BypassD: Enabling fast userspace access to shared SSDs

Sujay Yadalam
University of Wisconsin-Madison
Madison, Wisconsin, USA
sujayyadalam@cs.wisc.edu

Chloe Alverti
National Technical University of Athens
Athens, Greece
xalverti@cslab.ece.ntua.gr

Vasileios Karakostas
University of Athens
Athens, Greece
vkarakos@di.uoa.gr

Jayneel Gandhi
Meta
Menlo Park, California, USA
jayneel@meta.com

Michael Swift
University of Wisconsin-Madison
Madison, Wisconsin, USA
swift@cs.wisc.edu

Abstract

Modern storage devices, such as Optane NVMe SSDs, offer ultra-low latency of a few microseconds and high bandwidth of multiple gigabytes per second. At these speeds, the kernel software I/O stack is a substantial source of overhead. Userspace approaches avoid kernel software overheads but face challenges in supporting shared storage without major changes to file systems, the OS or the hardware.

We propose a new I/O architecture, *BypassD*, for fast, userspace access to shared storage devices. BypassD takes inspiration from virtual memory: it uses virtual addresses to access a device and relies on hardware for translation and protection. Like memory-mapping a file, the OS kernel constructs a mapping for file contents in the page table. Userspace I/O requests then use virtual addresses from these mappings to specify which file and file offset to access. BypassD extends the IOMMU hardware to translate file offsets into device Logical Block Addresses. Existing applications require no modifications to use BypassD. Our evaluation shows that BypassD reduces latency for 4KB accesses by 42% compared to standard Linux kernel and performs close to userspace techniques like SPDK that do not support device sharing. By eliminating software overheads, BypassD improves performance of real workloads, such as the WiredTiger storage engine, by ~20%.

CCS Concepts: • **Hardware → External storage**; • **Software and its engineering → File systems management**; • **Information systems → Storage virtualization**.

Keywords: I/O performance, low latency, direct access, sharing, userspace, SSD, storage systems

ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA

ACM ISBN 979-8-4007-0372-0/24/04.
https://doi.org/10.1145/3617232.3624854

ACM Reference Format:
Sujay Yadalam, Chloe Alverti, Vasileios Karakostas, Jayneel Gandhi, and Michael Swift. 2024. BypassD: Enabling fast userspace access to shared SSDs. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3617232.3624854

1 Introduction

Operating system storage stacks were developed in the era of spinning disk drives with latencies in the range of milliseconds, but modern SSDs, such as Intel Optane, provide access latencies as low as 4μs and bandwidth close to 7 GB/s [28]. Software overheads that were acceptable for slow devices are instead becoming the *dominant* cost in file access. For example, an Optane SSD can return a 4KB block in 4μs, while reading the block through standard Linux kernel takes almost 8μs. Prior efforts to reduce these software overheads fall into two main categories: i) optimizations to kernel storage stacks, and ii) userspace file/storage access.

In the OS kernel, researchers have optimized I/O scheduling [10, 26, 66], overlapped asynchronous operations [37], and used polling instead of interrupts [10, 64]. While these approaches greatly reduce overheads, they still require context switches to enter and leave the kernel, and security mitigations make these switches more costly [25, 57]. Dedicating a kernel thread to poll for requests from user mode [54] removes these switches but at the cost of an extra core.

SPDK and others [34, 49, 65] reduce latency by directly accessing an SSD from userspace. This reduces access latency but has limitations. First, it burdens application developers with replacing the kernel block layer and file system, and managing atomicity and crash consistency. Second, sharing a device securely between applications/users is challenging: devices are unaware of file layout or permissions, and consequently, userspace code gets access to all blocks on the device. Hence, a malicious process can read or corrupt the entire disk.

The need for device sharing is increasing. Within data centers, operators run multiple applications on the same system to increase utilization [14, 23, 50], meaning there are

often many workloads simultaneously seeking low-latency access to storage. Virtual machines (VMs) and containers share not only CPUs and memory but also storage devices such as NVMe SSDs. Some workloads may share access to same files [15, 20, 45, 52].

Thus, systems must provide *low-latency access* while allowing applications to *share a device*. Unfortunately, most existing userspace storage solutions [10, 34, 65] do not satisfy both requirements. Those systems that do [11, 33, 51], come with the cost of significant changes to devices, access protocol, and file systems. They rely on moving the entire file system [33] or parts [11, 51] to the device. This burdens devices that have limited compute power and memory, limiting their performance.

In this paper, we propose *BypassD*, a new I/O architecture that provides protected access to files directly from userspace. BypassD offloads permission checks to the hardware, ensuring a process can only access data with appropriate permissions. Our design builds on the key insight that the IOMMU hardware, currently used for translating virtual memory addresses to physical addresses, can be repurposed to translate file offsets to logical block offsets in the SSD. Much like NVM file systems rely on memory mapping for low-latency access to data in NVM [6, 32], BypassD constructs page tables in application address spaces that map virtual addresses to file data locations, and the SSD uses the IOMMU to check access and retrieve these mappings without kernel involvement.

File access in BypassD follows two paths. Metadata operations such as open and append are processed by in-kernel file systems. File reads and writes are sent directly to the device from a userspace library. During file open, the kernel maps the file contents into the application address space. Userspace read/write requests contain virtual addresses corresponding to file data. BypassD makes only minimal changes to existing file system code to map block addresses into address spaces; the IOMMU to support a new translation type; and SSDs to request translations of block addresses from the IOMMU. The entire mechanism is transparent to applications.

We implement BypassD in the Linux kernel with the ext4 file system. We evaluate the performance of BypassD using a set of microbenchmarks and storage-intensive applications, and observe that BypassD exposes the true latency of a device to applications with <0.8μs overhead. BypassD performs almost as fast as SPDK which lacks support for sharing, and 25% better on average than a standard Linux kernel. Compared to a recent work that embeds application code into the kernel for lower latency [70], BypassD improves the throughput of storage-intensive workloads by 9.6%.

In summary, the main contributions of this paper are:

- An end-to-end storage architecture that enables fast userspace access to shared devices called BypassD. BypassD performs metadata operations in kernel and data operations in userspace allowing for a clean design that avoids any application modifications.
- Modest modifications to the IOMMU hardware to ensure fast access and permission checks without kernel mediation.
- An implementation of BypassD which includes a user library, extensions to the Linux ext4 file system, and a model of the hardware modifications.
- Evaluation of BypassD across a variety of micro-benchmarks, real-world applications, and comparison to state-of-the-art approaches.

	Time (ns)	% of total time
Kernel user mode switch	160	2%
VFS + ext4	2,810	36%
Block I/O layer	540	7%
NVMe driver	220	3%
Device time	4,020	51%
User kernel mode switch	100	1%
Total	7,850	

Table 1. Latency breakdown of 4KB read() on Optane SSD.

2 Background and Motivation

New media technologies such as 3D XPoint [28] and low-latency NAND flash [13] have led to devices that offer sub-ten microseconds access latency and up to 1.6 million IOPS[4, 13, 27]. They leverage high-speed I/O mechanisms to remove hardware overheads in data access.

NVMe. The NVMe protocol was designed for low-latency storage requests [3]. It uses in-memory queue pairs for communication between host CPUs and devices. A queue pair comprises a Submission Queue (SQ) on which requests are sent to the device and a Completion Queue (CQ) from which completions are processed. NVMe supports up to 64K queue pairs per device, each with up to 64K queued requests. Queue pairs can be mapped into user-mode memory, allowing direct submission and completion of requests from application code without kernel involvement.

IOMMU and DMA protection. Storage devices perform DMA to read/write data from/to main memory. Using physical address for DMA make systems vulnerable to rogue devices and buggy drivers which could wrongly perform DMAs to arbitrary memory locations. To avoid this, processors use an IO Memory Management Unit (IOMMU) that supports DMAs with IO virtual addresses (IOVA) instead of physical addresses. Devices issue DMA using IOVA which the IOMMU translates to a physical address before accessing memory. IOMMUs use special page tables set up by the OS to translate IOVAs. With *Shared Virtual Addressing* (SVA) [41], a feature of modern processors, IOMMUs can access process page tables along with the IOMMU-specific page tables.

Software is the new I/O bottleneck. Modern SSDs, such as Intel Optane SSD [27], Samsung Z-SSD [13], and Toshiba's XL-Flash [4], provide access latencies under 10μs. With such fast devices, the time spent in the kernel I/O software stack is no longer negligible compared to the device access time.

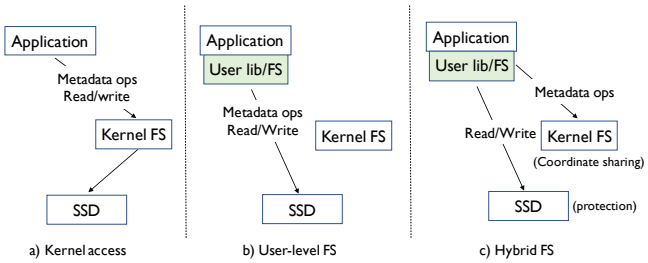

Figure 1. Different approaches for accessing an SSD.

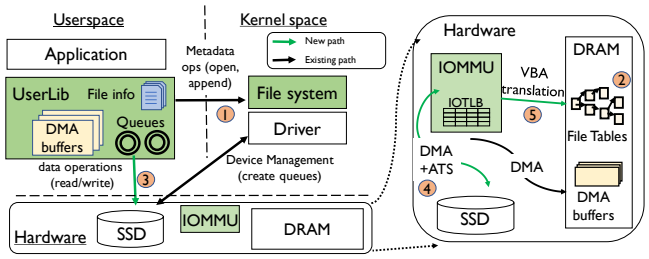

Figure 2. BypassD components (shown in green).

