Libnvmmio: Reconstructing Software IO Path with Failure-Atomic Memory-Mapped Interface

Jungsik Choi, *Sungkyunkwan University;* Jaewan Hong and Youngjin Kwon, *KAIST;* Hwansoo Han, *Sungkyunkwan University*

https://www.usenix.org/conference/atc20/presentation/choi

This paper is included in the Proceedings of the 2020 USENIX Annual Technical Conference.

July 15–17, 2020

978-1-939133-14-4

Open access to the Proceedings of the 2020 USENIX Annual Technical Conference is sponsored by USENIX.

Libnvmmio: Reconstructing Software IO Path with Failure-Atomic Memory-Mapped Interface

Jungsik Choi
Sungkyunkwan University

Jaewan Hong
KAIST

Youngjin Kwon
KAIST

Hwansoo Han
Sungkyunkwan University

Abstract

Fast non-volatile memory (NVM) technology changes the landscape of file systems. A series of research efforts to overcome the traditional file system designs that limit NVM performance. This research has proposed NVM-optimized file systems to leverage the favorable features of byte-addressability, low-latency, and high scalability. The work tailors the file system stack to reduce the software overhead in using fast NVM. As a further step, NVM IO systems use the memory-mapped interface to fully capture the performance of NVM. However, the memory-mapped interface makes it difficult to manage the consistency semantics of NVM, as application developers need to consider the low-level details. In this work, we propose Libnvmmio, an extended user-level memory-mapped IO, which provides failure-atomicity and frees developers from the crash-consistency headaches. Libnvmmio reconstructs a common data IO path with memory-mapped IO, providing better performance and scalability than the state-of-the-art NVM file systems. On a number of microbenchmarks, Libnvmmio gains up to 2.2× better throughput and 13× better scalability than file accesses via system calls to underlying file systems. For SQLite, Libnvmmio improves the performance of Mobibench and TPC-C by up to 93% and 27%, respectively. For MongoDB, it gains up to 42% throughput increase on write-intensive YCSB workloads.

1 Introduction

The recent surge of non-volatile main memory (NVM) technology such as PCM [32, 55], STT-MRAM [4, 30], NVDIMMs [45], and 3D Xpoint memory [21] allows applications to access persistent data via CPU `load`/`store` instructions directly. With the benefits of competitive performance, low power consumption, and high scalability, they are expected to complement or even replace DRAM in future systems [30, 33].

To leverage the performance and persistent features, researchers have proposed NVM-optimized file systems [8, 12, 13, 24, 28, 46, 65, 67, 68]. The most important challenge addressed in the series of work is to revise the inefficient behavior of the software IO stack, which presents a dominating overhead in fast NVM [2, 3, 9, 22, 26, 48, 69]. To reduce the overhead, state-of-the-art NVM-aware file systems discard the traditional block layer and the page cache layer in the IO path. Despite these optimizations, file accesses through the OS kernel's file system still incur significant overhead. For example, `read` and `write` system calls are still expensive ways to leverage the low latency of NVM, due to frequent user/kernel mode switches, data copies, and complicated VFS layers [7, 9, 24, 25, 27, 57, 62].

A promising approach to further reduces IO overhead of NVM file systems is to use memory-mapped IO [9, 35, 58, 60, 67, 68]. The memory-mapped IO naturally fits the characteristics of NVM. Applications can map files to their virtual address space and access files directly with `load`/`store` instructions without kernel interventions. Memory-mapped IO also minimizes the CPU overhead of file system operations by eliminating file operations such as indexing to locate data blocks and checking permissions [65]. With these benefits, the `mmap` would be a critical interface for file IO in future NVM systems.

While memory-mapped IO exposes the raw performance of NVM to applications, a lot of responsibility is laid on applications as well. One thing to keep in mind for application programmers is that memory-mapped IO does not guarantee atomic-durability. If a system failure occurs during memory-mapped IO, the file contents may be corrupted and inconsistent in the application context. In return for fast performance, developers should build application-specific crash-safe mechanisms. Cache lines should be flushed to ensure durability and memory barriers should be enforced to provide a correct persistent ordering for NVM updates. This mechanism often induces a serious software overhead, and makes it notoriously difficult to write accurate and efficient crash-proof code for NVM systems [38, 50–52, 71]. For an instance, applying cache flush and memory barrier instructions correctly in the

right locations is challenging; excessive use causes performance degradation, but omitting them in required locations leads to data corruption [39, 70]. This is the major obstacle blocking the adoption of memory-mapped IO to fully exploit the advantages of NVM.

We propose *Libnvmmio*, a user library that provides failure-atomic memory-mapped IO with `msync`. We add atomicity and ordering features to the existing `msync` at user-level. By separating failure-atomicity concerns from memory-mapped IO applications, Libnvmmio allows developers to focus on the main logic of programs. To make the `msync` failure-atomic, Libnvmmio uses user-level logging techniques. Our library stages written data to per-block, persistent logs and applies the updates to memory-mapped files in a failure-atomic manner on `msync`.

Implementing `msync` at user-level has many advantages. First, the user-level `msync` minimizes system call overhead. Existing `msync` imposes system call overhead, which takes locks and excessively serializes threads in a multi-threaded application. Second, it reduces write amplification. Kernel-level `msync` flushes rather large ranges whose size are multiples of the system page size (4KB, 2MB, or 1GB). Whereas, user-level `msync` can track dirty data at a cacheline granularity and flush them at cacheline level. Third, it avoids TLB-shootdown overhead. When applications invoke `msync` on NVM file systems, operating systems track down updated pages by searching for dirty bits in the page table and flush corresponding cache lines of those dirty pages to NVM. After the flush, they clear the dirty bits in the page table to enable tracking new updates. This incurs TLB invalidations in other cores, as dirty bit state is just kind of information in TLB along with the virtual to physical page mapping. As Libnvmmio's `msync` maintains user-level logs for update tracking, we can totally avoid TLB-shootdown overhead. Fourth, it takes advantage of non-temporal `store` instructions which bypass CPU caches with no need of cache flushing. Kernel-level `msync` flushes the entire range, even if updates are performed with non-terminal `store` instructions. In general, there is no other way to communicate with `msync` that the non-temporal stores are used. For all of these reasons, a user-level `msync` in Libnvmmio can perform better than a kernel-level `msync`.

Existing applications that use conventional file IO interface (*e.g.*, `read`/`write`, `fsync`, *etc.*) can also benefit from memory-mapped IO using Libnvmmio. Like FLEX [66] and SplitFS [24], Libnvmmio transparently intercepts the traditional file IO requests and then perform memory-mapped IO. When applications call `fsync`, Libnvmmio carries out its failure-atomic `msync`. Libnvmmio rebuilds the common IO path with efficient mechanisms for read and write performance, but the uncommon, complex file operations such as directory namespace and protection are passed to the slow path of the existing file systems.

Libnvmmio runs on any file systems that supports memory-mapped interface on NVM such as Ext4-DAX, XFS-DAX, PMFS [13], and NOVA [68]. Libnvmmio running on NOVA performs better than NOVA by 2.5× and Ext4-DAX by 1.18× in Mobibench and TPC-C.

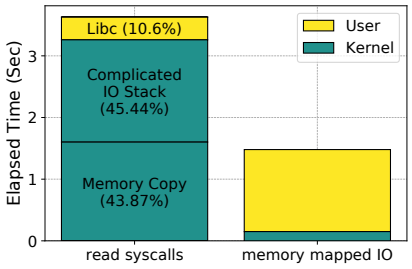

Figure 1: Read syscalls vs. memory mapped IO. Sequential read on a 16GB file. Both cases use `read` or `memcpy` to copy file data into the user buffer by 4KB.

Libnvmmio makes the following contributions:

- Libnvmmio extends the semantics of `msync`, providing failure-atomicity.
- With experimental evidences, Libnvmmio demonstrates lower-latency and higher-throughput with scalability than the state-of-the-art NVM file systems
- Design and implementation of Libnvmmio, running on Ext4-DAX, XFS-DAX, PMFS, NOVA. Libnvmmio is publicly available at: https://github.com/chjs/libnvmmio.

