

There are a few takeaways from this experiment. First, XRP improves latency over `read()`, because XRP saves one or more storage layer traversals when it traverses the index. Indeed, one can see that XRP's latency increases by about 3.2-3.5 µs for each additional I/O operation, which is almost identical to the device's latency (Table 1). This means that XRP achieves close to optimal latency for resubmitted requests. The same is true for io_uring: in the case of a single thread, `read()` and io_uring are almost equivalent. Second, SPDK exhibits better latency than XRP since XRP must pass through the kernel's storage stack once to initiate the index traversal, while SPDK completely bypasses the kernel. Nonetheless, XRP's marginal added latency when the depth of the B$^+$-tree is increased is nearly identical to SPDK's (3.2 µs-3.5 µs). For this reason, in the case of a 6-level index, XRP is only 17% slower than SPDK. Importantly, XRP achieves this without resorting to polling. This means that, unlike with SPDK, processes can continue to use CPU cores efficiently for other work; XRP's use of CPU time is limited to what is specifically needed to resubmit I/Os in the background and to keep I/O device utilization high.

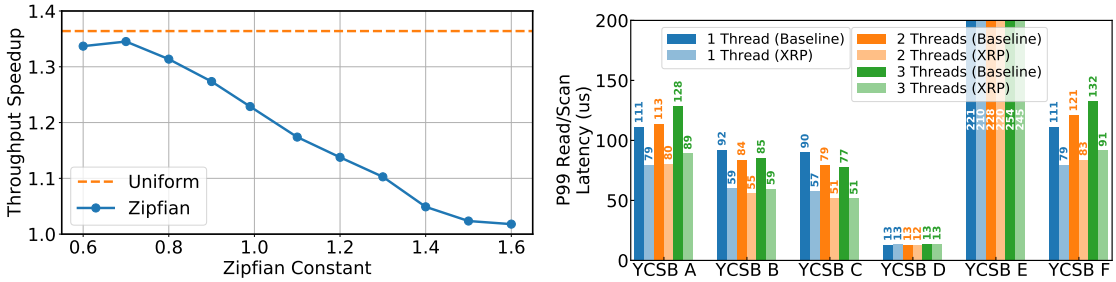
Figure 5a presents the 99th-percentile latency of XRP. When running with a single thread, similar to the average latency results, XRP reduces 99th-percentile latency by up to 30% against `read()` and io_uring. Note that our experiment runs as a closed loop, so XRP is running at a higher throughput than `read()` and io_uring. At identical throughput XRP would show additional improvement over these baselines. Interestingly, when the number of threads exceeds the number of cores (6), SPDK's p99 latency increases significantly. This is due to the fact that with SPDK all threads are busy-polling, and cannot effectively share the same core with other threads.

Throughput. Figure 5b shows the throughput of XRP. As expected, as the index depth increases, XRP's speedup is greater compared to standard system calls. Figures 5c and 5d show the throughput speedups with a varying number of threads with an index of depth 3 and 6, respectively. Both figures show the speedup of XRP relative to issuing standard system calls remains constant even as I/O and XRP BPF functions are scaled across several cores. Once again, XRP provides equal to or higher throughput compared to SPDK once the number of threads is 8 or higher.



(a) **(b)**

Figure 8: Throughput of reads/scans in WiredTiger with **(a)** varying client threads with a 512 MB cache and **(b)** varying cache size.



(a) **(b)**

Figure 9: (a) Throughput speedup of WiredTiger on YCSB C with a varying Zipfian constant and with a uniform distribution. **(b)** 99th-percentile latency of reads/scans in WiredTiger with varying number of threads with 512 MB cache.