Figure 1a gives a high-level view of a traditional kernel file system used to manage and access the data on the device. Typically, accessing the device through the kernel involves multiple storage layers as shown in Table 1. The table shows the time spent in each of these layers during 4KB reads to Optane P5800X SSD [27]: device processing is ~50% of access time and the remainder is software overhead. As devices get faster, the relative overhead will only worsen. It is therefore important to develop new techniques to reduce these software overheads and expose devices' full performance to applications [9].

Userspace access is challenging. Direct userspace access to storage as shown in Figure 1b is promising as it offers reduced overheads. However, the main challenge to userspace device access is permission checks for sharing. When accessing a file through the kernel, the kernel relies on process credentials, file permission metadata and/or access-control lists (ACLs) to check whether a process has permission to access a file. With userspace solutions such as SPDK [65] that bypass the kernel, there is no trusted entity that can enforce such permission checks. Applications can therefore access and modify from userspace all blocks on the device. Therefore, a device cannot be shared securely between multiple applications with approaches such as SPDK.

Protection in hardware. One way to overcome the challenge of userspace access is to offload the permission checks to the hardware as shown in Figure 1c. Prior designs empower the device to perform permission checks by copying permission data to the device [11, 33, 51] or by splitting the device into partitions with unsafe user-level sharing [49].

Drawbacks of protection in device. Enforcing permission checks on the device has several challenges. Prior designs [11, 33, 51] make significant changes to the device and access protocol to store permission information, yet suffer from several drawbacks (1) high reliance on the device's computational power to perform both I/O processing and permission book-keeping, (2) burden on the device's memory to store permission data especially when the device is busy, and (3) high costs of updating permission information which could lead to starvation in some cases [11].

The above drawbacks may also lead to unpredictable performance including sudden drops in throughput and high tail latencies. For instance, in MonetaD, while the device is updating permission data, it has to stop serving requests or temporarily suspend permission checking [11]. If the device exhausts memory and is unable to cache all permission data, MonetaD relies on an expensive miss handling procedure that involves both the userspace library and the kernel to access the missing permission data. This can increase the I/O latency by 8x [11].

3 BypassD Overview

BypassD is an I/O architecture that provides low-latency access to shared devices. BypassD's main goals are to:

1. Eliminate the software stack overheads and provide low latency access to devices.
2. Securely share devices between multiple applications.
3. Support unmodified POSIX-compatible applications.

These sharing and POSIX-compatibility goals ensure BypassD is widely usable. In achieving these goals, we follow four guiding principles:

1. *Kernel bypass*: For low latency, bypass the kernel and provide direct access to the device from userspace.
2. *Leverage existing file systems and drivers*: Make use of existing mature and well-tested kernel file systems and device drivers with only minimal changes.
3. *Offload permission checks to hardware*: Enforce file system access rules by offloading checks to hardware.
4. *Co-exist with kernel*: Ensure file access is available through both BypassD and the kernel to handle corner-cases and avoid performance regressions.

3.1 Design overview

Above principles push towards a design that leverages existing OS and hardware mechanisms by only offloading latency-critical file data accesses to userspace. BypassD adopts a split architecture, wherein latency-critical data operations such as read() and write() are handled directly in userspace, and file operations that modify the file metadata such as open() and appends are processed by the kernel. We call the direct path from userspace to device the *BypassD interface*, and the default path through the kernel, the *kernel interface*.

While accessing data through the *BypassD interface*, the permission checks are performed in the hardware. The kernel file system is responsible for policy decisions that the hardware enforces during device accesses. Unlike prior designs [11, 33, 51], BypassD performs permission checks on

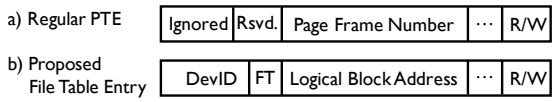

Figure 3. Format of File Table Entry.

the host (CPU) hardware, thereby avoiding the drawbacks of enforcing protection on the device (discussed in Section 2).

BypassD seeks inspiration from virtual memory and address translation. With virtual memory, processes access memory using virtual addresses that are translated to physical addresses by the Memory Management Unit (MMU). Along with the translation, MMU ensures that processes have the correct permissions to access memory. Similarly, in BypassD, processes access data on a device using virtual addresses, termed *Virtual Block Addresses* (VBA) that are mapped to *Logical Block Addresses* (LBA) using page tables. Enhanced IOMMU hardware translates Virtual Block Addresses in requests to Logical Block Addresses and ensures the process has permissions to access the blocks.

3.2 BypassD components

Figure 2 shows the key components in BypassD. We briefly describe these components and then discuss how they interact to achieve the goals of BypassD.

Kernel file system and device driver. BypassD reuses the kernel file system and device driver to manage the device. ① All file operations that modify the file system metadata are handled by the kernel. Unlike prior designs [11, 33, 51], the file system metadata is present only inside the kernel and not shared with either userspace libraries or the device. This avoids complexities involved in maintaining metadata consistently across multiple layers.

In addition to the regular responsibilities, the kernel file system in BypassD is responsible for virtualizing block addresses. In this regard, BypassD supports a new *fmap()* system call that creates VBA to LBA mappings for the process. fmap() is akin to mmap(): kernel maps the file to the process address space and returns a virtual address (Virtual Block Address). ② Additionally, the kernel creates *file tables* (see Section 3.4): file table entries that map a single file. File Table Entries (FTEs) are special page table entries that hold LBAs in place of Page Frame Numbers. The kernel attaches these file table entries to process' address space during fmap().

Userspace library. BypassD's UserLib is a shim library that intercepts system calls. UserLib handles data operations in userspace and forwards metadata operations to the kernel. It is transparent to applications so requires no effort to use BypassD. The library adds minimal software overhead during data operations thereby achieving low latency access.

UserLib maintains queues and DMA buffers required for interacting with the device. ③ When the library intercepts data operations such as read()/write(), it uses the queues to submit requests to the device and polls them for completion.

UserLib also tracks information about open files such as flags, offset, size and the starting VBA. It also tracks a list of ongoing partial-writes (more in Section 4.5)

Hardware. The kernel offloads permission checks to the hardware. BypassD builds on existing IOMMU translation and protection mechanisms for IO Virtual Addresses (IOVA). BypassD proposes minor enhancements to the IOMMU hardware: as shown in Figure 2, in addition to translating IO Virtual Address (IOVA) of DMA buffers, the IOMMU in BypassD translates VBAs to LBAs and performs read/write permission checks. To do so, the IOMMU walks the page tables of the requesting process to access the attached file table entries (FTEs).

④-⑤ SSDs communicate with the IOMMU to translate VBA in incoming requests and perform permission checks. After receiving translated LBAs, SSDs complete the I/O requests just like they do in systems today.

BypassD is the first system to enforce permission checks on the host hardware. BypassD avoids significant hardware changes and tries to reuse existing translation and protection mechanisms. As shown in Section 6, VBA translation can take as less as 550ns. With such minimal overheads and minor hardware enhancements, BypassD achieves low latency direct access from userspace and device sharing.

3.3 BypassD interface

BypassD provides a new interface enabling processes to access devices directly from userspace. Setting up the interface consists of three steps: i) setting up queue pairs to the device, ii) allocating DMA buffers, and iii) setting up file tables.

UserLib requests the kernel driver to create new queues and map it into the userspace. Upon receiving such a request, the kernel driver creates new queues and registers them with the device. Crucially, while registering with the device, we propose that the driver links the *Process Address Space ID* (PASID) with each queue. This PASID (like ASIDs in TLBs) is later used by the IOMMU to identify the proper page table to walk when translating VBAs to LBAs during data requests.

UserLib also allocates pinned DMA pages for passing data to and from the device. This is similar to SPDK [65] which likewise allocates huge pages during initialization for DMA.

When UserLib intercepts a file open(), it not only forwards it to the kernel but also issues fmap() requesting the kernel to setup the VBA to LBA mappings and attach the file table entries to process page tables. After a file is successfully fmap()-ed, the *BypassD interface* is initialized and UserLib can then issue I/O requests directly from userspace.

3.4 File tables

Special page table entries in the process page table tree map VBAs to the target file's Logical Block Addresses (LBAs). We name these *file table entries*. The kernel attaches file table entries to a process' page table during *fmap()*. Figure 3 shows their structure. A file table entry stores LBA (instead of PFN),

the device ID (DevID), and a special bit (File Table–FT) to differentiate it from a regular page table entry. DevID is used to identify the device containing the file. This ensures that a malicious process does not use the VBA to access files on another device. Prior work [36] has used similar entries for hardware paging.

3.5 Translation and protection in the hardware

When the device receives a request with VBA, directly from userspace, it issues a translation request to the IOMMU. Along with the VBA, the translation request includes operation (read/write), I/O size and PASID linked to the queue on which the request arrived. With SVA [41], the IOMMU uses PASID to walk the page tables of the process and translates VBA to LBA using the attached file table entries. During the translation process, IOMMU ensures that the permissions are met. Upon successful translation, IOMMU returns the LBAs to the device which proceeds to serve the request.

3.6 Revoking BypassD interface

At any point, the kernel can revoke a process' ability to access a file through the *BypassD interface*. The kernel does this under scenarios discussed in Section 4.5 using the below mechanism:

1. Kernel detaches FTEs corresponding to the file(s) that it wants to revoke access to.
2. Requests issued directly from userspace to such a file fail because the IOMMU cannot translate the VBA without the file table entries.
3. When an I/O fails, UserLib issues an fmap() again requesting the kernel to re-attach the file tables.
4. Kernel rejects the request returning a VBA of 0. An empty VBA indicates to UserLib that the file can no longer be accessed directly from userspace.
5. Thereafter the process falls back to the *kernel interface* for all subsequent I/O.

As we shall see, this is a powerful mechanism that removes the need to handle complex but rare sharing cases in userspace; instead BypassD falls-back to the kernel.

Race between access revocation and I/O access. When the kernel revokes direct access for a process, there could be on-going I/O for this file. A problem arises when the kernel deallocates blocks of a file and reallocates them to another. There is a race when the SSD has already translated addresses but not yet performed flash accesses, as invalidations will not cancel the requests. This race is very small, and is handled by delaying re-allocation of blocks until a sync point (e.g., fsync()) that forces completion of all pending I/Os.