2 Background

2.1 Need for Memory-Mapped IO

The fundamental difference between memory-mapped IO and read-write IO is the data path. The read-write interface copies the user buffer into a kernel buffer[1], searches the file system index to locate physical block address, and performs metadata operations if necessary. Whereas, the memory-mapped interface allows direct accesses to storage, skipping the index searching and copying to the kernel buffer. The simplified data path in memory-mapped IO drastically reduces the software overhead compared to the read-write interface, which significantly improves IO performance in fast non-volatile memory. To compare the performance, we run a micro-benchmark performing sequential reads on a 16 GB file. Figure 1 shows the performance difference. Memory-mapped IO shows 2.3× better performance than the `read` system call. The `read` system calls spends 43.9% out of the IO entire latency on copying user buffers to kernel buffers and 45.4% for the rest of kernel IO stack. Memory-mapped IO eliminates most of the software overhead. We observed that the total number of instructions to execute a single read

[1] Some NVM file systems such as NOVA avoid it.

is 69× less in the memory-mapped IO than the `read` system call.

2.2 Need for Atomic Updates

Modern processors guarantee only cache-sized, aligned stores (64 bit) to be atomic. The atomicity guarantee is not sufficient for general file IO which requires more complex and larger atomic updates. On writing a 4 KB or larger block, a crash may cause partially updated states, which needs significant costs to detect and recover the block. To avoid the hassle, researchers put an effort to make large updates *failure-atomic* in non-volatile memory file systems [24, 28, 67]. Existing file systems deploy a variety of techniques to implement the failure-atomicity guarantee: copy-on-write and journaling. These techniques work in different ways, and the advantages and disadvantages in terms of performance vary.

2.2.1 Copy-on-Write

When updating a block, the Copy-on-Write (CoW) (or shadow-paging) [12, 17, 42, 56, 67, 68] mechanism creates a copy of the original page and writes the new data to the copied page rather than updating the new data in place. Not only for data update but the CoW mechanism performs the out-of-place update for index. For a tree-based indexing structure, the CoW mechanism causes a change of a child node to update its parent node in an out-of-place manner, propagating all the changes of internal nodes up to the top of the tree (called *wandering tree problem*).

The CoW mechanism induce significant software overhead when used in the NVMM system. First, CoW dramatically increases write amplification. CoW usually performs writes at the page granularity, which is a typical node size of file systems indexing. Even if only a few bytes are updated, the entire page must be written. Besides, as the capacity of main memory has increased, the utilization of hugepages (*e.g.*, 2MB or 1GB) is increasing [6, 13, 14, 29, 47, 54]. This trend makes the use of the CoW technique more costly [9]. Second, the CoW technique causes TLB-shootdown overhead in memory-mapped IO. If the CoW technique is applied to memory-mapped files, the mapping of the virtual address must be changed from the original page to the copied page, necessitating TLB-shootdown whenever an update occurs. When a CoW occurs, the kernel flushes the local TLB and send flush requests to remote cores through inter-processor interrupt (IPI). The remote cores flush their TLB entries according to the information received by the IPI and report back when completed. If the remote core has interrupts disabled, the IPI may be kept pending. The initiator core expects to receive all acknowledge the process of flushing the TLBs. This process could take microseconds, causing a notable overhead [3, 61].

2.2.2 Journaling

Journaling (or logging) is a technique that is widely used in databases [43] and journaling file systems [13, 16, 22, 34, 49, 53] to ensure data-atomicity and consistency between data and metadata. It persists a copy of new or original data before updating the original file. If a system failure occurs during writing, the valid log can be used for recovery. Two logging policies are possible: undo logging and redo logging. Redo logging first writes new data to the redo log. When the new data becomes durable in the log, the data are overwritten to the original file. If a system failure occurs while updating the file, the new data in the log can be written again to the file. For read requests, applications need to check the log first because only the log may have the up-to-date data. Undo logging first copies the original data to the log. After the original data becomes persistent, undo logging updates the new data to the file in place. If a system failure occurs during the write, undo logging allows to roll back the original data using the undo log. Because the latest data are always in the file, applications can read the data directly from the file without checking the log. Therefore, undo logging is appropriate for the applications that perform read frequently (§3.4).

Logging techniques require writing data twice: once to the log and once to the original file, which may cause software overhead. However, redo logging allows updating the original file out of the critical path of execution. Because the log has the persistent data, redo logging can postpone updating the file in the background (§3.3). Besides, logging technique is convenient to implement the *differential logging* [1, 15, 23, 36]. Unlike page-based logging, which logs an entire page, the differential logging only logs differential data at the byte-granularity. Differential logging can significantly reduce write amplification especially when it is used for byte-granularity storage devices such as NVM [27].

2.3 Atomic Update for Memory-Mapped IO

While the direct access of memory-mapped IO is essential for reducing the software overhead in NVM file system, it pushes the burden of data atomicity to the application. The POSIX `msync` primitives provides durability and consistency between data and metadata but not atomicity. To support atomicity of large updates, application developers must implement their own reliability mechanism. However, implementing the in-house mechanism is tedious and notoriously buggy [50].

Researchers have proposed adding the atomicity guarantee to the `msync` interface in traditional storage [50] and NVM [67]. To provide atomicity to memory-mapped files, they take journaling-like approaches; dirty pages are staged first and copied to the original file. Providing atomicity at the kernel-level has a fundamental limit which impacts good performance. For example, NOVA [67] creates a replica page on a page fault and maps the replica page on the faulting

virtual address. On `msync`, kernel copies the replica page to the original page atomically. The minimum unit of copying is a page size (4 KB or 2 MB), which causes write amplification for small IO requests.

3 Libnvmmio

The purpose of Libnvmmio is eliminating software overhead, while providing low-latency, scalable file IO with ensured data-atomicity. Libnvmmio is linked with applications as a library, providing the efficient IO path by using the `mmap` interface. In particular, Libnvmmio has following design goals and implementation strategies.

Low-latency IO. Reducing software overhead is crucial to take advantage of low latency NVM. Since Libnvmmio aims to make the common IO path efficient for low-latency IO, it avoids using the complicated kernel IO path including the slow journaling for common cases.

Efficient logging for data atomicity. Libnvmmio transparently intercepts file APIs and provides atomicity for data operations by using logging. As sustaining low-latency file IO is essential, Libnvmmio endeavors to minimize write amplification and software overhead for data logging.

High-throughput, scalable IO with high concurrency. To sustain high throughput across different IO sizes, Libnvmmio uses varying sizes of log entries depending on IO sizes. To this end, Libnvmmio deploys a flexible data structure for indexing the log entries and handles various *log entry* sizes. Additionally, Libnvmmio aims to achieve high concurrency through fine-grained logging and scalable indexing structure.

Data-centric, per-block based organization. Libnvmmio constructs most of its data structures and metadata as data-centric. For example, Libnvmmio builds per-block logs and metadata rather than per-thread or per-transaction based logs. Data-centric design allows a single instance of a data structure and metadata for a corresponding data block. The singleton design makes it easy to coordinate shared accesses with locks. As multiple threads access the same large file concurrently in recent applications, they require more fine-grained locks than entire file locks [40]. With fine-grained locks at block level, Libnvmmio achieves scalability for data-centric logging. Per-inode logging improves scalability, when multiple accesses are performed on different files [67, 68]. However, it provides a limited degree of scalability for multiple accesses to the same file.

Transparent to underlying file systems. On top of existing NVM file systems, Libnvmmio improves the performance

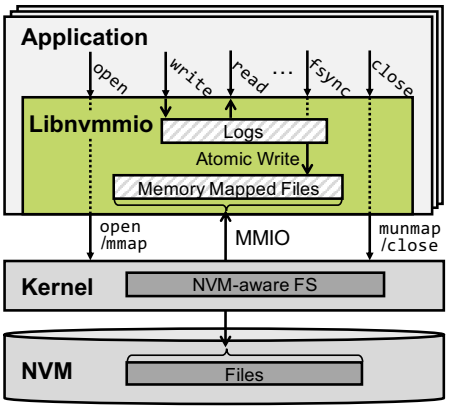

Figure 2: Libnvmmio Overview

for common data IO, keeping POSIX interfaces unchanged. For complex, uncommon IO operations, Libnvmmio leverages rich, well-tested features of existing file systems. Without breaking POSIX semantics, Libnvmmio offers extended POSIX APIs to applications for additional features. For example, POSIX semantics does not guarantee atomicity of `mmap`. While atomicity is useful, not all files need atomic update guarantees — it is unnecessary for temporal files. Libnvmmio extends `open` API to let applications indicate atomicity guarantee in a per-file basis. To communicate with the kernel, Libnvmmio translates the extended APIs to the conventional APIs with additional flags. With such a user-level extension design, Libnvmmio runs on any NVM file systems that support DAX-mmap, while enjoying file-system specific features such as fast snapshot and efficient block allocation.