6.2 Thread Scaling

Since storage applications often use a large number of concurrent threads that access I/O devices, for example in order to process concurrent requests and to perform background garbage collection [12, 19, 26, 40], XRP needs to be able to provide good tail latency and throughput under a large number threads. We analyze how XRP scales as a function of the number of threads and compare it to SPDK. We run an open loop experiment, where the amount of load matches the maximum bandwidth of the Intel device (5M IOPS for 512 B random reads). Figure 6a compares the throughput XRP (integrated with io_uring) compared with SPDK with BPF-KV using 6 on-disk levels, where each thread represents a different tenant. Two major observations are: 1) when using 6 working threads (the number of CPU cores on the machine) both SPDK and XRP can achieve a throughput close to the hardware limit (the red dashed line); 2) once the thread count exceeds the CPU cores, SPDK's throughput steadily decreases while XRP still provides stable throughput. SPDK's throughput collapse stems from its polling-based approach; SPDK threads never yield, leaving scheduling up to Linux's CFS which works in coarse 6 ms timeslices. However, idle XRP threads will voluntarily yield the CPU to busy threads, so more CPU cycles are spent on actual work. Figure 6b presents the throughput-latency relationship under 12 working threads as a function of the load. With more threads than CPU cores, both average and tail latencies also increase more significantly in SPDK, as each thread waits longer to be scheduled than in XRP.

6.3 Range Query

Figure 7 compares the average latency of running a range query with XRP against performing the query with `read()` system calls. In both cases the range query performs a single index traversal to find the first object, and traverses the leaf nodes of the index to find the address of subsequent objects. Even though the XRP range query can only retrieve 32 objects per syscall, the results show this adds negligible overhead. XRP's performance speedup remains relatively constant as a function of the length of the aggregation, since XRP performs only one storage stack traversal for every 32 values retrieved.

6.4 WiredTiger

To understand whether XRP can benefit a real-world database, we evaluate the performance of WiredTiger with and without XRP on YCSB [37]. We run the different YCSB workloads so that their runtime takes more or less the same time: YCSB A, B, C and E use 10M operations, D uses 50M operations and E uses 3M. The baseline WiredTiger uses `pread()` to read B-tree pages, while the WiredTiger with XRP uses `read_xrp()`. We populate the database with 1 billion key-value pairs and set the size of both key and value to 16 B. The total size of the database is 46 GB. WiredTiger runs eviction threads to evict pages when its cache usage is close to full, and we set the number of eviction threads to 2.

Throughput. Figure 8 shows the total throughput of WiredTiger with different cache sizes and different numbers of client threads. We configure WiredTiger with 512 MB,

1 GB, 2 GB, and 4 GB cache sizes to ensure that WiredTiger can cache at least 1% of its database while not exhausting all the available memory on the machine. We run up to 3 client threads to avoid context switches. The results show that XRP speeds up most workloads consistently by up to 1.25×. The throughput improvements are mostly affected by the cache size. The speedup generally goes down when the cache size becomes larger. In general, XRP provides a lower speedup on WiredTiger than on BPF-KV, because WiredTiger is less optimized than BPF-KV for reading from fast NVM storage, and only spends 63% of its total time on I/O. In particular, XRP does not provide significant improvements on YCSB D and YCSB E. This is because YCSB D follows a latest distribution where the newly inserted items are the most popular ones. Since new inserts are always written into in-memory buffers, most read operations hit those buffers in YCSB D. On the other hand, YCSB E only has inserts and scans. WiredTiger supports scans via an iterator interface, which only looks up one key-value pair at a time. XRP can only benefit the lookup of the first key-value pair of a scan operation, since the rest of the key-value pairs either reside on the same leaf node or require only one additional I/O to fetch the next leaf node.

To study the effect of access distribution on XRP, we run YCSB C with a varying Zipfian constant and with a uniform distribution. Figure 9a shows that XRP's benefit decreases when the Zipfian constant becomes larger (i.e. , the distribution is more skewed) because of the increased cache hit ratio. Note that skews greater than 0.99 represent very high skew levels. We also see that the throughput gain of XRP is lower than that of BPF-KV on the uniform YCSB C.