4 BypassD Implementation

We implemented BypassD with ext4 without data journaling. BypassD currently supports NVMe storage devices. The table below shows the size of implementation.

Component	Lines of code
Kernel changes	517 lines
ext4 changes	1303 lines
Device driver changes	885 lines
UserLib	1496 lines

Table 2. Total lines of code added/modified in BypassD.

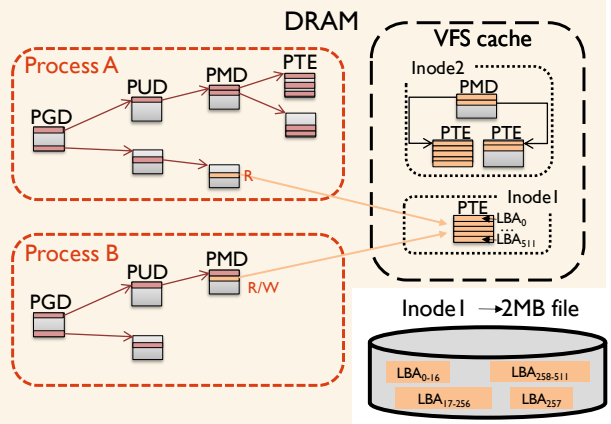

Figure 4. Attaching file table fragments to processes address spaces via the fmap() call.

BypassD interposes on file system access using `LD_PRELOAD`. Programmers can select whether a file is accessed using BypassD interface or the default kernel interface. Files requiring low latency can be accessed using the BypassD interface.

Table 3 summarizes how BypassD handles common file operations. Below, we discuss important actions performed in BypassD and then discuss sharing/concurrency scenarios.

4.1 Creating file table entries

During *fmap()*, the kernel allocates a region in the calling process' virtual address space that equals or exceeds the file's size and attaches the file table entries. The kernel returns the starting virtual block address of this virtual region from fmap(). If the kernel decides that a file is not eligible for being accessed directly from the userspace, it returns a VBA of 0.

The file system creates the file table entries by retrieving the Logical Block Addresses (LBAs) of the file blocks. We call this a *cold fmap()*. If the block mappings are cached in memory (e.g., ext4's extent status tree), obtaining LBAs is inexpensive. If not, the file system issues I/Os to read the file to block mapping from the device. We discuss the overheads of creating these file tables in Section 6.

Pre-populating file table entries. Prior research has studied the overheads of setting up page tables [6, 24, 36, 47]. To avoid the overheads of creating private file table entries for every process that opens the same file, BypassD uses shared pre-populated file table entries. File table entries can be pre-populated because the entries' content (LBAs) are known upfront and change slowly over time. A similar strategy is already implemented for persistent memory file mappings [6]. Note that BypassD does not support CoW (copy on write)

File operation	Userlib actions	Kernel FS actions
open()	Forward to kernel and issue fmap() indicating intent to access file through BypassD interface.	Attach file table entries to process page table.
read()/write() (incl. variants)	Create NVMe read/write request: VBA in request formed by adding offset to Starting VBA of file. Issue request on SQ and poll CQ for completion.	
appends (write to end of file)	Forward to kernel.	Allocate block, update file metadata. Create new file table entries and attach to page table. Issue write to new block directly to device without buffering.
fallocate()/ftruncate()	Forward to kernel.	Allocate/de-allocate blocks. Zero out newly allocated blocks. Attach/detach file table entries corresponding to the blocks.
fsync()/fdatasync()	Flush all queues (NVMe flush) to make data durable, then forward to kernel.	Update file modified timestamps. Flush file metadata to disk.
close()	Forward to kernel.	Detach file table entries.

Table 3. Actions performed by UserLib and kernel FS (in addition to normal actions) in BypassD during common file operations.

filesystems such as btrfs since the offset to block mapping changes on every write.

As files commonly grow or shrink starting at their final block and in a dense manner, we use entries of page table radix-trees to store the LBAs, building them in a bottom-up manner. Figure 4 shows how a single FTE leaf can store all the LBAs of a 2MB file. BypassD maintains file table entries in the file's cached VFS inode. Their lifetime equals the time that the inode remains cached. The file system also tracks all processes accessing a file in the file's cached inode.

BypassD leverages the cached file table entries to accelerate fmap() by attaching them in almost constant time to the calling process address space (*warm fmap()*). The kernel builds the page tables of the calling process up to an intermediate level and attaches the cached file table entries using simple pointer update operations during the fmap() call. BypassD maps the entire file (no partial maps). Since the attachment can happen only at the intermediate granularities of the page table radix-tree (e.g., PUD or PMD), BypassD allocates a virtual region for the file at the same granularity (multiples of 1GB or 2MB). If the allocated virtual region is greater than the file size, the remaining virtual space can be used to extend the mapping in place if the file grows in size. Since BypassD maps files completely, no page faults occur during file access.

BypassD file table entries are shared between processes opening the same file. The shared file table entries have the maximum access rights preset (the R/W bit is always set) by default. To support different read/write access permissions per open, BypassD sets the permission bit in the private part of the process's page table trees, where the entries are attached, similar to [6, 24]. Figure 4 shows how two processes open the same file with different permissions.

Extending or truncating files. When new blocks are allocated to a file, the kernel attaches new FTEs corresponding to the new blocks so that the process can access these blocks directly from userspace. Similarly, when blocks are deallocated, corresponding FTEs are detached to deny access to those blocks from the userspace.

The file system ensures that newly allocated blocks are zeroed out, and applications must be aware of zero padding at the end of a file if writes do not complete. Security dictates the blocks must be zeroed, as otherwise the application could read the previous data in those blocks.

4.2 Issuing requests through BypassD interface

UserLib intercepts all variations of read (read()/pread()) and write (write()/pwrite()) system calls. It issues all non-metadata-modifying operations (all reads and selected writes) to the device directly from userspace. UserLib converts the offset into an equivalent VBA and copies data between the user buffer and the DMA buffer.

Handling reads. Since reads do not modify metadata, they are issued directly from userspace.

Handling writes. UserLib handles overwrites and appends differently. Overwrites to existing blocks of a file do not modify any file metadata, so UserLib issues them directly to the device from userspace. Appends to the end of a file have to be handled carefully as they require allocation of new blocks and modify file metadata. UserLib detects such appends using stored file size and routes them to the kernel. The kernel issues appends directly to the device without buffering them in the page cache.

UserLib does not provide zero-copy I/O. Instead, UserLib uses pinned DMA buffers over user buffers for two reasons: i) to avoid the cost of pinning and unpinning user pages, and ii) ensuring correctness. When user buffers are used for I/O, programs must ensure that other program threads do not free those buffers during an I/O. UserLib could adopt Demikernel's [67] strategy of allocating memory to an application from a DMA-capable heap to support zero-copy I/O.

4.3 Hardware enhancements

In BypassD, the IOMMU hardware is entrusted to perform VBA translations and permission checks during device access. The proposed modifications are modest.

Device changes. Devices receive read/write requests that contain VBA instead of LBA. They then send VBA translation request to the IOMMU through PCIe Address Translation

Service (ATS) [48]. For reads, VBA translation and DMA are serialized because the device needs block addresses before reading from the blocks. For writes, the translation request and DMA are sent simultaneously because the data is first copied into device memory and then written to the block. Writes therefore do not experience any VBA translation overheads.

IOMMU changes. The IOMMU in commercially available processors translates virtual addresses from I/O requests to physical addresses in memory. With BypassD, we propose that the IOMMU is enhanced to also handle translation requests for VBAs. The IOMMU translates VBA just like it translates virtual memory addresses. It walks the page tables as usual but interprets the contents of leaf entries differently. BypassD adds a new bit to the PTE, *FT*, to distinguish regular page table entries from file table entries, as shown in Figure 3. If the bit is set, the IOMMU interprets the contents of the entry as a LBA instead of a PFN and will only use it for block address translations. The IOMMU also compares DevID in the FTE to requester ID in the translation request to ensure that the process is accessing the files on the correct device.

The IOMMU could receive requests for regions larger than a single page, both for memory buffers and with BypassD for blocks. Current IOMMUs walk the page table once for each page to fetch multiple entries. Similarly, in BypassD the IOMMU responds to large VBA translation requests with multiple pairs of (LBA, length) and coalesces them if possible.

Often there is no temporal locality in block accesses, so the IOMMU does not cache FTEs in the IOTLB, thereby preventing IOTLB pollution. However, the higher level entries of the page table are cached in IOMMU's translation caches. BypassD would therefore benefit from larger translation caches but not necessarily a larger IOTLB.

4.4 BypassD guarantees

Our implementation of BypassD is POSIX-compliant and provides the consistency guarantees of traditional kernel file systems such as ext4. Similar to ext4 without data journaling, BypassD provides metadata crash consistency but not data consistency. Currently, BypassD only supports synchronous operations, i.e., all operations are considered complete when the corresponding call returns to the user application.

Accessed and modified timestamps. BypassD deviates slightly from POSIX semantics when it comes to updating the accessed and modified timestamps. When a file has been accessed or written to, the file metadata in the kernel will not reflect it immediately. The timestamps are updated when the file is closed or during fsync()/fdatasync(). This strategy is similar to memory-mapped files and allowed by POSIX [1, 2].

4.5 Sharing and concurrency

One of the primary goals of BypassD is to enable device sharing between multiple applications. In this section, we explain how BypassD handles intra- and inter-process sharing.

4.5.1 Intra-process sharing.

Since BypassD provides a synchronous interface, all data operations complete only after the device accepts them. All metadata operations complete after the kernel returns. Therefore, the device acts as the point of coherence for data operations and the kernel for metadata operations. As all threads of a process share UserLib, it provides a consistent view of file information, such as starting VBA, size, and file offset.

BypassD supports sharing a device and files between threads of a process. However, special care has to be taken while handling partial writes that do not completely overwrite the smallest block size provided by the device.