3.1 Overall Architecture

Libnvmmio runs in the address space of a target application as a library and interacts with underlying file systems. Libnvmmio intercepts IO requests and turns them into internal operations. For each IO request, Libnvmmio distinguishes data and metadata operations. For all data requests, Libnvmmio services them in the user-level library, bypassing the slow kernel code. Whereas, for complex metadata and directory operations, Libnvmmio lets the operations be processed by the kernel. This design is based on the observation that data updates are the common, performance-critical operations. On the other hand, the metadata and directory operations are relatively uncommon and include complex implementation to support POSIX semantics. Handling them differently, the architecture of Libnvmmio follows the design principle of making the normal case fast [31] with a simple, fast user-level implementation.

Figure 2 shows the overall architecture of Libnvmmio. When an application opens a file, Libnvmmio interposes the `open` call with a user-level `open` API. Within the `open` API, it maps the whole content of the file onto the user memory

space and initializes *per-file metadata* (§3.5). The metadata Libnvmmio initializes includes inode number, logging policy, epoch number, *etc*. After the initialization, it returns the file descriptor to the application.

Memory-mapped IO. To directly access the NVM, Libnvmmio maps the file via `mmap` system call. Libnvmmio intercepts and replaces `read` calls with `memcpy`, and `write` calls with a non-temporal version of `memcpy` that uses the `movnt` instruction. There are two reasons why the memory-mapped IO allows faster NVM access than the traditional kernel-served read and write method. First, when persisting and obtaining data, the simple, the fast code path in Libnvmmio replaces the complex, slow kernel IO path [24, 28]. Second, `read` and `write` system calls involve indexing operations to locate physical blocks, which causes a non-trivial software overhead for fast NVM accesses. Whereas, in memory-mapped IO, the kernel searches the complex index when it maps the file blocks to the user address space on page faults. After the mapping is established, Libnvmmio can access the file data simply with offset in the memory-mapped address, eliminating the indexing operations in the steady state. Besides, finding file blocks through virtual addresses is offloaded to the MMU (*e.g.*, page table walkers, TLBs). Therefore, it reduces a sizable amount of the CPU overhead caused by file indexing [65].

Atomicity and durability with user-level logging. On `SYNC`[2] calls, Libnvmmio flushes the cache data and stores the data to NVM atomically via the logging mechanism. All write data are firstly persisted to the user-level log and later they are copied (called *checkpoint*) to the memory-mapped file. Data from both `write` and `memcpy` interfaces goes down the same path.

Providing atomicity via the user-level logging has several advantages over the kernel-level design. Using the user-level IO information, Libnvmmio can leverage the byte-addressability of NVM to log data in the fine-grained unit. On the other hand, in the kernel-level approach, the logging unit should be a page size, as `msync` relies on the page dirty bit to log the memory-mapped data, causing write amplification in case of small writes (*i.e.*, less than a page size). After `msync` is done, kernel must clear the dirty bit in the page table followed by TLB shootdown. However, user-level design uses own data structure to track dirty data without relying on the page dirty mechanism, saving unnecessary TLB shootdowns.

Application transparency. For applications using `read` and `write`, Libnvmmio can transparently replace them with the memory mapped IO operations. For applications using `mmap`, Libnvmmio can redirect the memory operations to NVM memory-mapped IO operations without effort.

[2]This term means both `fsync` and `msync`.

Providing atomic-durability on top of the `mmap` interface makes the case challenging, as Libnvmmio cannot distinguish the `memcpy` operations that requires atomic-durability from the ones that do not require.

Guaranteeing atomicity to all IO operations is prohibitively expensive. Some IO requests do not need atomicity such as logging internal traces or errors. To address the problem, Libnvmmio exposes two version of `memcpy`: POSIX version and Libnvmmio version. Libnvmmio versions are prefixed with `nv` (*e.g.*, `nvmmap`, `nvmemcpy`, `nvmunmap`, *etc.*) and provide atomic-durability. Libnvmmio avoids intrusive modifications of existing applications in order to use the Libnvmmio APIs. Instead, we instrument the application binary with an in-house tool, which lists the files the application accesses and asks developers which files need atomic-updates. With the list of files requiring atomic-durability, we patches the binary to use Libnvmmio APIs. In most cases, applications use `read`, `write`, or `memcpy` APIs, which are easy to patch for the application binary. However, in case of manipulating files with pointers, we need source-level modifications (*e.g.*, 182 lines in the MongoDB MMAPv1 engine).

3.2 Scalable Logging

Applications such as in-memory database and key-value stores, that benefit from Libnvmmio, require high concurrency level to sustain high throughput. Libnvmmio responds to the high concurrency requirement with scalable logging that is based on per-block data logging and indexing.

3.2.1 Scalable per-block logging

Finding proper logging granularity is necessary to achieve high concurrency. Application-centric techniques such as per-thread and per-transaction logging are widely adopted in databases, providing high concurrency. However, these techniques rely on the strong assumption that data is only visible and applicable to the current thread or transaction; *e.g.*, data in logs need not to be shared among threads or transactions, which is guaranteed by isolation property. Logging without needing to consider shared data allows for high scalability. However, the assumptions do not hold in general IO cases; sharing IO data among threads is a common use case. Moreover, the transaction boundary is not visible to the current design of Libnvmmio.

Instead, Libnvmmio performs data-centric logging. It divides the file space into multiple file blocks (4 KB$\sim$2 MB) and creates a log entry for each file block. Log entries in Libnvmmio are visible to all threads. The fine-grained, per-block logging allows a flexible way to share data among threads. When an update is made to a mapped file, Libnvmmio creates a log entry indexed by the offset, where the update occurred in the memory-mapped file. If other threads read the updated offset, it serves data from the log entry instead of the original

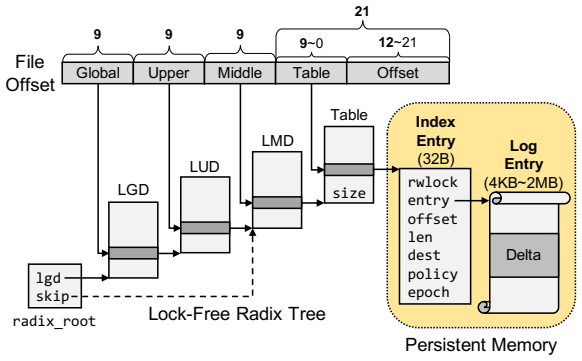

Figure 3: Indexing structure of Libnvmmio.

mapped file. When another update comes to the same file offset, it overwrites the update in the existing log entry. For shared data reads, per-block logging provides better performance than per-thread logging, as per-thread logging needs to search all the logs of all threads to gather all the updates made to the same file blocks. In addition to per-block logging, Libnvmmio takes advantage of the byte-addressable characteristics of NVM and reduces write amplification by performing differential logging for a partial update, where the update size is smaller than log block size.

3.2.2 Scalable log indexing

Along with data logging, indexing design is also critical to achieve high concurrency. Libnvmmio uses a file offset as an index key to a log block. To index many log blocks, Libnvmmio uses multi-level indexing to reduce space overhead. Similar to the page table, it uses radix trees for indexing. Fixed-depth trees allow lock-free mechanisms, which provide better concurrency than balanced trees such as red-black trees. As balanced trees require coarse-grained locks to protect the entire trees for tree re-balancing, their algorithms severely hurt concurrency [10, 11].