Tail latency. We measure the tail read latency of WiredTiger with and without XRP under a fixed load: 20 kop/s per client thread for YCSB A, B, C, D, F, and 5 kops/s per client thread for YCSB E. Since YCSB E has scans instead of reads, we set a lower load for it and measure the tail scan latency instead of the tail read latency. Figure 9b shows that XRP can reduce the 99th-percentile latency by up to 35.9%. Similar to the throughput, the 99th-percentile latency improvement mostly decreases with a larger cache size, and XRP does not have significant effect on YCSB D and E.

7 Related Work

There are four areas of related work: (a) using BPF to accelerate I/O (typically networking), (b) kernel-bypass systems, (c) near-storage compute, and (d) extensible operating systems and library file systems.

BPF for I/O. There is a large number of systems and frameworks that use BPF to accelerate I/O processing, primarily focused on networking and tracing use cases [2,4–6,14,17,24, 27,33,42,45,47]. Most closely related to XRP, XDP [27] accelerates networking I/O by adding a hook in the NIC driver's RX path. It then provides an interface for eBPF programs that either filter, redirect or bounce the packet.

There are no existing systems that use BPF to resubmit storage requests from within the kernel. Kourtis et al. [57] propose a system that uses eBPF functions as an interface to submit disaggregated storage requests in order to avoid crossing the network. In their system, resubmissions occur from a user space service sitting at the host and are not serviced by the kernel itself, since the network is the primary bottleneck (not the kernel software stack). Zhong et al. [80] provide motivation for using BPF for accelerating storage from within the kernel, but do not provide a concrete design, implementation or evaluation.

Kernel bypass. In order to reduce the kernel's overhead when processing I/O, several libraries and operating systems have been designed to let users directly access I/O devices [7,30,31,38,43,52,60,64,66,67,77–79]. Most relevant to our work, Intel's SPDK [77] is a popular kernel-bypass library for storage. In general, the downside of allowing users to access I/O directly is that applications must directly poll for I/O to obtain high performane. This means that cores cannot be shared among processes, which leads to significant underutilization when I/O is not the bottleneck.

Near-storage compute. There are several systems that allow applications to offload their storage functions to the processor embedded within or attached to a storage device [15, 21, 29, 34, 39, 46, 50, 56, 58, 69, 70, 72, 76]. The downside of this approach is that it requires specialized storage devices and/or dedicated hardware.

Extensible operating systems and library file systems. Our approach is reminiscent of extensible operating systems and library file systems from the 90's. Extensible operating systems (e.g., SPIN [32] and VINO [71, 74]) allow extension of kernel functionality via user-defined functions. For example, a client can write kernel extensions that read and decompress video frames from disk. Another related approach is library file systems, such as XN [44, 51]. Similar to XRP, XN allows userspace library file systems to load untrusted metadata translation functions into the kernel, while guaranteeing disk block protection without understanding the file systems' data structures. These approaches required using dedicated operating and file systems, while XRP is compatible with Linux and its standard file systems.

8 Conclusions

BPF has the potential to accelerate applications using fast NVMe devices by moving computation closer to the device. XRP lets applications write functions that can resubmit dependent storage requests to achieve speedups close to kernel-bypass while retaining the advantages of being OS-integrated. Beyond fast lookups, we envision XRP can be used for many types of functions, such as compaction, compression and deduplication. In addition, XRP in the future can be developed as a common interface for other use cases where computation needs to be moved closer to storage, such as programmable storage devices and disaggregated storage systems.

References

[1] 3D Xpoint: A Breakthrough in Non-Volatile Memory Technology. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-micron-3d-xpoint-webcast.html.

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

[3] bpf: Introduce function-by-function verification. https://lore.kernel.org/bpf/20200109063745.3154913-4-ast@kernel.org/.

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

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

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

[7] DPDK Data Plane Development Kit. https://www.dpdk.org/.

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

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

[10] HDFS Architecture Guide. https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.

[11] 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.

[12] LevelDB. https://www.percona.com/software/mysql-database/percona-tokudb.

[13] Linux Socket Filtering Documentation. https://www.kernel.org/doc/Documentation/networking/filter.txt.

