SubZero: Zero-copy IO for Persistent Main Memory File Systems

Juno Kim
UC San Diego
juno@eng.ucsd.edu

Yun Joon Soh
UC San Diego
yjsoh@eng.ucsd.edu

Joseph Izraelevitz
University of Colorado, Boulder
joseph.izraelevitz@colorado.edu

Jishen Zhao
UC San Diego
jzhao@eng.ucsd.edu

Steven Swanson
UC San Diego
swanson@eng.ucsd.edu

ABSTRACT

POSIX-style `read()` and `write()` have long been the standard interface for accessing data in files. However, the data copy into and out of memory these methods require imposes an unnecessary overhead when files are stored in fast persistent memories (PMEMs). To avoid the copy, PMEM-aware file systems generally provide direct-access (DAX)-based `mmap()`, but in doing so force the programmer to manage write-atomicity and concurrent accesses to the file.

In this work, we propose two new system calls – `peek()` and `patch()`, and collectively called SubZero – that read and update PMEM-backed files *without any copies*. To show its potential, we implemented SubZero in two state-of-the-art PMEM file systems, XFS-DAX and NOVA. Measurements of simple benchmarks show that SubZero can outperform copy-based `read()` and `write()` by up to 2× and 2.8×, respectively. At the application level, `peek()` improves GET performance of the Apache Web Server by 3.6×, and `patch()` boosts SET performance of Kyoto Cabinet up to 1.3×.

CCS CONCEPTS

• **Information systems → Phase change memory**; **Record storage systems**.

KEYWORDS

Persistent Memory; Non-volatile Memory; Direct Access; DAX; File Systems

ACM Reference Format:
Juno Kim, Yun Joon Soh, Joseph Izraelevitz, Jishen Zhao, and Steven Swanson. 2020. SubZero: Zero-copy IO for Persistent Main Memory File Systems. In *11th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '20), August 24–25, 2020, Tsukuba, Japan.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3409963.3410489

1 INTRODUCTION

POSIX `read()` and `write()` have been the most common interface for accessing file contents for many years and across many generations of storage hardware. The semantics of these system calls rely crucially on copying data between (volatile) memory and a storage medium (e.g., a disk).

Copy-based semantics are a natural fit for both the performance characteristics and hardware interface of conventional storage technologies. Disks are slow enough that the overhead of the copy is not significant. And even if disks were fast, processors cannot operate directly on the data they hold, because they are not memory.

The copy-based, atomic semantics that `read()` and `write()` provide are also convenient for the programmer. The copy that `read()` creates will not change if the file it reads is overwritten, and `write()` atomically transfers a fully-prepared buffer of data into the file.

The appearance of fast, persistent memory (PMEM) that resides on the processor's memory bus, however, upends both of these long-standing assumptions: PMEM is fast enough that the overhead of a copy is detrimental to performance and the copy is not necessary since the data is already directly-accessible to the processor.

Despite the costs of copy-based operations and because of their convenience and ubiquity, even file systems specifically designed for PMEM [9, 10, 15, 21, 23–25] implement POSIX-compliant `read()` and `write()`. That said, PMEM file systems generally also provide direct-access (DAX) `mmap()` as an alternative that dispenses with the copy overhead. However, DAX-`mmap()` forces the programmer to implement atomicity and concurrency control manually, complicating the programming model. Moreover, its unclear interaction with `read()` and `write()` has discouraged them from taking the potentially interesting path that leverages both interfaces in a single program.

To bridge the gap between POSIX `read()` and `write()` and DAX `mmap()`, we propose a new IO interface called SubZero that does not rely on copy-based semantics for data access but still preserves the ease of use that `read()` and `write()` provide. SubZero offers DAX-like speed with a simple, POSIX-like interface that interacts cleanly with the legacy POSIX interface.

Concretely, SubZero provides two new system calls – `peek()` and `patch()` – that give programs access to file data and interoperate cleanly with `read()` and `write()` system calls. The `peek()` system call returns the virtual address of a memory region holding a snapshot of requested file contents. The snapshot is atomic with respect to other file operations and its contents do not change if the underlying file changes. The `patch()` system call takes a pointer to a memory region containing new data and atomically incorporates that region into the target file. `patch()` causes the memory region to become read-only.