Figure 3 shows the index design of Libnvmmio. Each internal node is an array of buckets pointing to the next level internal nodes. Each set of 9 bits from file offset is used to locate a bucket in a corresponding internal node. Each leaf node points to an *index entry*, where `entry` field points to *log entry*. The index entry also contains other metadata for the given file offset. Libnvmmio supports variable-size log entries for large IO requests. Log entries range from 4KB to 2MB, doubling the size. To index 4KB log entries, it uses 9 bits for `Table` and 12 bits for `Offset`. For 2 MB log entry, it uses 21 bits for `Offset` without using `Table`.

In an index entry, `offset` and `len` are used for updated data offset within a log entry and update size, respectively. If update size in `len` is smaller than the log entry size, it means the log entry contains partial updates (Delta). The log entry can hold a single delta chunk indicated by `offset` and `len`. If another delta chunk needs to be added in the same log entry, the two chunks are merged. The virtual address of the memory mapped file specified in `dest` is the location where the log will be checkpointed. The logging policy for the corresponding data is specified in `policy`, which decides whether Libnvmmio uses undo log or redo log (§3.4). To determine if the log entry should be checkpointed, the number in `epoch` is used (§3.3).

The radix tree has a fixed depth to implement a lock-free mechanism. The four-level radix tree can support 256 TiB file size, but it can cause unnecessary search overhead for small files. Libnvmmio uses a skip pointer to implement a lock-free radix tree while also reducing the search overhead. As shown in Figure 3, the `radix_root` has a `skip` field. If the file size is small, Libnvmmio uses the field to skip unnecessary parent nodes. When the file size changes, Libnvmmio can adjust the `skip` pointer.

To achieve fast indexing, Libnvmmio manages the internal nodes of the radix tree in DRAM and does not persist them to NVM. It persists only the index entries and the log entries. Libnvmmio does not need to build the entire radix tree for recovery. On a crash, it simply scans the persisted index and log entries, which are committed but not checkpointed yet. It can copy the log entries to the original file by referring the `dest` attribute in the corresponding index entries and the per-file metadata. To achieve high concurrency, Libnvmmio does not use any coarse-grained locks to update internal nodes of the radix tree. Instead, it updates each bucket of internal nodes with an atomic operation. Only when it needs to update index entry, it holds the per-entry, reader-writer lock.

3.3 Epoch-based Background Checkpointing

Log entries are committed on `SYNC`[3]. The committed log entries must be checkpointed to the corresponding memory-mapped file and cleaned. To make the checkpoint operations out of the performance critical path, Libnvmmio checkpoints the log entries in the background. It periodically wakes up checkpointing threads for copying and cleaning log entries[4]. While checkpointing, the background threads do not need to obtain a coarse-grained tree lock. This minimizes disruption on on-going `read`/`write` operations. The background threads holds a per-entry writer lock to serialize checkpoint operations and `read`/`write` requests on the log entry.

Libnvmmio uses per-block logging. When an application calls `SYNC`, it must convert many of the corresponding per-block logs to committed status. This increases the commit overhead significantly. To avoid such overhead, Libnvmmio performs committing and checkpointing based on the epoch, which increases monotonically. Libnvmmio maintains two

[3]This term means both `fsync` and `msync`.

[4]Through sensitivity studies, we configured Libnvmmio wakes up the threads every 100 microsecond.

types of epoch numbers; each index entry has an epoch number for its update log and per-file metadata carries the current global epoch number. When allocating an index entry, it assigns the current global epoch number for file to the epoch number for the index entry. Libnvmmio increases the current global epoch number, when applications issue `SYNC` calls to the file. The epoch numbers are used to distinguish committed (but yet to be checkpointed) log entries from the uncommitted ones. If a log entry has a smaller epoch number than the current global epoch number, it indicates that the log entry is committed. If the epoch number of a log entry is the same as the global epoch number, the log entry is not yet committed. Libnvmmio checkpoints only committed log entries in the background threads. After being checkpointed, log entries are cleaned and reused later.

The epoch-based approach allows fast commit of log entries, as Libnvmmio does not need to traverse the radix tree and mark log entries as committed. Instead, it simply increases the current global epoch number in the per-file metadata, which reduces `SYNC` latency greatly. Commit operations are performed synchronously and atomically, when the application calls `SYNC`. Meanwhile, checkpoint operations are done asynchronously by background threads. Consequently, there are committed logs and uncommitted logs mixed in the radix tree. When applications request writes, the corresponding log entries are overwritten for uncommitted ones. Meanwhile, Libnvmmio synchronously checkpoints the committed logs first for committed ones. After completing the checkpointing, it allocates a new uncommitted log and processes write requests.

3.4 Hybrid Logging

Libnvmmio uses a hybrid logging technique to optimize IO latency and throughput. As pointed out in §2.2.2, undo logging performs better when accesses are mostly reads, whereas redo logging is better when accesses are mostly writes. To achieve the best performance of both logging policies, Libnvmmio transparently monitors the access patterns of each file and applies different logging policies depending on current read and write intensity.

Libnvmmio maintains counters to record read and write operations for a file (§3.5). When `SYNC` is called, Libnvmmio checks the counters to determine whether which type of logging would be better for the next epoch. If the logging policy changes, Libnvmmio carries out both committing and checkpointing synchronously. `SYNC` is a clean transition point for changing the logging policy, as current log data are checkpointed and cleaned. This allows Libnvmmio to avoid complex cases where it otherwise has to maintain two log policies at the same time. The per-file, hybrid logging enables the fine-grained logging policy, allowing Libnvmmio to adopt the individually best logging mechanism for each file. By default, Libnvmmio uses undo logging. It switches to redo logging, when the ratio of write operations becomes higher than or equal to 40%. The policy for the new epoch is determined by the write ratio in the previous epoch. The threshold ratio is obtained from the sensitivity analysis in §4.2.1.

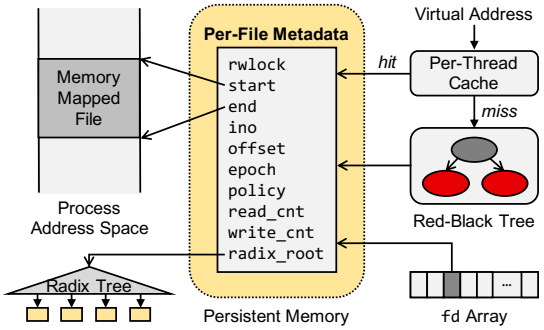

Figure 4: Per-File Metadata

3.5 Per-File Metadata

Libnvmmio maintains two types of metadata in persistent memory; the index entry is the metadata for each log entry, and the per-file metadata shown in Figure 4 is the metadata for each file. Libnvmmio stores both metadata as well as log entries in NVM, which enables Libnvmmio to recover its data in case of system failures.

When Libnvmmio accesses a file, it first gets the per-file metadata of the file and the index entry corresponding to the file offset. If applications access a file with `nvmemcpy` interface, it needs to find the per-file metadata by using access address of the `nvmemcpy`. The approach Libnvmmio takes for this purpose is to employ a red-black tree and perform range searches with virtual addresses. To speed up the search process, Libnvmmio caches recently used per-file metadata in the per-thread cache. Meanwhile, Libnvmmio can quickly obtain the per-file metadata through the file descriptor, if applications access files with `read`/`write` interface.

The per-file metadata consists of ten fields. The `rwlock` is a reader-writer lock. During `SYNC` process, this lock prevents other threads from accessing the file. The `start` and `end` fields store the location of the virtual address to which the file is mapped. The `ino` and `offset` fields record which part of a file is mapped. The `epoch` field stores the current global epoch number for the file. The `policy` field stores the current logging policy for the file. The `read_cnt` and `write_cnt` are counters of read and write operations during the current epoch, respectively. The `radix_root` field stores the root node of the radix tree indexing for index entries and log entries.

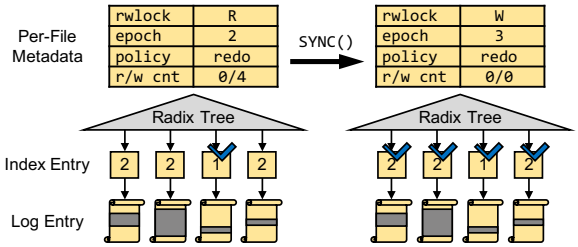

Figure 5: Epoch-based committing

3.6 Putting all together: `write` and `SYNC`

Figure 5 shows the steps of the epoch-based checkpointing in Libnvmmio. The numbers in the index entries indicate per-entry epoch numbers, and the check marks indicate their log entries are committed. A simplified version of per-file metadata is shown in tables.