Serializing partial writes. UserLib handles such writes by reading the old block data, modifying and writing it back. Without synchronization, two threads may clobber each other's writes. UserLib serializes partial writes to the same file to avoid data inconsistencies. It maintains a list of offsets of ongoing partial writes for each open file. When another thread issues a write, UserLib looks up for matching offsets in the list. If a match exists, it delays the later write.

4.5.2 Inter-process sharing.

BypassD interface supports multiple processes performing reads and overwrites to the same file but cannot support operations on shared files that modify metadata. The kernel revokes direct access (see Section 3.6) when it identifies multiple processes changing a file's metadata.

Concurrent access through BypassD and kernel interface. It could happen that a file is opened through BypassD interface in one process and through kernel interface in another. Achieving coherency and consistency is challenging. BypassD avoids this scenario by not supporting concurrent access through both BypassD and kernel interfaces. If a file is already open through the kernel interface, fmap() returns zero. Likewise, if a file is mapped for userspace access and then opened for kernel access, BypassD revokes a process' direct access to the file using the mechanism described in Section 3.6.

5 Discussion

5.1 Enhancements to BypassD

Appends from the userspace. Appends are routed through the kernel since they modify file metadata. To accelerate frequent appends, BypassD could pre-allocate blocks using fallocate() and then issue overwrites to the pre-allotted blocks. We implement this as an *optimized append* operation. An alternate but more intrusive approach is to adopt the *relink* operation of SplitFS [32] that can atomically swap newly appended blocks from a staging file into the target file.

Non-blocking writes. Currently, BypassD only supports synchronous writes that return after data is written to the device. To improve write latency and throughput, BypassD could be modified to support non-blocking writes, i.e., writes do not wait for data to be written out to the device. However,

this change incurs synchronization and consistency overheads to ensure that reads always see the latest data which could be buffered in an unprocessed write request. With this approach, the per-inode range-based locking of CrossFS [51] can be used to increase concurrency.

Alternate data structures. BypassD stores VBA translations in page tables, which adds cost to fmap() for large files. Using a different data structure with a new hardware walker (e.g., rIOMMU [42]) could reduce this cost.

5.2 BypassD in virtualized environments

Containers. BypassD supports sharing an SSD securely between multiple containers without requiring additional modifications. Container environments use mount namespaces to provide processes an isolated view of the device. The kernel ensures that a containerized application can only open and access files within its namespace. Since BypassD relies on the kernel for access control and metadata operations, BypassD works readily with containers.

Virtual Machines (VMs). BypassD can also enable direct userspace access from processes within a VM. BypassD requires similar hardware virtualization support for direct access to devices required by a guest OS, namely SR-IOV [30] or Scalable-IOV [29]. When a process inside a VM accesses the device directly, the IOMMU performs a nested address translation to translate VBAs to LBAs. As isolation between VMs is provided at the block level by Scalable-IOV or SR-IOV, this design does not support file sharing across virtual machines.

5.3 Qualitative security evaluation

BypassD assumes a threat model in which the user process, including UserLib, could be malicious while the kernel and the hardware (device and IOMMU) are part of the trusted computing base (TCB). In BypassD, a malicious process can only read and write files it has permissions for; it cannot access any other files owned by other users, exactly the same as normal file access.

BypassD ensures that a process can only access blocks of files for which it has right permissions. Much like virtual memory, processes are required to accesses blocks using VBAs instead of LBAs. If a process tries to access a block using a LBA or an invalid VBA, the VBA translation fails in the IOMMU and the SSD returns an error code to the process without completing the I/O. A process can therefore only access blocks with valid VBAs.

VBAs are valid if corresponding file table entries (FTEs) exist in the process page tables. A valid VBA only exists when the file has been successfully opened through the kernel; a valid VBA implies that the kernel has approved the process' access to the file. When a file is closed or when blocks of a file are deallocated, the kernel detaches the FTEs from the process page tables, invalidating the VBAs. A process cannot access the blocks from userspace thereafter. Therefore, a process can only access files after they are successfully opened through the kernel and before they are closed.

BypassD guarantees confidentiality of user's data even when blocks are reallocated to another user's file. BypassD zeros a block during allocation, thereby ensuring that direct accesses to the file does not return the previously stored data. As mentioned in Section 3.6, BypassD avoids race between direct I/O and block reallocation by delaying the reuse of a block until a sync point.

6 Evaluation

We answer the following questions about BypassD:

1. How does BypassD perform while accessing a low latency SSD? How does it compare to the standard Linux kernel, SPDK and state-of-the-art solutions?
2. What are the overheads in BypassD?
3. Can a device be shared efficiently between multiple applications?
4. How does performance of real world applications improve with BypassD?

6.1 Methodology

Experimental setup. We run all our experiments on an Intel Xeon Gold 5317 CPU with 12 cores (24 with Hyper-Threading enabled) and 96GB memory, connected to an Intel Optane P5800X SSD. We use Ubuntu 20.04 with Linux 5.4. We disable turbo boost and frequency scaling (C states).

Workloads. We run 3 sets of workloads to evaluate BypassD's performance. We use micro-benchmarks to study the latency and throughput, real world workload to understand BypassD benefits in production systems, and state-of-the-art research workloads to understand the maximum potential of BypassD.

6.2 VBA translation modeling

BypassD proposes minor enhancements to the IOMMU enabling it to translate Virtual Block Addresses (VBAs) to Logical Block Addresses (LBAs). Since commercially available processors do not have this feature, we emulate the overheads of VBA translation by adding a delay while issuing requests from UserLib. VBA translation overhead includes two costs: PCIe bus latency and IOMMU translation latency.

PCIe round-trip latency. To capture the round-trip latency across PCIe, we composed an experiment that repeatedly reads 64 bytes from the Optane SSD device register. On average, the PCIe round trip latency was about **345ns** (similar to [21, 46]). We assume that the measured PCIe latency is symmetric. Address Translation Service (ATS) requests can be reordered with respect to other requests. Therefore under contention, translation requests can be prioritized to limit the total VBA translation overheads.

Experiment configuration	Latency
IOMMU off	1120 ns
IOMMU on; constant src and dest (IOTLB hit)	1134 ns
IOMMU on; varying src, const dest (IOTLB miss)	1317 ns

Table 4. IOMMU translation overheads experiment: IOAT DMA copy latency.

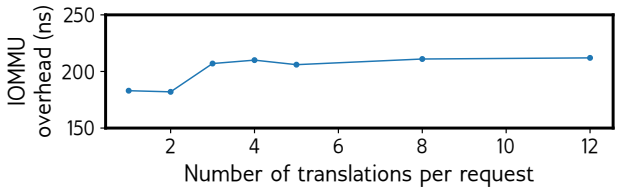

Figure 5. IOMMU overhead versus number of translations per ATS request.

PCIe latencies not only depend on the generation but also depend on the host CPU and the device. We believe that future systems will have lower latencies to support low-latency CXL memories [39, 43]. This would benefit BypassD as the VBA translation overhead would be lower.

IOMMU translation latency. We measured the IOMMU translation costs in modern processors using Intel's I/O Acceleration Technology (IOAT) DMA engine, similar to [7]. The IOAT engine uses the IOMMU to translate buffer virtual addresses to physical addresses. We ran experiments with different configurations as shown in Table 4 to measure translations overheads. We forced IOTLB hits by using a fixed buffer for the DMAs and forced IOTLB misses by changing the buffer virtual addresses. First, we noticed that the translation overhead is negligible if the IOMMU is enabled and the translation hits in the IOTLB (1134ns vs 1120ns). Second, an IOTLB miss resulting in a page walk adds about **183ns** (1317ns instead of 1134ns). Third, we noticed that the IOMMU translation overhead does not increase significantly with number of translations per request. Figure 5 shows the IOMMU translation overhead for different number of translations with contiguous virtual addresses. We see a slight increase going from 2 to 3 translations per request. Thereafter, the translation overheads does not increase with the number of translations. This is mostly because a single cacheline (64B) fits 8 page table entries. Therefore, a single cacheline read could translate an address range of 32KB.

In our experiments, we emulate PCIe latency and IOMMU translation overheads by adding a delay in UserLib (nop() loop) . The delay added depends on the translation size with a minimum delay of 550ns.

6.3 Microbenchmarks

We use fio tester benchmark [8] to measure the latency and throughput benefits of BypassD. For all experiments, we issue direct I/O (`O_DIRECT`) and set queue depth to 1. We compare BypassD against the following approaches:

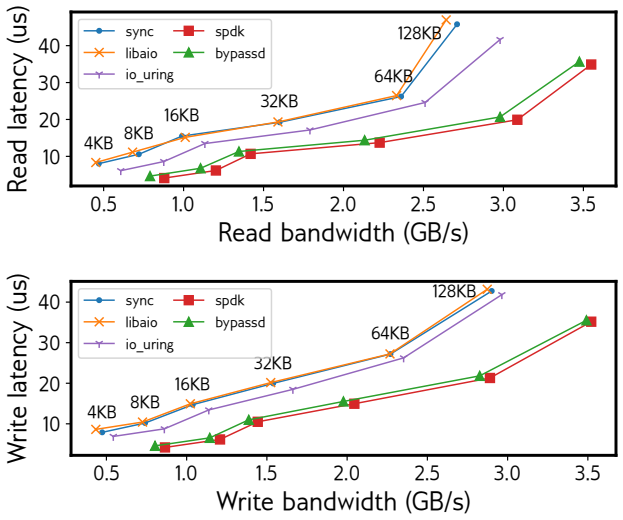

Figure 6. FIO single threaded random-access latency.

sync: Baseline Linux with synchronous system calls.
libaio: Linux's native asynchronous I/O interface [19].
io_uring: Newer async I/O interface that avoids context switches by using ring buffers to communicate between user and kernel [54]. We use fixed buffers and enable submission queue polling by the kernel for maximum performance.
spdk [65]: Userspace driver to access storage directly from userspace without a file system.