A PMEM file system can implement `peek()` by simply manipulating the program's page table. It can implement `patch()` by adjusting the file's layout to incorporate the patched pages, as long as the pages are in persistent memory. Benefiting from `peek()` and `patch()` requires changes to how the application accesses file data and how it allocates and disposes of IO buffers.

We implemented SubZero in two of the state-of-the-art PMEM file systems, XFS-DAX and NOVA. Our measurements show that SubZero outperforms copy-based `read()` and `write()` by up to 2× and 6×, respectively. At the application level, `peek()` improves GET performance of the Apache Web Server by 3.6×, and `patch()` boosts the SET performance of Kyoto Cabinet up to 1.3× with non-invasive changes.

The remainder of the paper is organized as follows. Section 2 describes the motivation of our work. Section 3 describes the details of SubZero IO interface and semantics. Section 4 discusses the key implementation details and Section 5 evaluates these techniques with micro-benchmarks and applications. Section 6 discusses related works and Section 7 concludes.

2 MOTIVATION

Although conventional POSIX `read()` and `write()` interfaces have proven easy-to-use, they become a major source of inefficiency in PMEM file systems since the media latency of PMEM is low relative to the software overheads on top of PMEM. In the PMEM file system software stack, the copies inherent in the semantics of `read()` and `write()` system calls are a major source of inefficiency.

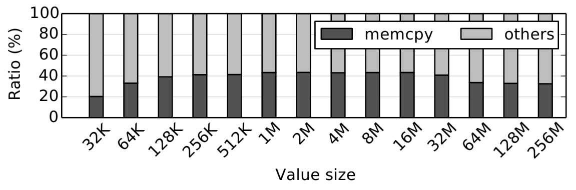

Figure 1: Memory copy overhead in Kyoto Cabinet. Memory copy overheads are significant when updating large key-value pairs in Kyoto Cabinet. "others" means application and file system level overheads except for the memory copy.

For example, a simple data copying step between the user buffer and PMEM pages during `write()` system call takes 20 – 45% of the total execution time when updating large key-value pairs in Kyoto Cabinet database [11] backed by the NOVA file system (Figure 1). Critically, this copying overhead is a property of the interface, not the file system — all PMEM file systems that implement POSIX IO incur this performance penalty.

PMEM file systems provide DAX-`mmap()` as an alternative to avoid this overhead. However, this interface jettisons the good with the bad. Although it eliminates all common-case file system overheads for data access, it also forces the application to manage crash consistency and concurrency control on its own, complicating the programming model and inviting bugs.

As a solution, SubZero bridges the gap between POSIX IO and DAX IO by combining the advantages of both, offering DAX-like speed with a simple, POSIX-like interface that interacts cleanly with itself and the legacy POSIX interface.

3 SUBZERO IO

SubZero IO (or just SubZero) is a suite of new system calls that avoids copy-based semantics and allows for more efficient data access and modification in PMEM-based file systems. SubZero strives to provide simple semantics that are easy for programmers to reason about when building sophisticated applications. To these ends, SubZero has the following design goals:

- **Zero data movement** SubZero should not require costly data movements to implement read and write operations. In particular, SubZero performs *no* data movement while the conventional "zero-copy" IO (i.e., conducting IO with `O_DIRECT`) in disk-based file systems still requires one data movement between the storage media and memory.
- **Atomicity** SubZero should provide atomicity guarantees similar to those for POSIX `read()` and `write()`, since those guarantees have proven useful in building IO-intensive applications.

	Function	Semantics
Read	`void* peek(int fd, off_t pos, size_t len)`	Open a read-only mapping to the target range of a PMEM file.
	`int unpeek(void *addr)`	Close a mapping opened by `peek()`.
Write	`int patch(int fd, void *buf, size_t len, off_t pos)`	Update a target file with a PMEM buffer. `patch()` returns an error when `buf` is misaligned with `pos` or the buffer and the target file do not belong to the same file system.
	`int create_pmem_pool(char *path, size_t size)`	Create a PMEM pool from which PMEM pages are allocated and return a unique pool id.
	`void delete_pmem_pool(int pool_id)`	Delete a PMEM pool.
	`void* alloc_pmem(int pool_id, off_t pos, size_t len)`	Allocate a PMEM buffer. `pos` is the target location where the buffer will be patched.
	`void free_pmem(void* buf)`	Reclaim and unmap a PMEM buffer. PMEM pages are reclaimed to the allocator only when they do not belong to files.