Write. ① The thread holds the reader lock in the per-file metadata of the file and increases the write counter with atomic operations. Holding the reader lock in per-file metadata allows multiple threads to access the file concurrently. ② The thread traverses the in-memory radix tree to locate the corresponding index entry and holds the writer lock for the index entry. ③ Depending on the current logging policy in the per-file metadata, Libnvmmio creates an undo or redo log entry. ④ The thread writes data to the log entry with the non-temporal `store` instruction, and Libnvmmio updates the index entry of the log entry. ⑤ Libnvmmio calls `sfence` indicating logging is done and unlocks the index entry and per-file metadata, and returns to the application.

SYNC. ① Libnvmmio holds the writer lock in the per-file metadata and increases the global epoch counter by one. Holding the writer lock of the per-file metadata prevents other threads from accessing the file. ② Libnvmmio calculates the write ratio from the write and read counters. In the example in Figure 5, Libnvmmio continues to use redo logging for the next epoch, as the access pattern is write-intensive (4 writes out of 4 accesses). After determining the logging policy, Libnvmmio initializes the counters. When logging policy is unchanged, Libnvmmio lets checkpointing threads commit log entries in the background. If Libnvmmio decides to change logging policy, it synchronously checkpoints all committed log entries before the new epoch begins. ③ Finally, Libnvmmio unlocks the per-file metadata and returns to the application.

3.7 Crash Consistency and Recovery

Libnvmmio preserves write ordering of a sequence of write requests. For each write, Libnvmmio writes data to the log and flushes the CPU cache. The order-preserving write provides the prefix semantics [63], guaranteeing every thread to see a consistent version of data updates. Along with the consistency of data, Libnvmmio guarantees consistency between metadata and data. Libnvmmio maintains two persistent metadata: per-file metadata and index entries. Libnvmmio strictly orders between the sequence of [data update, index entry update] and `SYNC` call.

In the recovery phase, Libnvmmio checks whether the index entries are committed, while scanning the index entries. If Libnvmmio finds a committed log, whose epoch number is smaller than the global epoch number, it finds the per-file metadata from the index entry's `dest` attribute. Then, it redoes or undoes according to the logging policy. Libnvmmio can efficiently parallelize this recovery task by using multi-threading.

NVMM	Rand Read	Rand Write	Seq Read	Seq Write
NVDIMM-N	35.84	20.61	92.42	20.65
Optane DC	3.588	1.026	13.64	4.30

Table 1: NVMM Characteristics (GB/s)

4 Evaluation

We implemented Libnvmmio from scratch. Our prototype of Libnvmmio has a total 3,452 LOC[5] in C code. To persist data to NVM, Libnvmmio employs the PMDK library [20].

4.1 Experimental setup

To evaluate Libnvmmio on different types of NVM, we used NVDIMM-N [45] and Intel Optane DC Persistent Memory Module [19]. The system with 32GB NVDIMM-N has 20 cores and 32GB DRAM. Another system with 256GB Optane has 16 cores and 64GB DRAM. In the Optane server, we used two 128GB Optanes configured in *interleaved App Direct* mode. Table 1 shows the results of measuring the performance of each memory using Intel Memory Latency Checker (MLC) [18].

In our experiment, Libnvmmio used NOVA [68] running on Linux kernel 5.1 as its underlying file system. To compare Libnvmmio with various file systems, we experimented with four file systems: Two of these, Ext4-DAX and PMFS [13], journal only metadata and perform in-place writes for data. The two others, NOVA and SplitFS [24], guarantee data-atomicity for each operation. We configured NOVA to use CoW updates, but without enabling checksums. For SplitFS, we configured it to use *strict* mode. We ran PMFS and SplitFS on Linux kernel 4.13, and Ext4-DAX and NOVA on Linux kernel 5.1. Kernel versions are the latest versions that support the underlying file systems.

[5] we measure LOC with sloccount [64]

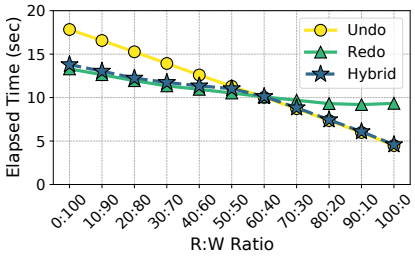

Figure 6: Performance on different logging policies

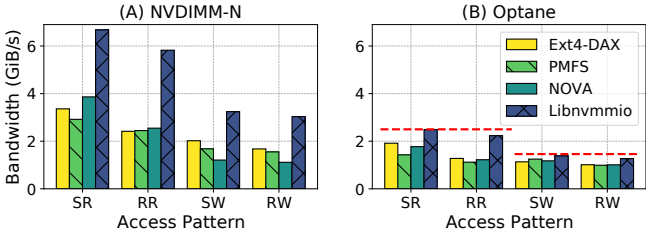

Figure 7: Performance on different access patterns

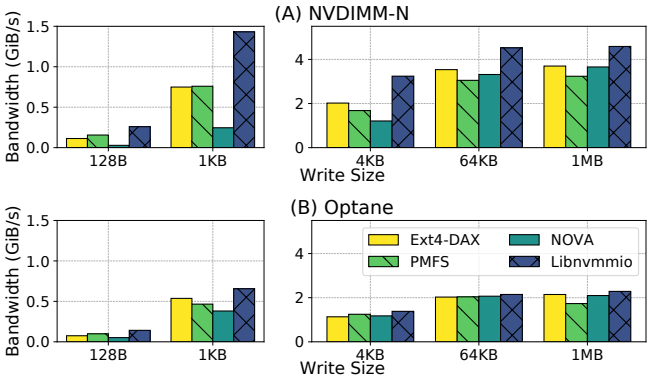

Figure 8: Performance on different write sizes

4.2 Microbenchmark

4.2.1 Hybrid logging

Most logging systems adopt only one logging policy (redo or undo). Each logging policy has different strengths and weaknesses, depending on the type of file accesses. While redo logging is better for write-intensive workloads, undo logging is better for read-intensive workloads.

Figure 6 shows how logging policies (redo, undo, and hybrid logging) affect the performance of Libnvmmio. Undo logging shows better performance than redo, when the workload has high read ratio. Redo logging shows better performance than undo, when the workload has high write ratio. When the *R:W ratio* is 60:40, the two logging policies show the same level of the performance. Based on this observation, Libnvmmio uses the ratio as a change point for its hybrid logging policy. As shown in Figure 6, hybrid logging in Libnvmmio achieves the best case performance of the two logging policies.

4.2.2 Throughput

We measured the bandwidth performance by using FIO [5]. It repeatedly accesses a 4GB file in units of 4KB for 60 seconds in a single thread. Two graphs in Figure 7 show the experiment results on NVDIMM-N (A) and Optane (B), respectively. Four file access patterns are used for our experiment: sequential read (SR), random read (RR), sequential write (SW), and random write (RW). All the other file systems except Libnvmmio perform the file IO at kernel level. Libnvmmio avoids the kernel IO stack overhead and performs file IO mostly at user level.

As shown in Figure 7, Libnvmmio provides the highest throughput on all access patterns, outperforming the other file systems by 1.66∼2.20× on NVDIMM-N and 1.14∼1.74× on Optane. The performance improvements are more noticeable in NVDIMM-N than in Optane. The maximum achievable bandwidths on Optane are 2.5GB/s and 1.46GB/s for FIO mmap based read and write without atomicity support. These are indicated as red dotted lines in Figure 7 (B). The performance results on Optane are almost near the maximum achievable bandwidths for Libnvmmio, which suggests the performance on Optane is limited by the hardware limit, not by the mechanisms in Libnvmmio.

The performance in Libnvmmio is also improved over the other file systems by maximizing logging efficiency in hybrid logging. For read access patterns (SR and RR), Libnvmmio performs only user-level `memcpy` from the memory-mapped file to the user buffer under the undo logging. For write access patterns (SW and RW), Libnvmmio updates only the log, not the memory-mapped file, under the redo logging and asynchronously writes the data from the redo log on `SYNC` call at the file close.