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

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

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

[17] 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/.

[18] 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.

[19] Percona TokuDB. https://github.com/google/leveldb.

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

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

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

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

[24] udplb. https://github.com/moolen/udplb.

[25] 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.

[26] WiredTiger storage engine. https://docs.mongodb.com/manual/core/wiredtiger/.

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

[28] Nadav Amit and Michael Wei. The design and implementation of hyperupcalls. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 97–112, 2018.

[29] Antonio Barbalace, Anthony Iliopoulos, Holm Rauchfuss, and Goetz Brasche. It's time to think about an operating system for near data processing architectures. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, page 56–61, New York, NY, USA, 2017. Association for Computing Machinery.

[30] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe user-level access to privileged CPU features. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 335–348, Hollywood, CA, October 2012. USENIX Association.

[31] 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.

[32] Brian N Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc E Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility safety and performance in the SPIN operating system. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 267–283, 1995.

[33] Marco Spaziani Brunella, Giacomo Belocchi, Marco Bonola, Salvatore Pontarelli, Giuseppe Siracusano, Giuseppe Bianchi, Aniello Cammarano, Alessandro Palumbo, Luca Petrucci, and Roberto Bifulco. hXDP: Efficient software packet processing on FPGA NICs. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 973–990. USENIX Association, November 2020.

[34] Wei Cao, Yang Liu, Zhushi Cheng, Ning Zheng, Wei Li, Wenjie Wu, Linqiang Ouyang, Peng Wang, Yijing Wang, Ray Kuan, Zhenjun Liu, Feng Zhu, and Tong Zhang. POLARDB meets computational storage: Efficiently support analytical workloads in cloud-native relational database. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 29–41, Santa Clara, CA, February 2020. USENIX Association.

[35] 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.

[36] Alexander Conway, Abhishek Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard Spillane, Amy Tai, and Rob Johnson. SplinterDB: Closing the bandwidth gap for NVMe key-value stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 49–63, 2020.

[37] 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.

[38] Henri Maxime Demoulin, Joshua Fried, Isaac Pedisich, Marios Kogias, Boon Thau Loo, Linh Thi Xuan Phan, and Irene Zhang. When idling is ideal: Optimizing tail-latency for heavy-tailed datacenter workloads with perséphone. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 621–637, New York, NY, USA, 2021. Association for Computing Machinery.

[39] Jaeyoung Do, Sudipta Sengupta, and Steven Swanson. Programmable solid-state storage in future cloud datacenters. *Communications of the ACM*, 62(6):54–62, 2019.

[40] 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.

[41] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. Reducing DRAM footprint with NVM in Facebook. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–13, 2018.

[42] 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.

[43] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 281–297. USENIX Association, November 2020.

[44] Gregory R Ganger and M Frans Kaashoek. Embedded inodes and explicit grouping: Exploiting disk bandwidth for small files. In *USENIX Annual Technical Conference*, pages 1–17, 1997.

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

[46] Boncheol Gu, Andre S. Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, Jaeheon Jeong, and Duckhyun Chang. Biscuit: A framework for near-data processing of big data workloads. 44(3):153–165, June 2016.

[47] 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.

[48] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner, and Christos Kozyrakis. GhOSt: Fast & flexible user-space delegation of Linux scheduling. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page

588–604, New York, NY, USA, 2021. Association for Computing Machinery.

[49] Jaehyun Hwang, Midhul Vuppalapati, Simon Peter, and Rachit Agarwal. Rearchitecting linux storage stack for µs latency and high throughput. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 113–128. USENIX Association, July 2021.

[50] Yanqin Jin, Hung-Wei Tseng, Yannis Papakonstantinou, and Steven Swanson. KAML: A flexible, high-performance key-value SSD. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 373–384. IEEE, 2017.

[51] 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.

[52] 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.

[53] Kostis Kaffes, Jack Tigar Humphries, David Mazières, and Christos Kozyrakis. Syrup: User-defined scheduling across the stack. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 605–620, New York, NY, USA, 2021. Association for Computing Machinery.