Table 1: SubZero IO functions. The API includes replacements for conventional `read()` and `write()`, in addition to ancillary functions for allocating PMEM buffers for IO. Only `peek()`, `unpeek()`, and `patch()` need to be system calls.

- **Clean integration with POSIX** Combined with their atomicity, SubZero should integrate cleanly with POSIX `read()` and `write()` operations. This allows programmers to freely intermingle those conventional IO operations with SubZero operations.

Below, we describe the SubZero IO interface at a high level and discuss its semantics in more detail. Then we present an example of its use and discuss the changes it requires to existing programs.

3.1 The SubZero Interface

SubZero introduces two new IO operations: `peek()` and `patch()`. Table 1 summarizes the SubZero interface, which includes these two and several ancillary functions. Below we describe the interface in detail.

`peek()` The `peek()` system call returns a pointer to a memory region that contains the contents of a file at a particular file offset. The region reflects a snapshot of the file contents at the time `peek()` is executed. The snapshot is atomic with respect to file modifications (e.g., `write()` or `patch()`). The snapshot is immutable, so attempts to alter its contents result in a segmentation fault. There are no alignment restrictions (or guarantees) on the file offset or the returned pointer.

Since `peek()` allocates the memory region containing the snapshot, the application must eventually release the memory by passing the snapshot address to `unpeek()`.

Lines 1–3 in Figure 2 illustrate how to `peek()` an entire file. Line 9 deallocates the resulting buffer with `unpeek()`.

The `peek()` system call resembles DAX-`mmap()`, since both map file contents into the user address space. However, there are two key differences. First, `peek()` does not impose any alignment restrictions on the file offset of the region to be peeked, while `mmap()` requires the offset to a multiple of the file system page size. Similarly, `unpeek()` relaxes `munmap()`'s alignment restriction. Second, `peek()` is easier to use than `mmap()`, since the snapshot is explicitly atomic relative to other file modifications and immutable.

`patch()` The `patch()` system call modifies a file by merging the contents of a buffer into a file at the given offset. In essence, the buffer *becomes* part of the file rather than being copied into it. After the `patch()`, the buffer becomes immutable. The state of the `patch()`'d buffer is identical to the state of a `peek()`'d buffer: both are immutable mappings of a file's contents. The `patch()`'d buffer is closed by calling `free_pmem()`.

The change that `patch()` makes to the file is atomic with respect to other `patch()` and `write()` operations. Like `write()`, its effects are not guaranteed to be permanent until the program calls `fsync()` or `fdatasync()`.

The benefit of `patch()` is realized when two conditions are satisfied. First, the buffer should comprise PMEM that is managed by the same file system instance as the file being written to. A program can acquire such PMEM by creating a temporary file in the same file system and `mmap()`ing it. Second, the buffer should be *access-aligned*. Intuitively, this means the page boundaries of the buffer must align with the page boundaries in the file. That is, for a `patch()` operation using a buffer `B` and a file offset `off` on a file system with page size `S`, the `patch()` is access-aligned if `B % S == off % S`.

```
 1  int in_fd = open("/mnt/in.uu", O_RDONLY);                        // Open the input file
 2  int input_length = lseek(in_fd, 0L, SEEK_END);                   // Find its length
 3  char *in_buf = peek(in_fd, 0, input_length);                     // Peek its contents
 4  int pool_id = create_pmem_pool("/mnt", 1073741824);              // Create a pool
 5  void *out_buf = alloc_pmem(pool_id, 0, input_length);            // Allocate an access-aligned buffer
 6  int output_length = uudecode(in_buf, out_buf, input_length);     // Construct uudecoded output in the buffer
 7  int out_fd = open("/mnt/out.dat", O_WRONLY);                     // Open the output file
 8  patch(out_fd, out_buf, output_length, 0);                        // Patch the uudecoded output into the file
 9  unpeek(in_buf);                                                  // Unpeek the input
10  free_pmem(out_buf);                                              // Unmap the output buffer
11  delete_pmem_pool(pool_id);                                       // Delete the pool
```

Figure 2: Computation between two PMEM files without any copies using SubZero. An application can use `peek()` and `patch()` to access and update files without any copies. Here, `uudecode()` reads directly from the input file's pages and writes its result directly to the physical pages that will become the output file.