Figure 8 shows the performance of the FIO sequential write on various IO sizes. Libnvmmio performs per-block logging, but provides various log block sizes. With this feature, Libnvmmio can keep the high performance across different IO sizes. The performance generally improves on the increased IO sizes for all file systems and Libnvmmio, as the number of write system calls decreases within the 60 second duration of FIO experiment. Libnvmmio shows significantly higher performance than the other file systems when the IO size is smaller than the page size (128B, 1KB). This is mainly due to the differential logging feature in Libnvmmio. For file systems that use CoW for atomicity, such as NOVA, write amplification becomes a large overhead on sub-page size data writes.

Figure 9 shows the performance of the FIO sequential write on different `fsync` intervals. The horizontal axis represents the fsync frequency. For example, the interval *10* means FIO performed fsync after every ten writes issued. The performance of Ext4-DAX and PMFS slightly increased as the fsync interval increased. Since Ext4-DAX and PMFS perform

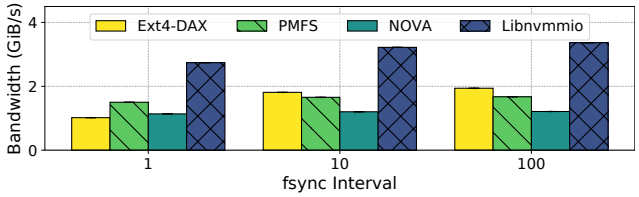

Figure 9: Performance on different `fsync` intervals

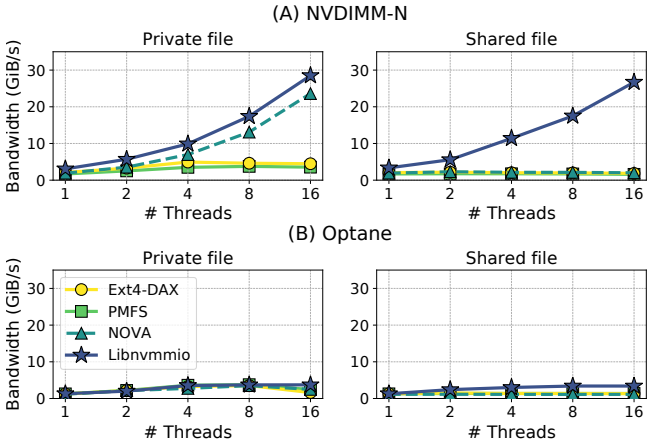

Figure 10: Scalability: FIO random write with multithreads

Latency (us)	Ext4-DAX read	Ext4-DAX write	PMFS read	PMFS write	NOVA read	NOVA write	Libnvmmio read	Libnvmmio write
Avg.	1.73	50.43	2.21	6.16	1.73	4.43	1.12	4.14
99th	3	61	3	9	3	9	2	10
99.9th	6	552	4	12	3	10	3	12
99.99th	8	605	8	239	6	15	5	15
99.999th	12	648	17	258	8	5216	7	76

Table 2: 4KB read and write latencies on Optane

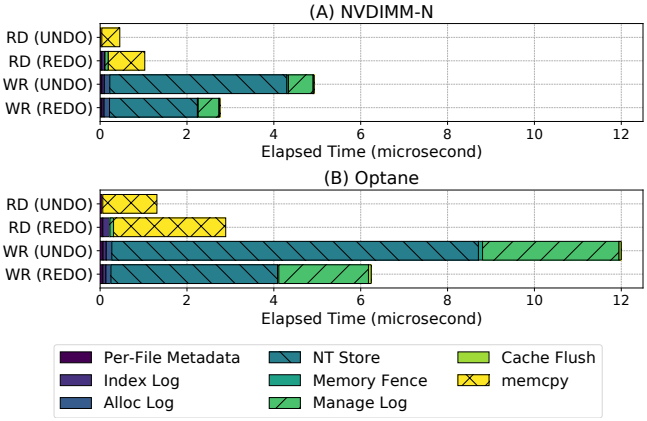

Figure 11: Latency breakdown

only metadata journaling, there is no dramatic performance improvement. NOVA shows the same performance regardless of the fsync interval. Since NOVA performs all the writes atomically and fsync actually does nothing, its performance is not sensitive to the fsync intervals. Libnvmmio implements fsync efficiently with almost little overhead by increasing the current global epoch number at user level. A heavy-lifting work for checkpointing data log is processed in the background. As the fsync interval increases, checkpointing can be done in a batch even in the background. Thus, Libnvmmio can slightly increase the performance on long intervals.

4.2.3 Scalability

Figure 10 shows the performance of multithreaded file IO with FIO random write. In *private file* configuration, each thread writes data to its private file. Whereas, all threads write data to one shared file in *shared file* configuration. In private file configuration on NVDIMM-N, Libnvmmio and NOVA show highly scalable performance. Libnvmmio still shows 29% better performance than NOVA. In contrast, only Libnvmmio sustains scalable performance in shared file configuration on NVDIMM-N. Libnvmmio achieves 13× better performance on 16 threads run than NOVA. It is common for modern applications to access shared files simultaneously from multithreads [40]. While NOVA uses per-inode logging with entire file locks, Libnvmmio uses per-block logging with fine-grained per-block locks. This makes Libnvmmio achieve scalable performance, even when multithreads access the shared file simultaneously. The scalability on Optane is limited mainly due to the memory bandwidth limitation, but Libnvmmio on Optane still shows a little promising results than the others. The other two file systems, Ext4-DAX and PMFS rarely scale on multi-threaded experiments.

4.2.4 Latency

We measured write and read latencies of various NVM-aware file systems and Libnvmmio. To make a fair comparison, all operations are synchronous (`fsync` on every write operation). Table 2 shows the latency of 4KB IO by a single thread. The results were measured on Optane. Libnvmmio outperforms all the other file systems. The advantage of Libnvmmio comes from writing logs in user space and background checkpointing. Ext4-DAX requires copying data between user and kernel buffers, PMFS involves modification of complex data structures, and NOVA requires CoW. Low tail latencies on 99.999th show that Libnvmmio has a high chance to meet the demand for target applications. Since Libnvmmio hooks read/write calls and does not involve any kernel mode switches, Libnvmmio on any file systems can remove the complex techniques the kernel level file systems use. The Libnvmmio latencies on other file systems exhibit almost the same as the ones in Table 2. Our results indicate that applications sensitive to tail latency can adopt Libnvmmio on top of their file systems and drop tail latency dramatically.

Figure 11 shows the latency breakdown of `read` and `write` for two logging policies (undo and redo). As for write, the

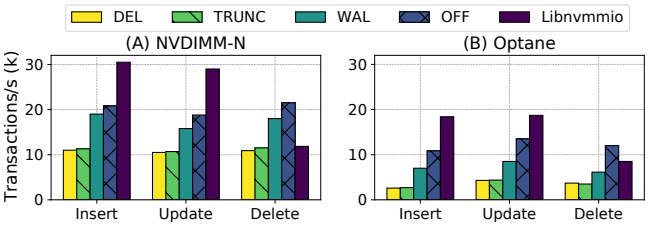

Figure 12: Mobibench on SQLite

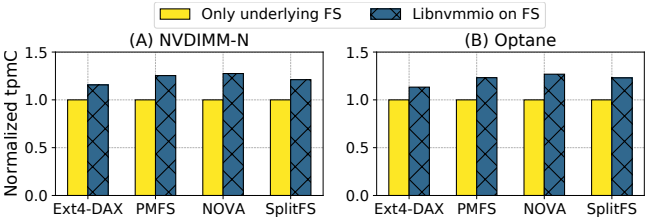

Figure 13: TPC-C on SQLite

portion of non-temporal store (NT Store) is dominating. However, the overheads of the memory fence and cache flush is low due to NT store. In this experiment, we confirmed that it is crucial to select an appropriate logging policy according to access types, as the time spent on memory copy (memcpy, NT Store) varies greatly depending on logging policy. The actual seconds for read and write latencies in Figure 11 are bigger than the latency in Table 2, as time measurement routines for breakdown have been injected.