[54] Junaid Khalid, Eric Rozner, Wesley Felter, Cong Xu, Karthick Rajamani, Alexandre Ferreira, and Aditya Akella. Iron: Isolating network-based CPU in container environments. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 313–328, Renton, WA, April 2018. USENIX Association.

[55] Marios Kogias, Rishabh Iyer, and Edouard Bugnion. Bypassing the load balancer without regrets. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 193–207, 2020.

[56] Gunjae Koo, Kiran Kumar Matam, I Te, HV Krishna Giri Narra, Jing Li, Hung-Wei Tseng, Steven Swanson, and Murali Annavaram. Summarizer: trading communication with computing near storage. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 219–231. IEEE, 2017.

[57] 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.

[58] Jaewook Kwak, Sangjin Lee, Kibin Park, Jinwoo Jeong, and Yong Ho Song. Cosmos+ OpenSSD: Rapid prototype for flash storage systems. *ACM Transactions on Storage (TOS)*, 16(3):1–35, 2020.

[59] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. F2FS: A new file system for flash storage. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 273–286, 2015.

[60] 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.

[61] Yoshinori Matsunobu, Siying Dong, and Herman Lee. MyRocks: LSM-tree database storage engine serving Facebook's social graph. *Proceedings of the VLDB Endowment*, 13(12):3217–3230, 2020.

[62] Steven McCanne and Van Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *USENIX Winter 1993 Conference (USENIX Winter 1993 Conference)*, San Diego, CA, January 1993. USENIX Association.

[63] J. Mogul, R. Rashid, and M. Accetta. The packer filter: An efficient mechanism for user-level network code. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, SOSP '87, page 39–51, New York, NY, USA, 1987. Association for Computing Machinery.

[64] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 361–378, Boston, MA, February 2019. USENIX Association.

[65] 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.

[66] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 1–16, Broomfield, CO, October 2014. USENIX Association.

[67] George Prekas, Marios Kogias, and Edouard Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 325–341, 2017.

[68] 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.

[69] 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.

[70] Robert Schmid, Max Plauth, Lukas Wenzel, Felix Eberhardt, and Andreas Polze. Accessible near-storage computing with FPGAs. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–12, 2020.

[71] Margo I. Seltzer, Yasuhiro Endo, Christopher Small, and Keith A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, OSDI '96, page 213–227, New York, NY, USA, 1996. Association for Computing Machinery.

[72] 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.

[73] Dharma Shukla, Shireesh Thota, Karthik Raman, Madhan Gajendran, Ankur Shah, Sergii Ziuzin, Krishnan Sundaram, Miguel Gonzalez Guajardo, Anna Wawrzyniak, Samer Boshra, et al. Schema-agnostic indexing with Azure DocumentDB. *Proceedings of the VLDB Endowment*, 8(12):1668–1679, 2015.

[74] Christopher A Small and Margo I Seltzer. Vino: An integrated platform for operating system and database research. 1994.

[75] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, et al. CockroachDB: The resilient geo-distributed SQL database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1493–1509, 2020.

[76] Mark Wilkening, Udit Gupta, Samuel Hsia, Caroline Trippel, Carole-Jean Wu, David Brooks, and Gu-Yeon Wei. RecSSD: near data processing for solid state drive based recommendation inference. *arXiv preprint arXiv:2102.00075*, 2021.

[77] 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.

[78] 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.

[79] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S. Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, Pedro Henrique Penna, Max Demoulin, Piali Choudhury, and Anirudh Badam. The demikernel datapath os architecture for microsecond-scale datacenter systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 195–211, New York, NY, USA, 2021. Association for Computing Machinery.

[80] Yuhong Zhong, Hongyi Wang, Yu Jian Wu, Asaf Cidon, Ryan Stutsman, Amy Tai, and Junfeng Yang. BPF for storage: An exokernel-inspired approach. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, page 128–135, New York, NY, USA, 2021. Association for Computing Machinery.