The requirement is similar to the alignment requirement imposed by opening a file with `O_DIRECT`.

To make it easy for a program to satisfy these criteria, we have implemented a simple allocator to allocate and deallocate access-aligned buffers. Program can create a pool in a file system and use it to allocate buffers for `patch()` operations.

Lines 4-8 of Figure 2 demonstrate how `patch()` can work with `peek()` to avoid any unnecessary copies. The code allocates a PMEM buffer and calls `uudecode()` to populate it by processing the `peek()`'d contents of the input. The `patch()` on line 8 causes the output buffer to become the contents of the file, and therefore no copies are necessary.

3.2 Using SubZero

SubZero removes the implicit copy from the semantics of accessing and modifying a file's contents. Realizing its benefits will require changes to how applications perform IO.

The most significant changes affect how the program allocates and manages IO buffers. First, the program can no longer pre-allocate read buffers, since `peek()` allocates and returns a populated buffer. The program will also need to call `unpeek()` when it is finished with the buffer. Second, the application needs to allocate write buffers from PMEM.

How invasive this change is will depend on how the application performs writes. High-performance applications that maintain their own page-aligned buffer pools and leverage `O_DIRECT` will have less trouble since page-aligned buffers are automatically access-aligned. Applications that perform more ad-hoc writes will need to allocate an access-aligned buffer for each write.

4 IMPLEMENTING SUBZERO

To illustrate its potential, we implemented SubZero in two state-of-the-art, in-kernel PMEM file systems: NOVA [23], a file system built from scratch for PMEM, and XFS-DAX, a linux file system adapted to accomodate direct access to PMEM. SubZero can be implemented without invasive changes if the file system has the ability to 1) allow multiple files to share data pages and 2) support copy-on-write updates when a write modifies shared pages.

4.1 NOVA file system

NOVA [23] is a log-structured file system for persistent memory. It manages a contiguous region of PMEM and presents a POSIX-compatible file system interface. It supports DAX-style `mmap()`, and all file and directory operations are atomic.

NOVA stores per-inode log that contains *write entries* pointing to data pages and describing the file's layout. To perform a write, NOVA allocates new pages, populates them, and then appends a write entry to the log incorporating the pages into the file. Some old pages may become obsolete as a result — NOVA reclaims these during garbage collection. Log append is atomic, so writes are atomic as well. On file open, NOVA scans the log and builds an in-DRAM index that maps file offsets to physical pages.

`peek()` in NOVA The implementation of `peek()` in NOVA mirrors its implementation of `mmap()` with a few additions. Since `peek()` maps the target pages of the file into the application's address space, the implementation must protect against two types of unexpected changes to the `peek()`'d data. First, it must prevent stores by the application from altering the underlying file. Second, it must prevent changes to the file from altering the data visible to the program.

To address the first, NOVA maps the pages as read-only so that attempts to alter the contents from the `peek()`'d address will see the segmentation fault.

To address the second, NOVA does not modify data in place, so the PMEM pages that `peek()` mapped remain unchanged even if the file's contents change. NOVA must take care, however, to prevent the mapped pages from being reclaimed by garbage collection until they are `unpeek()`'d.

`patch()` in NOVA Patch allows programs to insert populated buffers of data directly into a file. This requires solving two problems.

The first is implementing `alloc_pmem()` to allocate buffers in PMEM that are suitable for being `patch()`'d into the file. We solve this by building a userspace library that creates temporary files in PMEM and maps them with DAX-`mmap()` to return pointers to programs that call `alloc_pmem()`.

The second is performing the `patch()` itself. In NOVA, `patch()` is a special case of a normal `write()`: `write()` allocates PMEM pages, initializes them, and appends a write log entry to add the new pages to the file. A `patch()` uses the pages provided by the application, skips the allocation and initialization, and appends the write entry.