Latency. Figure 6 shows single-threaded read and write latency and throughput for different block sizes. BypassD achieves lower latency and higher bandwidth than all kernel approaches. Compared to sync and libaio, BypassD improves the read latency by 30.5% on average and write latency by 27.8%. BypassD avoids context switches, bypasses the VFS and file system layer in the kernel, uses polling instead of interrupts and has an optimized software stack.

io_uring avoids context switches by using a kernel thread that polls for I/O requests. Figure 6 shows that io_uring improves latency over sync and libaio but cannot perform as well as userspace approaches. The latency overhead in io_uring can be attributed to the kernel software stack.

BypassD's performance is very close to that of SPDK. BypassD has slightly higher latency due to VBA translations.

Figure 7 shows the amount of time spent in user, kernel and device while performing random reads. In the sync baseline, the amount of time spent in the kernel is significant for small reads which is exactly what BypassD aims to overcome. In BypassD, very little time is spent in the UserLib while accessing the device. The majority of the time is spent in copying data between user and DMA buffer.

VBA translation latency sensitivity study. As mentioned in Section 6.1, we estimated VBA translation overheads in

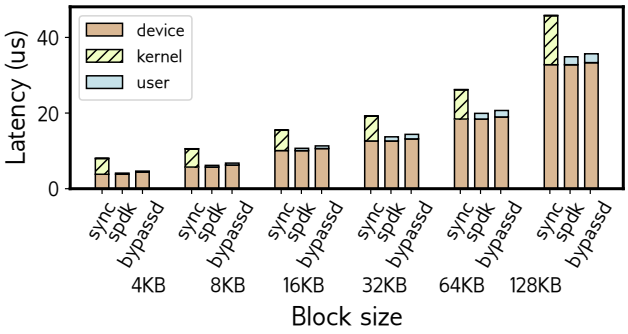

Figure 7. Random read latency breakdown.

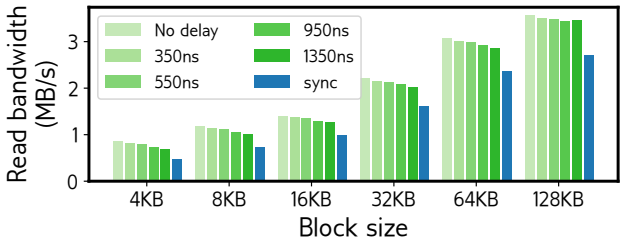

Figure 8. Effect of VBA translation latency on single thread read bandwidth.

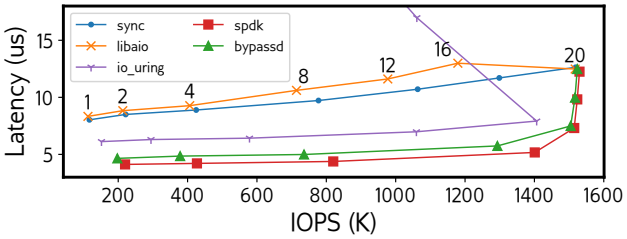

Figure 9. Random read latency and IOPS with increasing number of threads. Number of threads is indicated by annotations on the graph.

BypassD to be around 550ns. However, translation overheads could vary across systems. The latency of access increases with slower translations. Figure 8 shows the effect of translation latency on the single thread read bandwidth. As expected, increase in translation latency causes a slight decrease in bandwidth. Even with a translation latency of 1.3us, BypassD achieves significantly higher bandwidth than the baseline.

This experiment also shows caching FTEs in the IOTLB is not critical: The difference in bandwidth when FTEs are cached (350ns) versus not cached (550ns) is minimal.

Scaling. Figure 9 shows I/O latency and IOPS scaling with threads. For lower thread counts, SPDK and BypassD achieve lower latency than kernel approaches. At higher thread count (>12), the device reaches saturation at which point the I/O latency is dominated by the device time and the software overheads become negligible. BypassD does not provide any benefits when accessing an overloaded device. This aligns

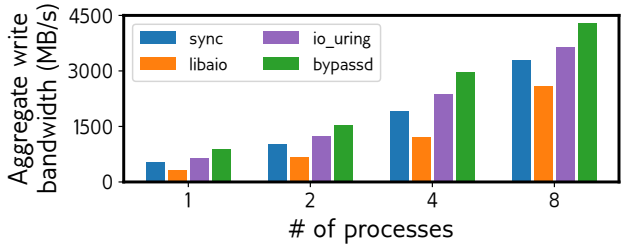

Figure 10. Aggregate write bandwidth when device is shared between multiple writer processes.

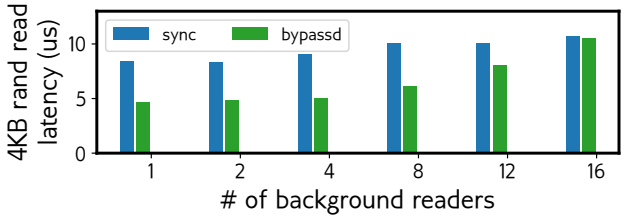

Figure 11. I/O scheduling in the device: Random read latency with multiple background reader processes.

with past research [62] on Optane SSD that suggests that the overall load needs to be moderated to achieve low latency.

Performance of io_uring drops drastically after 12 threads. This is because io_uring requires additional cores on which kernel threads poll for I/O requests [16]. io_uring needs twice as many cores to achieve performance close to BypassD.

Unlike kernel approaches, BypassD observes constant access latency until the device is saturated (threads>8). This is because UserLib avoids synchronization costs by allocating private queues and buffers to each thread. BypassD would incur very small overheads due to synchronization if multiple threads had to share queues and/or DMA buffers.

Device sharing. Figure 10 shows the total write throughput when multiple processes are accessing the device. Read performance is similar. Each process accesses a private file to avoid any contention. Notice that there are no bars for SPDK since multiple processes cannot access the device simultaneously with SPDK. On the contrary, in BypassD, each process can access the device directly from its userspace. Therefore, each process experiences lower I/O access latency and the total throughput improves. Furthermore, all processes achieve identical latency and throughput. In short, BypassD provides the performance benefits even when the device is being shared between multiple processes.

It is important to achieve fairness while sharing a device. Kernel I/O schedulers distribute the device throughput among applications. Since BypassD bypasses the kernel, it relies on the I/O scheduling in the device to balance device time among different applications. Figure 11 shows the average latency of 4KB random read latency when sharing the device with different number of reader processes. BypassD achieves latency lower than the baseline even when there

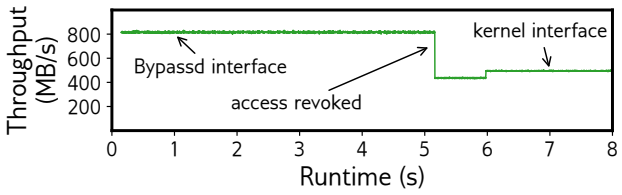

Figure 12. Read throughput of a process over time. Process starts with *BypassD interface*. At 5s, its access gets revoked and switches to *kernel interface*.

File size	Default open (us)	Open + Warm fmap (us)	Open + Cold fmap (us)
4KB	1.28	1.96	2.68
1MB	1.38	1.96	3.67
64MB	1.74	2.76	85.51
256MB	1.59	5.79	333.93
1GB	1.80	17.94	1330.75
16GB	2.10	259.94	21197.88

Table 5. fmap() overheads in BypassD.

are 16 other process reading from the device. NVMe devices implement a round-robin scheduling across queues. These results show that this device-side I/O scheduling is good enough to balance the load. If necessary, devices could implement more sophisticated schedulers [59] to achieve better fairness when accessing device directly from userspace.

Figure 12 depicts the behaviour of a system when direct access is revoked (Section 4.5). A reader process initially accesses the device through *BypassD interface*. During its execution, another process opens the file in buffered mode, and the kernel revokes the direct access for the first process. The reader process then accesses the device through the default *kernel interface* and experiences a drop in throughput.

fmap() overheads. In BypassD, the UserLib fmap()s a file before accessing it from userspace. If the file table entries are cached in DRAM, then the kernel only has to attach them to page tables during fmap() (warm fmap). As explained in Section 4.1, the file system populates the file tables if they do not already exist (cold fmap). Table 5 shows the latency of warm and cold fmap() for different file sizes in BypassD. The overhead of warm fmap is negligible unless the file is huge (GBs). Cold fmap()s are costly even for files that are a few MB in size. The cost of a cold fmap() is amortized by the low-latency direct I/O accesses. For instance, with a 16GB file, BypassD benefits an application issuing more than ~5000 I/Os. The delay of cold fmap() could be hidden by issuing kernel-interface operations during startup until fmap() completes, and then switching to BypassD-interface operations or by building partial file tables. We do not currently implement this.

Memory overheads. To avoid the cost of a cold fmap(), BypassD caches pre-populated file table entries in memory.

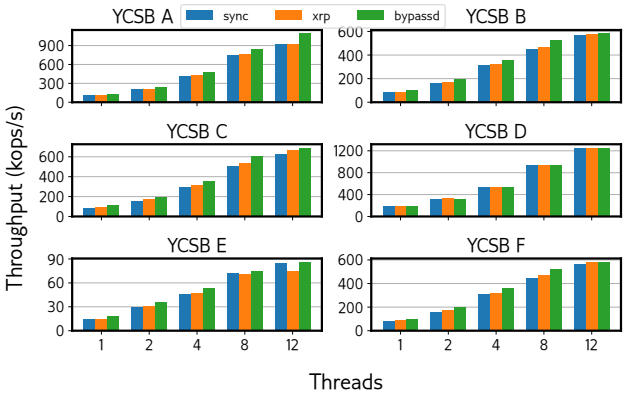

Figure 13. WiredTiger throughput scaling with threads.

This adds memory overheads in a system with thousands of files. Each file table entry is the same size as a regular page table entry which is 8 bytes long. A 4KB page can hold 512 file table entries. Optane SSD uses a page size of 4KB, so 512 FTEs would map 2MB of a file. In other words, every 2MB of a file adds a modest memory overhead of 4KB (0.2% overhead).