4.3 Real applications

4.3.1 SQLite

We experimented with SQLite [59] to see how Libnvmmio performs in real applications. To guarantee data-atomicity, SQLite uses its own journaling by default. SQLite calls `fsync` on commit to ensure that all data updated in a transaction is persistent. Libnvmmio keeps updated data in its logs and atomically writes to the original file when `fsync` called. This is how data-atomicity can be guaranteed on SQLite on Libnvmmio without the journaling provided by SQLite. However, the file systems we experimented with cannot turn off the journaling. Even file systems that provide data-atomicity for each operation cannot guarantee the atomicity at transaction level without the journaling.

We used Mobibench [41] to evaluate the basic performance of SQLite. In this experiment, we ran SQLite on NOVA with various journal modes: delete (*DEL*), truncate (*TRUNC*), write-ahead logging (*WAL*), no-journaling (*OFF*). Figure 12 shows that Libnvmmio outperforms all journaling modes on insert and update queries. Even when no journaling is provided from SQLite, Libnvmmio outperforms as all file accesses are handled at user level. Compared to WAL mode on NVDIMM-N, insert and update queries have 60% and 93%

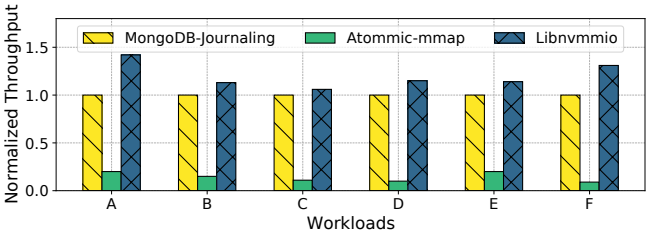

Figure 14: YCSB performance on MongoDB

performance gains in Libnvmmio, respectively. On Optane, the performance gains become 162% and 120%. Mobibench queries request about 100B data IOs. Libnvmmio excels on such small size IOs. On delete transactions, Libnvmmio performs not quite well. According to our call trace, files are truncated frequently on delete workload. When a file is truncated, Libnvmmio needs to adjust the mapping size along with the file size as in FLEX [66] and SplitFS [24]. This incurs relatively high overhead on Libnvmmio. To mend this problem, Libnvmmio needs to optimize file size changes by reflecting file size changes on file close.

We evaluate Libnvmmio on four different file systems by running TPC-C with SQLite. Figure 13 shows that running on Libnvmmio exhibits better performance than running only on underlying file systems. The performance gains range from 16% to 27% on NVDIMM-N and from 13% to 27% on Optane. Since Libnvmmio processes file IO at user level, most of file IO operations can be handled efficiently. As for SplitFS [24], which is built as user-level file system, Libnvmmio uses only mmap interface from SplitFS and performs all other functionalities with its own mechanism. This is why the performance on SplitFS is better for Libnvmmio than only SplitFS. Data updates are kept in its staging files in SplitFS. When applications call `fsync`, SplitFS relinks the updated blocks in staging files into the original file without additional data copying. To make the relink mechanism work, a complete content of the block is required. If applications update only part of a block, SplitFS must copy the rest of the partial data for that block on fsync. The relink mechanism also needs splitting and remapping the existing mapping. Since mapping changes require expensive TLB-shootdown, remapping can cause a higher cost than copying [37]. Additionally, frequent relinks can cause extent fragmentation, as SplitFS uses Ext4-DAX as its underlying file system.

4.3.2 MongoDB MMAPv1

To evaluate Libnvmmio for applications that use memory-mapped IO, we experimented with MongoDB [44] MMAPv1 engine. MongoDB MMAPv1 maps DB files onto its address space, and read/write data with `memcpy`. We have modified 182 lines of source code to make MongoDB MMAPv1 engine use interfaces in Libnvmmio. Figure 14 shows the

performance of YCSB workloads on MongoDB. *MongoDB-Journaling* represents the performance when MongoDB uses its own journaling. In order to ensure that all modifications to a MongoDB data set are durably written to DB files, MongoDB, by default, records all modifications to a journal file. After persisting the data in journal, MongoDB writes the data to a memory-mapped file. Then, it calls `msync` periodically to flush the data in the memory to its file image on the persistent storage. If a system failure occurs during the synchronization, MongoDB can redo the updates by using the journal. *Atomic-mmap* represents the performance when MongoDB uses atomic-mmap provided by NOVA [67]. NOVA maps the replica pages of files onto the user memory, and later when `msync` is called, it copies the replica pages atomically to the original file. In this case, MongoDB can guarantee data-atomicity without using its own journaling. Libnvmmio also ensures the same level of data-atomicity as the atomic-mmap in NOVA. *Libnvmmio* represents the performance when Libnvmmio is used without MongoDB journaling. Compared to MongoDB journaling, Libnvmmio shows 31∼42% performance gains on write intensive workloads (A and F). On read intensive workloads (B, C, D, and E), it shows 6∼15% gains.

Libnvmmio shows the highest performance for all workloads. In YCSB workloads, the default record size is 1KB. Since MongoDB-Journaling uses msync provided by the OS kernel, the synchronization is performed at page granularity. This increases the write amplification but also incurs TLB-shootdown overhead. Whereas, Libnvmmio uses differential logging and user-level msync to minimize write amplification and eliminate unnecessary TLB-shootdown. Atomic-mmap also performs synchronization at page granularity. Besides, as all the replica pages of the file are synchronized regardless of their states (clean or dirty), huge write amplification occurs. Due to such inefficiency, the atomic-mmap feature has been removed from the latest NOVA [68].

5 Related Work

In NVMM systems, file operations travel through memory bus led significantly improved latency. In traditional systems, storage latency was dominant in the total file IO overhead, but in NVMM systems, inefficient behavior of software stacks becomes a dominating overhead. State-of-the-art NVMM-aware file systems bypass the block layer and the page cache layer to avoid the software overhead. Many optimizations take the characteristics of NVMM into account in the file system design. Some suggest to fundamentally change the way file operations work from kernel space to user space.

BPFS and PMFS are early versions of NVMM-aware file systems. BPFS [12] manages the CPU cache based on epoch to provide an accurate ordering and provides atomic data persistence with short-circuit shadow paging. PMFS [13] came up with eXecute In Place (XIP) which nowadays call Direct Access (DAX). PMFS pointed out that NVMM systems should bypass the block layer and page cache to remove unnecessary management schemes from past days.

NOVA [67, 68] suggested more efficient software layer to manage NVMM. NOVA extends the log-structuring technique optimized for block devices to NVMM. NOVA gives each inode a separate log. This technique is suited well in NVMM utilizing fast random access characteristics of NVMM. NOVA provides protection against media errors as well as software errors.

Aerie [62] is a user-level file system that provides flexible file system interfaces. Aerie maximizes the benefits of low-latency NVMM by implementing file system functionality at the user-level. However, Aerie does not guarantee data-atomicity and does not support POSIX semantics.

Strata [28] is a cross-media file system that suggested separation of kernel and user responsibilities. While providing fast performance for read and write, Strata does not support atomic memory-mapped IO. Strata brought data into user space and processes metadata in kernel space.

FLEX [66] replaces read/write system calls with memory-mapped IO to avoid entering the OS kernel. FLEX provides transparent user-level file IO, allowing existing applications to utilize the characteristics of NVMM efficiently. However, FLEX does not guarantee data-atomicity.

SplitFS [24] supports user-level IO while providing flexible crash-consistency guarantees. The relink mechanism proposed by SplitFS allows atomic file updates with minimal data copying. SplitFS handles common data operations at the user level and offloads complex and uncommon metadata operations to kernel file systems. SplitFS proposed the proper role of user libraries and kernel file systems for efficient file IO.

6 Conclusion

Libnvmmio is a simple and practical solution, which provides low-latency and scalable IO while guaranteeing data atomicity. Libnvmmio rebuilds performance-critical software IO path for NVM. It leverages the memory-mapped IO for fast data access and makes applications free from the crash-consistency concerns by providing failure-atomicity. Source code is publicly available at: https://github.com/chjs/libnvmmio.

Acknowledgments

This research was supported in part by Samsung Electronics and the National Research Foundation in Korea under PF Class Heterogeneous High Performance Computer Development NRF-2016M3C4A7952587. We would like to thank our shepherd, Ric Wheeler, and the anonymous reviewers for their insightful comments and suggestions.