Implementing `alloc_pmem()` and `patch()` this way means the pages used by `patch()` belong to two files – the temporary file used to allocate the buffer and the target file. Similar to `peek()`, the shared pages should be protected from updates occurring either by modifying the buffer with stores or by modifying the target file with `write()`. To support this, NOVA sets the pages as read-only after they are used by `patch()`, and always performs copy-on-write on modification by `write()`. For non-page aligned accesses and patch sizes, it may be required to overwrite an additional page before and/or after the patched pages. This is easily done by appending additional log entries describing these overwrites.

4.2 XFS-DAX file system

XFS is a widely-deployed, high-performance journaling file system. It organizes files with variable-sized extents and maintains a B+tree for each inode to map file offsets to those extents. As an additional mode, XFS-DAX allows direct access to the extents in PMEM, bypassing the page cache layer.

Although XFS-DAX, by default, updates data in-place, it also has facilities to support out-of-place data updates as part of its implementation of "reflink", a feature that allows multiple files to share data pages [6, 12, 19]. Reflink allows sharing pages between files and supports copy-on-write updates when the shared pages are modified, the same mechanisms required by SubZero. Therefore, implementing SubZero in XFS-DAX mainly involved enabling the reflink to work with DAX mode in addition to the page table manipulations to mark PMEM pages as read-only.

5 EVALUATION

We evaluate the performance of SubZero against copy-based `read()` and `write()`, as well as DAX-`mmap()`, on two PMEM file systems, NOVA [23] and XFS-DAX [12] under Linux kernel 4.19. We answer the following questions:

- How much speedup do `peek()` and `patch()` achieve?
- How much effort is required to modify applications to use SubZero?
- How much does SubZero improve performance on real applications?

We performed experiments on a dual-socket machine provided by Intel Corporation. The CPUs are 24-core Cascade Lake engineering samples with a similar spec as the previous-generation Xeon Platform 8160. Each core has exclusive 32 kB L1 instruction and data caches, and 1 MB L2 caches. All cores share a 33 MB L3 cache. Each CPU has two iMCs and six memory channels (three channels per each iMC), and each memory channel is attached with a 32 GB Micron DDR4 DIMM and a 256 GB Intel Optane DC Persistent Memory Module (Optane DCPMM). Overall, the system has 384 GB of DRAM and 3 TB of PMEM. Every experiment is configured to access the DRAM and PMEM in the same socket.

5.1 Micro-benchmarks

To understand how SubZero performs compared to other legacy IO operations, we compare the performance of SubZero operations against that of `read()`, `write()`, and DAX-`mmap()`-based IO methods. To calculate the latency, we read or write a large number of files with each IO method while varying the IO size from 4 kB to 4 MB. We repeated this single-thread benchmark 100 times and report the average latency.

Most importantly, understanding the performance benefit of SubZero requires investigating not only the cost of IO system calls themselves, but also that of their related operations – memory allocation, population, consumption, etc. For this, the latency in all IO methods in our experiments includes the time to allocate/free the buffer (if applicable). In case of read, the latency also includes the time to load all bytes from the target file after each IO method. In case of write, the latency also includes the time to persist all bytes to the target file.

Read Latency Figure 3 compares the latency of `peek()` against `read()` and DAX-`mmap()`-based load. Here, we differentiate two different `read()` cases: `read` denotes cases where the DRAM buffer is *not* reused for subsequent `read()` operations whereas `read-opt` represents cases where the buffer is reused either by the application or the glibc `malloc()`.

As a result, the relative speedup of `peek()` largely depends on whether the buffer is reused in using `read()`: `peek()` outperforms `read` by 1.6–2× and 1.6–1.7× in NOVA and XFS-DAX, respectively while the speedup of `peek()` over `read-opt` reduces to 6–17% in both file systems. The reduction in speedup mainly comes from the operating system

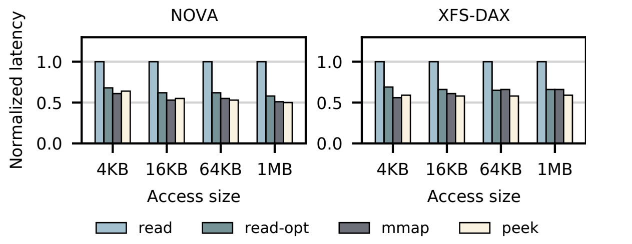

Figure 3: Read operation latency. All latencies are normalized to `read` latency. Lower is better.