6.4 Production workload

WiredTiger. WiredTiger is a NoSQL storage engine used by MongoDB [5]. It uses an LSM tree to store data in multiple levels and each level is a single file. The files are indexed using a B-tree with the key-value pairs stored in the leaf nodes. WiredTiger caches the B-tree in memory to avoid issuing an I/O on every access. WiredTiger issues multiple I/Os to read multiple pages in the B-tree chain while traversing it.

We configure the WiredTiger B-tree page size to 512B which is equal to Optane SSD's block size. We create a store with 1 billion key-value pairs, key and value sizes set to 16B, resulting in a database of 46GB. We set the cache size to 6GB.

We compare the performance of BypassD against XRP, a state-of-the art research proposal that overcomes kernel software stack overheads [70]. XRP uses eBPF scripts to run user-defined storage functions from a hook in the kernel driver. XRP accelerates operations, such as index lookups and aggregation, that issue back-to-back I/Os.

Figure 13 shows the throughput of baseline, XRP and BypassD for different YCSB workloads. BypassD improves throughput by 18% on average over baseline and 13% over XRP. The improvement is larger at smaller thread count. At higher thread count, although BypassD improves I/O latency, the WiredTiger cache becomes the point of contention which hides the benefits of faster I/O. BypassD does not provide benefits for YCSB D workload which is an insert heavy workload. YCSB D follows a distribution where the newly inserted key-value pairs are the most popular ones. Therefore, it spends very little time on I/O. With YCSB E, a single I/O returns multiple key-value pairs and consecutive I/Os are not issued. XRP cannot therefore improve performance. BypassD is able to accelerate all I/O and improves performance.

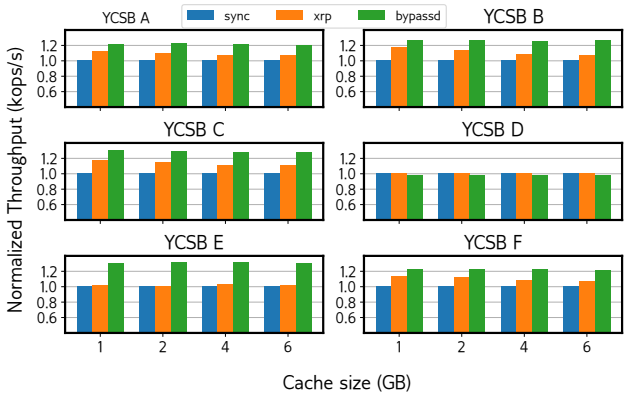

Figure 14. WiredTiger single-thread throughput with different cache sizes, relative to synchronous kernel I/O.

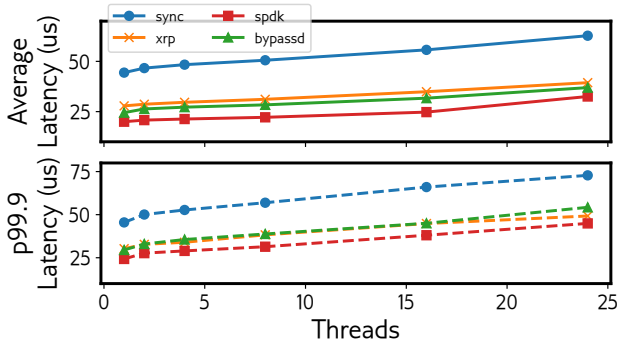

Figure 15. Avg and p99.9 request latency in BPF-KV.

Figure 14 shows WiredTiger throughput normalized to baseline for different cache sizes. As the cache size increases, fewer B-tree pages miss in the cache while traversing the B-tree. The benefits of XRP therefore decreases as it cannot issue consecutive I/Os bypassing the kernel software stack, whereas BypassD accelerates every I/O operation providing consistent improvements irrespective of cache size.

6.5 Research prototypes

We experiment on state-of-the-art research prototypes that achieve much better performance than production software when using more advanced I/O methods.

BPF-KV. It is a key-value store designed to evaluate the performance of XRP [70]. It uses a B+-tree to locate objects in the store which are stored in an unsorted log. The index and log are stored in a single large file.

BPF-KV is configured to have fixed-sized keys (8B) and values (64B). Each index node in BPF-KV is 512B. We create a store with 920 million objects that results in a 6-level index. We disable caching to focus on the overheads of reading data from disk. Each object lookup requires 7 I/Os, 6 for reading an index node at each level of the B+-tree and one for reading the final data.

Figure 15 shows the average and 99.9th percentile latency for retrieving objects from the key value store. As expected, the Linux baseline has the highest latency. XRP goes through the kernel software stack only once and issues subsequent I/Os from the driver. BypassD does not have to go to the kernel even once so achieves slightly lower latency than XRP. Compared to SPDK, BypassD takes an additional 550ns for VBA translation on every request. For a lookup involving 7 I/Os, BypassD takes about 4us more than SPDK.

On average, BypassD improves the throughput (not shown in Figure) by 72% over baseline and 9.6% over XRP.

KVell. KVell [38] is a fast persistent key-value store designed to take full advantage of modern storage device characteristics. KVell leverages random access performance to avoid sorted data on disk and batches I/O operations to utilize the high throughput that these devices offer. KVell compares favorably against production KVs, including RocksDB and WiredTiger.

We initialize a KVell database with 50 million objects, and set the key size to 16B and value size to 1024B resulting in a database of size 54GB. We evaluate the performance using YCSB read/write workloads. By default, KVell uses asynchronous I/O (libaio) to perform I/O. We measure the performance of KVell with a queue depth of 1 (KVell_1) and 64 (KVell_64). We also implemented a synchronous I/O interface and evaluated the performance of BypassD.

Figure 16 shows the throughput and latency of requests. BypassD achieves more throughput than KVell_1 but lesser than KVell_64. KVell_64 achieves high throughput at the cost of latency. BypassD improves the latency over KVell_64 by two orders of magnitude. For YCSB A, which is a 50/50 read/write workload, BypassD achieves throughput close to that of KVell_64 while reducing the latency. This is because of a bottleneck in ext4 while handling concurrent writes to the same file [44, 51]. BypassD avoids this bottleneck since writes are issued directly from userspace. For YCSB B and

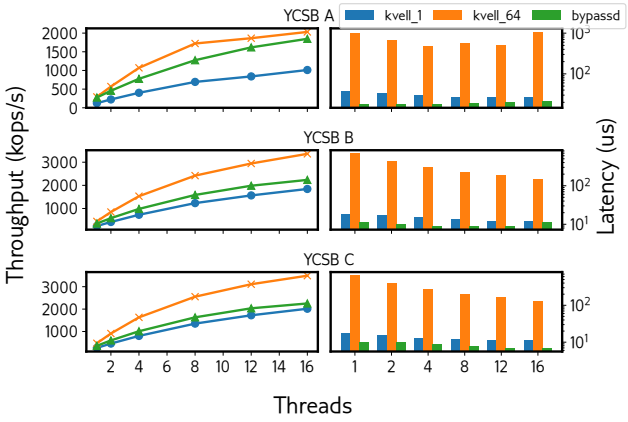

Figure 16. KVell throughput and latency for YCSB.

	Low latency	Sharing	No device overheads
Kernel file system [10, 26, 37, 66]	✗	✓	✓
Userspace drivers [34, 65]	✓	✗	✓
Hybrid solutions [11, 33, 51]	✓	✓	✗
BypassD	✓	✓	✓

Table 6. Comparison of BypassD with prior approaches.

C workloads, which are read heavy, BypassD outperforms KVell_1 by 33% and 24% respectively.

7 Related work

Recent advancements in storage technology such as NVM [28] and low-latency SSDs [4, 13, 27] have placed a spotlight on the kernel overheads and motivated researchers to address the I/O architecture. Table 6 summarizes prior approaches.

Kernel optimizations. The most straight-forward approach to overcoming the software bottleneck is to optimize the kernel I/O software stack. Researchers have proposed several optimizations to the kernel I/O stack: i) reduce interrupt overheads [10, 58], ii) optimize the block layer [10, 26, 66], and iii) overlap computations with device activity [37]. These solutions improve the performance of a specific layer in the I/O stack, but they cannot eliminate the overheads entirely. For instance, these solutions have to pay the price of context switching and the overheads of the VFS layer in the kernel.

Userspace access. Another approach is to access devices from userspace without going through the kernel software stack [11, 18, 32, 34, 35, 49, 60, 63, 65, 69]. These solutions avoid the high cost of context switching and generally provide high performance compared to traditional kernel file systems. Some of these designs require byte-addressable storage such as NVM [18, 32, 35, 60, 63, 69] and do not support block storage devices. SPDK [65] and NVMeDirect [34] provide a userspace I/O framework for NVMe devices but require developers to implement own file systems. More importantly, they do not support device sharing as the driver is mapped into a user process which has complete control over the device. For example, SpanDB [12] uses a low latency SSD to store LSM tree and write-ahead logs. It builds a file system called TopFS on top of SPDK to access low latency SSDs and sacrifices device shareability. Applications like SpanDB can benefit from BypassD as it provides low latency access without requiring new file systems and allows the device to be shared.

MonetaD [11] is the closest work to BypassD. Like BypassD, MonetaD accesses the device from userspace and enforces protection in the hardware. The kernel installs permission records into a table on the device. The device uses this table to validate accesses from userspace. However, i) updating the permission data on the device is expensive as the device pauses servicing requests, ii) MonetaD performs poorly under fragmentation, and iii) it suffers from high tail latency when the permission check misses in the table on the device. BypassD overcomes the above limitations and achieves the same goals through a less intrusive design.