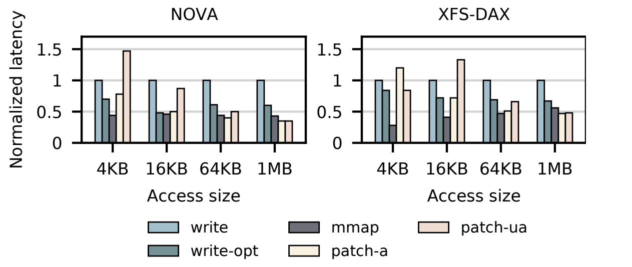

Figure 4: Write operation latency. All latencies are normalized to `write` latency. Lower is better.

overheads avoided by the buffer reuse in `read()`. Otherwise, any new buffer allocation normally requires a system call (i.e., `mmap()` or `sbrk()`) and page fault overheads. Overall, the result indicates that replacing `read()` with `peek()` is mostly beneficial when the target program does not reuse the DRAM buffer frequently or it is hard to do so due to the memory allocation pattern in the program.

Compared to `mmap()`, `peek()` shows comparable performance (-5–10%) since the underlying page table mapping mechanism is fundamentally identical.

Write Latency Figure 4 compares the latency of `patch()` against `write()` and DAX-`mmap()`-based store. As with read, `write-opt` denotes the buffer reuse case. For `patch()`, `-a` and `-ua` indicate page-aligned and unaligned access, respectively. After `write()` and `patch()`, we call `fdatasync()` to ensure the written data is made durable.

Of note, aligned `patch()` (`patch-a`) outperforms `write()` up to 2.8× and 2.2× in NOVA and XFS-DAX, respectively. The speedup is small when the access size is small, but it starts increasing as the access size increases. When the buffer is reused during `write()`, aligned `patch()` performs 10–30% slower for 4 kB, but starts outperforming from 16 kB, and achieves up to 1.7× in both file systems. Unaligned `patch()` (`patch-ua`) performs slower than aligned `patch()` due to the additional overheads from copying head and tail pages. The gap reduces to close to zero as the access size grows. 4 kB unaligned `patch()` falls back to the copy-based `write()` operation since there is no page to share.

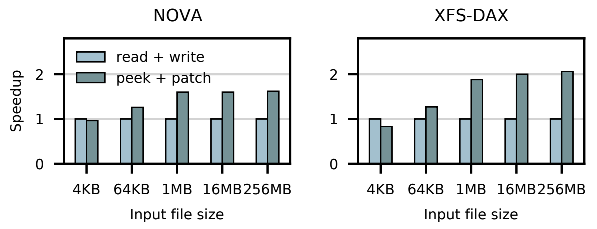

Figure 5: Boosted computation between PMEM files. Combining `peek()` and `patch()` allows computation between PMEM files without any copies, therefore boosts the performance.

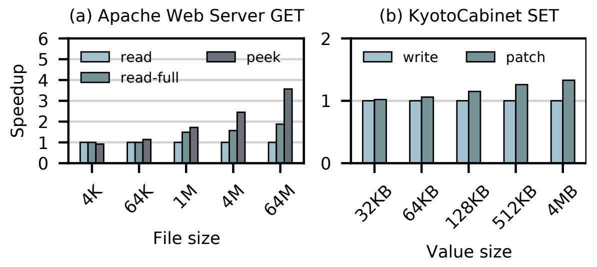

Figure 6: Application performance. SubZero boosts application performance by a wide margin with small code changes.

Compared to `mmap()`, aligned `patch()` underperforms 1.8× and 4.3× for 4 kB in NOVA and XFS-DAX, respectively, but the gap reduces as the access size grows. Beyond 64 kB, aligned `patch()` outperforms `mmap()`. The main reason is that `patch()` saves the page fault cost by using pre-faulted buffer pages whereas `mmap()` includes the cost in the critical path. Also, despite its higher speed on the smaller accesses, `mmap()` only provides the atomicity for 8 bytes while `patch()` provides the crash-consistency of each IO operation.

Uudecoding files We evaluate the example code seen in Figure 2 where a combined use of `peek()` and `patch()` can boost the performance of computation between different PMEM files. We used the same code in Figure 2 with base64 encoding schemes with results in Figure 5. As a result, using `peek()` and `patch()` over `read()` and `write()` achieved up to 1.6× and 2× speedups on NOVA and XFS-DAX, respectively.

5.2 Applications

We explored the impact of SubZero on real applications with two examples, Apache Web Server and Kyoto Cabinet.

Apache Web Server To apply SubZero to Apache Web Server, we modified the Apache Portable Runtime library, a supporting library for the web server, to use `peek()`. We measured the performance of HTTP GET request serving static files using a built-in web performance benchmark, ApacheBench [1], on our modified NOVA. When reading file contents with `read()`, Apache Web Server uses an 8 kB buffer by default. For `peek()`, we set the access size to be the same as the file size since it performs the best. Figure 6 (a) plots the throughput of GET requests normalized to the `read()`-based method. As a result, `peek()` outperforms `read()` from 1 MB (1.7×) and achieves up to 3.6× better throughput. Since the default `read()` suffers the overhead of frequent system calls, we increased the `read()`'s buffer size to be the same as the file size (`read-full`). On this setting, `peek()` still offers up to 1.9× speedup.

To benefit from these speedups, `peek()` required 48 LOC changes.

Kyoto Cabinet Kyoto Cabinet [11] (KC) is a high performance database library that stores variable-sized key-value records in a single file. KC supports fast access to the records via hash table or B+tree, and makes every operation transactional using write-ahead logging (WAL) to a separate log file. By default, KC updates the database and the WAL file using the `write()` operation. To demonstrate the benefit of SubZero, we applied `patch()` to the hash table-based HashDB database to speed up updating key-value records in the underlying database file and measured the throughput of transactional SET operations. Note that our modified KC performs unaligned patches to the database file as each record is padded with preceding metadata fields. In Figure 6 (b), `patch()` performs similar to the `write()`-based operation for value sizes 32, 64 kB, but after the 128 kB value, it begins to outperform `write()` and the speedup monotonically increases up to 1.3×.

To experience these speedups with SubZero, we required 57 LOC changes.

6 RELATED WORK

Avoiding data movement in storage systems The techniques to avoid data movement have been explored in a variety of systems, ranging from persistent storage to DRAM. For file systems, Ext4 supports an `ioctl` operation, `EXT4_IOC_MOVE_EXT`, to allow swapping extents between files by modifying inodes [2, 4]. A recent system SplitFS [13] extended this feature in Ext4-DAX but still requires the data movement that SubZero avoids since it supports copy-based `read()`, `write()` semantics. Other file systems [6, 12, 19] have similar functionality called "reflink" that could be a useful facility to implement SubZero. The technique to remap pages to avoid data movements has also been explored for flash drives [14, 16, 22]. For DRAM, operating systems, such as OSX, that inherit the Mach [5] support memory copy via `vm_copy` that remaps the regions as copy-on-write pages [3].

PMEM allocator For fast and efficient PMEM allocation, several schemes have been proposed from both industry and academia. PMDK [18] is an open-source, PMEM library bundle from Intel. It offers large virtual address pools by memory-mapping to PMEM files. Schewalb *et al.* [20] proposed a general-purpose memory allocator for PMEM that combines both DRAM and PMEM for fast allocation and recovery. Makalu [7] offers an integrated allocator and garbage collector that avoids memory leaks on failures while offering better integration with existing PMEM libraries [8]. Pallocator [17] improves defragmentation by maintaining multiple PMEM regions instead of having a single large pool. Compared to these allocators, our allocator focuses on highly fast allocations (by pre-faulting pages and partitioning) and the correct recovery of both buffer and regular files. The SubZero allocator is currently very simple; it could be improved by incorporating techniques from these projects.