Arrakis [49] uses SR-IOV [30] to achieve device sharing. A device presents itself as a *Virtual Function (VF)* to each VM. Arrakis proposes significant changes to the device, and trusts applications with metadata updates or to serve file operations that may be unsafe in many environments.

uFS [40] accesses devices from userspace using a file system micro-kernel. It uses a trusted process to submit requests to the device. Similar to io_uring, uFS requires additional cores on which uServer threads are pinned for high performance. Demikernel [67] proposes a flexible datapath architecture for kernel-bypass devices.

Device file systems. A third approach to avoiding the kernel overheads is to push the file system to the device [33, 51]. Since the file system resides on the device, processes can access the device directly from userspace. These designs offer good performance but require significant changes to the device firmware and rely heavily on devices' compute power and memory. BypassD does not require any significant compute power on the device.

Near-storage compute. A completely different way to avoid the kernel is to reduce the number of times processes go to the device. Several systems propose offloading their storage functions to the device [17, 22, 31, 53, 55, 56, 61, 68]. These solutions rely heavily on the computing capabilities of the device. They also require rewriting applications.

XRP. XRP [70] pushes the storage function to the kernel device driver. It uses BPF scripts to run user defined functions in the driver. XRP accelerates applications that perform chained IO, for example B-tree traversal. XRP requires significant effort for porting new applications and only works with data structures that have a fixed layout on disk.

8 Conclusion

With devices getting faster, software overheads dominate the access latencies. BypassD overcomes the bottleneck by providing direct access from userspace. Unlike prior userspace solutions [34, 65], BypassD allows multiple users or applications to share the device securely by offloading permission checks to the IOMMU hardware. BypassD improves I/O latency by up to 45% compared to Linux and performs close to SPDK which is known to achieve the lowest latency.

Acknowledgments

We would like to thank our shepherd Huaicheng Li and other anonymous reviewers for their feedback. We would like to thank Dr. Arkaprava Basu for his support and guidance. This work was supported in part by PRISM, one of seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA, and by NSF grant CNS 1900758.

References

[1] [n. d.]. File timestamps. https://www.gnu.org/software/coreutils/manual/html_node/File-timestamps.html.

[2] [n. d.]. mmap(2) — Linux manual page. https://man7.org/linux/man-pages/man2/mmap.2.html.

[3] [n. d.]. NVMe specifications. https://nvmexpress.org/specifications/.

[4] [n. d.]. Toshiba XL-Flash. https://www.kioxia.com/en-jp/about/news/2019/20190806-1.html.

[5] [n. d.]. WiredTiger storage engine. https://www.mongodb.com/docs/manual/core/wiredtiger/.

[6] Chloe Alverti, Vasileios Karakostas, Nikhita Kunati, Georgios Goumas, and Michael Swift. 2022. DaxVM: Stressing the Limits of Memory as a File Interface. In *55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[7] Nadav Amit, Muli Ben-Yehuda, and Ben-Ami Yassour. 2010. IOMMU: Strategies for mitigating the IOTLB bottleneck. In *International Symposium on Computer Architecture (ISCA)*.

[8] Jens Axboe. 2005. Fio-flexible i/o tester synthetic benchmark. *URL https://github. com/axboe/fio (Accessed: 2015-06-13)* (2005).

[9] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. 2017. Attack of the killer microseconds. *Commun. ACM* (2017).

[10] Adrian M. Caulfield, Arup De, Joel Coburn, Todor I. Mollow, Rajesh K. Gupta, and Steven Swanson. 2010. Moneta: A High-Performance Storage Array Architecture for Next-Generation, Non-volatile Memories. In *43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[11] Adrian M. Caulfield, Todor I. Mollov, Louis Alex Eisner, Arup De, Joel Coburn, and Steven Swanson. 2012. Providing Safe, User Space Access to Fast, Solid State Disks. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[12] Hao Chen, Chaoyi Ruan, Cheng Li, Xiaosong Ma, and Yinlong Xu. 2021. {SpanDB}: A fast,{Cost-Effective}{LSM-tree} based {KV} store on hybrid storage. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. 17–32.

[13] Samsung Corp. [n. d.]. Samsung Z-SSD. https://semiconductor.samsung.com/ssd/z-ssd/.

[14] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*.

[15] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 53, 1 (2008).

[16] Diego Didona, Jonas Pfefferle, Nikolas Ioannou, Bernard Metzler, and Animesh Trivedi. 2022. Understanding modern storage APIs: a systematic study of libaio, SPDK, and io_uring. In *Proceedings of the 15th ACM International Conference on Systems and Storage (SYSTOR)*.

[17] Jaeyoung Do, Sudipta Sengupta, and Steven Swanson. 2019. Programmable solid-state storage in future cloud datacenters. *Commun. ACM* (2019).

[18] Mingkai Dong, Heng Bu, Jifei Yi, Benchao Dong, and Haibo Chen. 2019. Performance and protection in the ZoFS user-space NVM file system. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*.

[19] Daniel Ehrenberg. [n. d.]. The Asynchronous Input/Output (AIO) interface. https://github.com/littledan/linux-aio.

[20] Exim Internet Mailer [n. d.]. Exim Internet Mailer. http://www.exim.org/.

[21] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. 2022. Direct access,{High-Performance} memory disaggregation with {DirectCXL}. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 287–294.

[22] Boncheol Gu, Andre S Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, et al. 2016. Biscuit: A framework for near-data processing of big data workloads. (2016).

[23] Tejun Heo, Dan Schatzberg, Andrew Newell, Song Liu, Saravanan Dhakshinamurthy, Iyswarya Narayanan, Josef Bacik, Chris Mason, Chunqiang Tang, and Dimitrios Skarlatos. 2022. IOCost: block IO control for containers in datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[24] Jian Huang, Anirudh Badam, Moinuddin K. Qureshi, and Karsten Schwan. 2015. Unified Address Translation for Memory-Mapped SSDs with FlashMap. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*.

[25] Jack Tigar Humphries, Kostis Kaffes, David Mazières, and Christos Kozyrakis. 2021. A case against (most) context switches. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS)*.

[26] Jaehyun Hwang, Midhul Vuppalapati, Simon Peter, and Rachit Agarwal. 2021. Rearchitecting Linux Storage Stack for μs Latency and High Throughput.

[27] Intel Corp . [n. d.]. Intel Optane P5800X SSD. 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.

[28] Intel Corp. [n. d.]. Intel Optane Persistent Memory. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[29] Intel Corp. 2018. Intel Scalable I/O Virtualization. https://www.intel.com/content/www/us/en/developer/articles/technical/introducing-intel-scalable-io-virtualization.html.

[30] Intel Corp. 2018. Recent Enhancements in Intel Virtualization Technology for Directed I/O (Intel VT-d). https://01.org/blogs/ashokraj/2018/recent-enhancements-intel-virtualization-technology-directed-i/o-intel-vt-d.

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

[32] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. 2019. SplitFS: Reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*.

[33] Sudarsun Kannan, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, Yuangang Wang, Jun Xu, and Gopinath Palani. 2018. Designing a true direct-access file system with DevFS. In *16th USENIX Conference on File and Storage Technologies (FAST)*.

[34] Hyeong-Jun Kim, Young-Sik Lee, and Jin-Soo Kim. 2016. NVMeDirect: A user-space I/O framework for application-specific optimization on NVMe SSDs. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*.

[35] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. 2017. Strata: A cross media file system. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*.

[36] Gyusun Lee, Wenjing Jin, Wonsuk Song, Jeonghun Gong, Jonghyun Bae, Tae Jun Ham, Jae W. Lee, and Jinkyu Jeong. 2020. A Case for Hardware-Based Demand Paging. In *ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*.

[37] Gyusun Lee, Seokha Shin, Wonsuk Song, Tae Jun Ham, Jae W Lee, and Jinkyu Jeong. 2019. Asynchronous I/O stack: A low-latency kernel I/O stack for ultra-low latency SSDs. In *USENIX Annual Technical Conference (ATC)*.

[38] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. 2019. Kvell: the design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating*

Systems Principles (SOSP).

[39] Huaicheng Li, Daniel S Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, et al. 2023. Pond: CXL-based memory pooling systems for cloud platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 574–587.

[40] Jing Liu, Anthony Rebello, Yifan Dai, Chenhao Ye, Sudarsun Kannan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2021. Scale and performance in a filesystem semi-microkernel. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP)*.

[41] LWN.net. 2021. Shared Virtual Addressing. https://lwn.net/Articles/747230/.

[42] Moshe Malka, Nadav Amit, Muli Ben-Yehuda, and Dan Tsafrir. 2015. rIOMMU: Efficient IOMMU for I/O devices that employ ring buffers. *ACM SIGPLAN Notices* 50, 4 (2015), 355–368.

[43] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. 2023. TPP: Transparent page placement for CXL-enabled tiered-memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 742–755.

[44] Changwoo Min, Sanidhya Kashyap, Steffen Maass, and Taesoo Kim. 2016. Understanding manycore scalability of file systems. In *USENIX Annual Technical Conference (ATC)*.

[45] MySQL [n. d.]. MySQL. https://www.mysql.com/.

[46] Shweta Pandey, Aditya K Kamath, and Arkaprava Basu. 2023. Scoped Buffered Persistency Model for GPUs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 688–701.

[47] Anastasios Papagiannis, Giorgos Xanthakis, Giorgos Saloustros, Manolis Marazakis, and Angelos Bilas. 2020. Optimizing Memory-mapped I/O for Fast Storage Devices. In *USENIX Annual Technical Conference (ATC)*.

[48] pcisig.com. 2021. Address Translation Services. https://members.pcisig.com/wg/PCI-SIG/document/download/8255.

[49] Simon Peter, Jialin Li, Irene Zhang, Dan RK Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. 2015. Arrakis: The operating system is the control plane. *ACM Transactions on Computer Systems (TOCS)* (2015).

[50] Charles Reiss, Alexey Tumanov, Gregory R Ganger, Randy H Katz, and Michael A Kozuch. 2012. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the third ACM symposium on cloud computing*.