7 CONCLUSION

We have described and implemented SubZero, a new IO mechanism that avoids most or all data movement for reads and writes to PMEM-backed files. In addition to minimizing movement, our implementation of SubZero provides both fast read access and strongly consistent updates. Our evaluation shows that SubZero outperforms copy-based `read()` and `write()` by a wide margin. In summary, SubZero IO is a straight-forward way for programmers to improve their applications' performance on PMEM file systems.

ACKNOWLEDGMENTS

This work was supported in part by CRISP, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. It was also supported by NSF Awards 2011212 and 1629395.

REFERENCES

[1] Apache bench (ab) - apache http server benchmarking tool. https://httpd.apache.org/docs/2.4/en/programs/ab.html.

[2] Extent swap in Ext4. https://www.kernel.org/doc/Documentation/filesystems/ext4.txt.

[3] A programmer's guide to the mach system calls. http://shakthimaan.com/downloads/hurd/A.Programmers.Guide.to.the.Mach.System.Calls.pdf.

[4] CVE-2010-2066, 2010. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-2066.

[5] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for unix development. pages 93–112, 1986.

[6] Apple File System, 2017. https://en.wikipedia.org/wiki/Apple_File_System.

[7] Kumud Bhandari, Dhruva R. Chakrabarti, and Hans-J. Boehm. Makalu: Fast recoverable allocation of non-volatile memory. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 677–694, New York, NY, USA, 2016. ACM.

[8] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 433–452, New York, NY, USA, 2014. ACM.

[9] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better I/O through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 133–146, New York, NY, USA, 2009. ACM.

[10] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System Software for Persistent Memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 15:1–15:15, New York, NY, USA, 2014. ACM.

[11] FAL Labs. Kyoto Cabinet: a straightforward implementation of DBM, 2010. http://fallabs.com/kyotocabinet/.

[12] Silicon Graphics International. XFS: A High-performance Journaling Filesystem. http://oss.sgi.com/projects/xfs.

[13] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. Splitfs: Reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 494–508, New York, NY, USA, 2019. Association for Computing Machinery.

[14] Dong Hyun Kang, Gihwan Oh, Dongki Kim, In Hwan Doh, Changwoo Min, Sang-Won Lee, and Young Ik Eom. When address remapping techniques meet consistency guarantee mechanisms. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, Boston, MA, 2018. USENIX Association.

[15] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A cross media file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 460–477, New York, NY, USA, 2017. ACM.

[16] Gihwan Oh, Chiyoung Seo, Ravi Mayuram, Yang-Suk Kee, and Sang-Won Lee. Share interface in flash storage for relational and nosql databases. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 343–354, New York, NY, USA, 2016. ACM.

[17] Ismail Oukid, Daniel Booss, Adrien Lespinasse, Wolfgang Lehner, Thomas Willhalm, and Grégoire Gomes. Memory management techniques for large-scale persistent-main-memory systems. *Proc. VLDB Endow.*, 10(11):1166–1177, August 2017.

[18] pmem.io. Persistent Memory Development Kit, 2017. http://pmem.io/pmdk.

[19] Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: The Linux B-Tree Filesystem. *Trans. Storage*, 9(3):9:1–9:32, August 2013.

[20] David Schwalb, Tim Berning, Martin Faust, Markus Dreseler, and Hasso Plattner. nvm malloc: Memory allocation for nvram. In *ADMS@VLDB*, pages 61–72, 2015.

[21] Haris Volos, Sanketh Nalli, Sankarlingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael M. Swift. Aerie: Flexible file-system interfaces to storage-class memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 14:1–14:14, New York, NY, USA, 2014. ACM.

[22] Zev Weiss, Sriram Subramanian, Swaminathan Sundararaman, Nisha Talagala, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Anvil: Advanced virtualization for modern non-volatile memory devices. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 111–118, Santa Clara, CA, 2015. USENIX Association.

[23] Jian Xu and Steven Swanson. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 323–338, Santa Clara, CA, February 2016. USENIX Association.

[24] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. NOVA-Fortis: A fault-tolerant non-volatile main memory file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 478–496, New York, NY, USA, 2017. ACM.

[25] Jian Yang, Joseph Izraelevitz, and Steven Swanson. Orion: A distributed file system for non-volatile main memory and RDMA-capable networks. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST '19)*, 2019.