[51] Yujie Ren, Changwoo Min, and Sudarsun Kannan. 2020. CrossFS: A cross-layered direct-access file system. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[52] RocksDB [n. d.]. RocksDB. http://rocksdb.org/.

[53] Zhenyuan Ruan, Tong He, and Jason Cong. 2019. INSIDER: Designing In-Storage Computing System for Emerging High-Performance Drive. In *USENIX Annual Technical Conference (ATC)*.

[54] Samsung. [n. d.]. Samsung Z-SSD. https://lwn.net/ml/linux-fsdevel/20190112213011.1439-1-axboe@kernel.dk/.

[55] Robert Schmid, Max Plauth, Lukas Wenzel, Felix Eberhardt, and Andreas Polze. 2020. Accessible near-storage computing with FPGAs. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys)*.

[56] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. 2014. Willow: A User-Programmable SSD. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[57] Dimitrios Skarlatos, Qingrong Chen, Jianyan Chen, Tianyin Xu, and Josep Torrellas. 2020. Draco: Architectural and operating system support for system call security. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[58] Amy Tai, Igor Smolyar, Michael Wei, and Dan Tsafrir. 2021. Optimizing Storage Performance with Calibrated Interrupts. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[59] Arash Tavakkol, Mohammad Sadrosadati, Saugata Ghose, Jeremie Kim, Yixin Luo, Yaohua Wang, Nika Mansouri Ghiasi, Lois Orosa, Juan Gómez-Luna, and Onur Mutlu. 2018. FLIN: Enabling fairness and enhancing performance in modern NVMe solid state drives. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 397–410.

[60] Haris Volos, Sanketh Nalli, Sankarlingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael M Swift. 2014. Aerie: Flexible file-system interfaces to storage-class memory. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys)*.

[61] Mark Wilkening, Udit Gupta, Samuel Hsia, Caroline Trippel, Carole-Jean Wu, David Brooks, and Gu-Yeon Wei. 2021. RecSSD: near data processing for solid state drive based recommendation inference. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[62] Kan Wu, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2019. Towards an unwritten contract of Intel Optane SSD. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*.

[63] Jian Xu and Steven Swanson. 2016. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*.

[64] Jisoo Yang, Dave B Minturn, and Frank Hady. 2012. When poll is better than interrupt.. In *10th USENIX Conference on File and Storage Technologies (FAST)*.

[65] Ziye Yang, James R Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E Paul. 2017. Spdk: A development kit to build high performance storage applications. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*.

[66] Young Jin Yu, Dong In Shin, Woong Shin, Nae Young Song, Jae Woo Choi, Hyeong Seog Kim, Hyeonsang Eom, and Heon Young Yeom. 2014. Optimizing the block I/O subsystem for fast storage devices. *ACM Transactions on Computer Systems (TOCS)* (2014).

[67] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, et al. 2021. The demikernel datapath os architecture for microsecond-scale datacenter systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP)*.

[68] Jian Zhang, Yujie Ren, and Sudarsun Kannan. 2022. FusionFS: Fusing I/O Operations using CISCOps in Firmware File Systems. In *20th USENIX Conference on File and Storage Technologies (FAST)*.

[69] Shengan Zheng, Morteza Hoseinzadeh, and Steven Swanson. 2019. Ziggurat: A Tiered File System for Non-Volatile Main Memories and Disks. In *17th USENIX Conference on File and Storage Technologies (FAST)*.

[70] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, et al. 2022. XRP: In-Kernel Storage Functions with eBPF. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

A Artifact Appendix

A.1 Abstract

This artifact includes the source code of BypassD which includes three components: Linux kernel, a kernel module and the user library (UserLib). Table 2 summarizes the lines of coded added/modified in BypassD. Changes related to the file system and file table entries are part of the Linux kernel. The kernel module supports allocating DMA buffers and mapping NVMe queue pairs to the userspace. UserLib is a shim library that intercepts system calls and handles file reads and writes directly from userspace.

This artifact also includes the scripts and tools to evaluate BypassD and reproduce some of the key results in this paper. We open-source this artifact to allow other researchers and developers to use and improve it in their own work. In Section A.7, we describe the steps to use BypassD with a new workload. In this appendix, we briefly describe the steps to compile and install BypassD, and to reproduce the results.

A.2 Artifact check-list (meta-information)

- **Compilation:** BypassD includes several software components, each of which is compiled separately. All of them use `make` build tool along with `gcc/g++`. The scripts for compilation handle the dependencies required for compilation.
- **Data set:** No external dataset. The experimental setup will create some arbitrary files with garbage data.
- **Hardware:** Preferably Intel Xeon CPU with atleast 24 cores (including Hyper-Threading) with atleast 32GB of memory, and importantly, an Intel Optane P5800X NVMe SSD.
- **Run-time environment:** Experiments are carried out on a bare-metal instance. All experiments are to be run on the custom Linux kernel included in the repository.
- **Execution:** The repository contains scripts to generate the results presented in the paper. Running the experiments is as simple as launching a script.
- **Output:** Output of all experiments are stored under a `results/` sub-directory that is created by the scripts. The output contains stats about I/O latency and throughput. The scripts will also plot graphs similar to the ones in the paper. The graphs are saved as pdfs under the experiment's sub-directory.
- **Experiments:** Using storage I/O workloads with FIO tester suite [8], the experiments exhibit the performance benefits (latency and bandwidth) that can be achieved using BypassD under different scenarios.
- **How much disk space required (approximately)?:** About 12GB for the source code and about 16GB for experiment files on the NVMe device used for evaluation.
- **How much time is needed to prepare workflow (approximately)?:** <30 minutes.
- **How much time is needed to complete experiments (approximately)?:** About 60 minutes.
- **Publicly available?:** Yes, all of the source code relating to BypassD is publicly available at https://github.com/multifacet/Bypassd and also archived on Zenodo.
- **Code licenses (if publicly available)?:** GNU GPL.
- **Archived (provide DOI)?:** BypassD artifact is divided across 3 components:
  1. Main repository with scripts: https://doi.org/10.5281/zenodo.10069841
  2. BypassD kernel and module: https://doi.org/10.5281/zenodo.10038719
  3. UserLib: https://doi.org/10.5281/zenodo.10038717

A.3 Description

A.3.1 How to access. The artifact is publicly available on GitHub at https://github.com/multifacet/Bypassd. This repository contains all of the source code related to BypassD and links (submodules) to dependencies. The repository is also archived on Zenodo and can be accessed at https://doi.org/10.5281/zenodo.10069841.

A.3.2 Hardware dependencies. While BypassD itself is not tied to any particular hardware architecture, the scripts in the repository are written for an Intel CPU. The preferred hardware configuration for evaluation would be similar to that described in Section 6: Intel Xeon CPU with atleast 24 cores (with Hyper-threading) and 32GB of memory, interfacing a low latency Intel P5800X SSD. The source code (repository) and dependencies require 12GB of storage space.

The experiments have to be carried out a bare-metal machine after installing the custom Linux kernel included in the repository. It is recommended to turn off CPU frequency scaling and run the cores at maximum frequency. This artifact includes scripts to turn off CPU frequency scaling for Intel CPUs (except those using p-states).

A.3.3 Software dependencies. BypassD has been built with Linux-5.4 (included in the artifact). However, the changes can be applied to more recent Linux versions but will require manual effort. This artifact has been tested on Ubuntu-20.04 which includes `gcc-9.4` and `binutils-2.34`.

The artifact contains links to external code-bases including SPDK [65] and fio tester suite [8]. These packages in-turn have other dependencies. Please refer to their documentation for more details.

The graphs are plotted with `Python3` using `matplotlib`.

A.4 Installation

The README.md in the repository lists the steps to setup BypassD and run experiments which is summarized below.

1. Clone the repository and initialize the submodules.
   ```
   $> git clone https://github.com/multifacet/Bypassd
   $> cd Bypassd
   $> git submodule update –init –recursive
   ```
2. Build and install the custom Linux kernel included in the repository. The repository also includes a sample config file for reference.
   ```
   $> bash utils/build_Linux_kernel.sh
   ```

3. Build dependencies: fio and SPDK.
   `$> bash utils/build_fio.sh`
   `$> bash utils/build_spdk.sh`
4. Reboot into the built custom kernel.
   `$> sudo grub-reboot "Advanced options for Ubuntu>Ubuntu, with Linux 5.4.0"`
   `$> sudo reboot`
5. The final step is to build and load the kernel module. This is optional as the evaluation scripts take care of loading and unloading the module.
   `$> cd kernel/module`
   `$> make`
   `$> sudo insmod bypassd.ko`

A.5 Experiment workflow

The `artifact_evaluation/` directory in the repository contains scripts to run experiments and generate the graphs. These scripts configure and compile the kernel module and UserLib. They also mount/unmount the device and enable or disable cpu frequency scaling. They launch all experiment runs needed for one figure. The results are generated in `results/` sub-directory. The scripts finally plot graphs similar to those in the paper.

1. Run the scripts by providing the device file under `/dev` and a mountpoint to mount the device.
   `$> bash run_exp.sh /dev/nvme0n1 /mnt/bypassd`

A.6 Evaluation and expected results

The scripts inside the evaluation directory will generate graphs that are very similar to the ones in the paper. There could be slight variations due to the performance of the device, CPU performance and CPU operating frequency.

Results can vary significantly for some reasons:

1. CPU frequency scaling is not disabled.
2. Instead of a low latency SSD such as the Intel Optane P5800X, an SSD with higher latency is used.
3. The system doesn't have enough cores (atleast 20).

A.7 Experiment customization

One could use the repository in a stand-alone fashion to run BypassD with any application to improve its I/O performance. This can be done by configuring and compiling the UserLib separately. Refer to `utils/enable_bypassd.sh` for more information on compiling the UserLib. You can then run any workload with these steps:

1. Ensure the custom kernel is installed.
2. Load the BypassD kernel module using `sudo insmod bypassd.ko`.
3. Run any application with the UserLib as the shim library: `sudo LD_PRELOAD=./libshim.so <app>`