References

[1] Saurabh Agarwal, Rahul Garg, Meeta S. Gupta, and Jose E. Moreira. Adaptive incremental checkpointing for massively parallel systems. In *Proceedings of the 18th Annual International Conference on Supercomputing*, ICS '04, pages 277–286, New York, NY, USA, 2004. ACM.

[2] Jaehyung Ahn, Dongup Kwon, Youngsok Kim, Mohammadamin Ajdari, Jaewon Lee, and Jangwoo Kim. DCS: A Fast and Scalable Device-centric Server Architecture. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48. ACM, 2015.

[3] Nadav Amit. Optimizing the tlb shootdown algorithm with page access tracking. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '17, pages 27–39, Berkeley, CA, USA, 2017. USENIX Association.

[4] Dmytro Apalkov, Alexey Khvalkovskiy, Steven Watts, Vladimir Nikitin, Xueti Tang, Daniel Lottis, Kiseok Moon, Xiao Luo, Eugene Chen, Adrian Ong, Alexander Driskill-Smith, and Mohamad Krounbi. Spin-transfer torque magnetic random access memory (stt-mram). *J. Emerg. Technol. Comput. Syst.*, 9(2), May 2013.

[5] Jens Axboe. Flexible I/O Tester. https://github.com/axboe/fio.

[6] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. Efficient virtual memory for big memory servers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 237–248, New York, NY, USA, 2013. ACM.

[7] Adrian M. Caulfield, Todor I. Mollov, Louis Alex Eisner, Arup De, Joel Coburn, and Steven Swanson. 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 XVII. ACM, 2012.

[8] J. Choi, J. Ahn, J. Kim, S. Ryu, and H. Han. In-memory file system with efficient swap support for mobile smart devices. *IEEE Transactions on Consumer Electronics*, 62(3):275–282, 2016.

[9] Jungsik Choi, Jiwon Kim, and Hwansoo Han. Efficient Memory Mapped File I/O for In-Memory File Systems. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*. USENIX Association, 2017.

[10] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. Scalable address spaces using rcu balanced trees. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, page 199–210, New York, NY, USA, 2012. Association for Computing Machinery.

[11] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. Radixvm: Scalable address spaces for multithreaded applications. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, page 211–224, New York, NY, USA, 2013. Association for Computing Machinery.

[12] 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. ACM, 2009.

[13] 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. ACM, 2014.

[14] Izzat El Hajj, Alexander Merritt, Gerd Zellweger, Dejan Milojicic, Reto Achermann, Paolo Faraboschi, Wen-mei Hwu, Timothy Roscoe, and Karsten Schwan. Spacejmp: Programming with multiple virtual address spaces. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 353–368, New York, NY, USA, 2016. ACM.

[15] R. Gioiosa, J. C. Sancho, S. Jiang, and F. Petrini. Transparent, incremental checkpointing at kernel level: a foundation for fault tolerance for parallel computers. In *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, pages 9–9, Nov 2005.

[16] R. Hagmann. Reimplementing the cedar file system using logging and group commit. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, SOSP '87, pages 155–162, New York, NY, USA, 1987. ACM.

[17] Dave Hitz, James Lau, and Michael Malcolm. File system design for an nfs file server appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, WTEC'94, pages 19–19, Berkeley, CA, USA, 1994. USENIX Association.

[18] Intel Memory Latency Checker. https://software.intel.com/en-us/articles/intelr-memory-latency-checker.

[19] Intel Optane™ DC Persistent Memory. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[20] Intel Persistent Memory Programming. https://pmem.io/pmdk/.

[21] Intel and Micron's 3D XPoint™ Technology. https://www.micron.com/about/our-innovation/3d-xpoint-technology.

[22] Jonathan Corbet. Supporting filesystems in persistent memory, 2014. https://lwn.net/Articles/610174/.

[23] Juchang Lee, Kihong Kim, and S. K. Cha. Differential logging: a commutative and associative logging scheme for highly parallel main memory database. In *Proceedings 17th International Conference on Data Engineering*, pages 173–182, April 2001.

[24] 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, pages 494–508, New York, NY, USA, 2019. ACM.

[25] Hyeong-Jun Kim, Young-Sik Lee, and Jin-Soo Kim. 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 '16. USENIX Association, 2016.

[26] Hyunjun Kim, Joonwook Ahn, Sungtae Ryu, Jungsik Choi, and Hwansoo Han. In-memory file system for non-volatile memory. In *Proceedings of the 2013 Research in Adaptive and Convergent Systems*, RACS '13, page 479–484, New York, NY, USA, 2013. Association for Computing Machinery.

[27] Wook-Hee Kim, Jinwoong Kim, Woongki Baek, Beomseok Nam, and Youjip Won. NVWAL: Exploiting NVRAM in Write-Ahead Logging. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16. ACM, 2016.

[28] 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.

[29] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. Coordinated and efficient huge page management with ingens. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 705–721, USA, 2016. USENIX Association.

[30] E. Kültürsay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu. Evaluating stt-ram as an energy-efficient main memory alternative. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '13, April 2013.

[31] Butler W. Lampson. Hints for computer system design. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, SOSP '83, pages 33–48, New York, NY, USA, 1983. ACM.

[32] B. C. Lee, P. Zhou, J. Yang, Y. Zhang, B. Zhao, E. Ipek, O. Mutlu, and D. Burger. Phase-change technology and the future of main memory. *IEEE Micro*, 30(1):143–143, Jan 2010.

[33] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable dram alternative. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09. ACM, 2009.

[34] Edward K. Lee and Chandramohan A. Thekkath. Petal: Distributed virtual disks. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VII, pages 84–92, New York, NY, USA, 1996. ACM.

[35] Gyusun Lee, Wenjing Jin, Wonsuk Song, Jeonghun Gong, Jonghyun Bae, Tae Jun Han, Jae W. Lee, and Jinkyu Jeong. A case for hardware-based demand paging. In *Proceedings of the 47th Annual International Symposium on Computer Architecture*, ISCA '20, pages 1103–1116, New York, NY, USA, 2020. ACM.

[36] Sang-Won Lee and Bongki Moon. Design of flash-based dbms: An in-page logging approach. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, page 55–66, New York, NY, USA, 2007. Association for Computing Machinery.

[37] Bojie Li, Tianyi Cui, Zibo Wang, Wei Bai, and Lintao Zhang. Socksdirect: Datacenter sockets can be fast and compatible. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page

90–103, New York, NY, USA, 2019. Association for Computing Machinery.

[38] Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Khan. Pmtest: A fast and flexible testing framework for persistent memory programs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 411–425, New York, NY, USA, 2019. Association for Computing Machinery.

[39] Amirsaman Memaripour and Steven Swanson. Breeze : User-Level Access to Non-Volatile Main Memories for Legacy Software. In *2018 IEEE 36st International Conference on Computer Design*, ICCD '18. IEEE, 2018.

[40] Changwoo Min, Sanidhya Kashyap, Steffen Maass, and Taesoo Kim. Understanding manycore scalability of file systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 71–85, Denver, CO, June 2016. USENIX Association.

[41] Mobibench. https://github.com/ESOS-Lab/Mobibench.

[42] C. Mohan. Repeating history beyond aries. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, page 1–17, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[43] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, March 1992.

[44] MongoDB. https://www.mongodb.com.

[45] Netlist NVvault DDR4 NVDIMM-N. https://www.netlist.com/products/specialty-dimms/nvvault-ddr4-nvdimm.

[46] Jiaxin Ou, Jiwu Shu, and Youyou Lu. A High Performance File System for Non-volatile Main Memory. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16. ACM, 2016.

[47] Ashish Panwar, Aravinda Prasad, and K. Gopinath. Making huge pages actually useful. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 679–692, New York, NY, USA, 2018. ACM.

[48] Jim Pappas. Annual Update on Interfaces, 2014. https://www.flashmemorysummit.com/English/Collaterals/Proceedings/2014/20140805_U3_Pappas.pdf.

[49] Daejun Park and Dongkun Shin. ijournaling: Fine-grained journaling for improving the latency of fsync system call. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 787–798, Santa Clara, CA, July 2017. USENIX Association.

[50] Stan Park, Terence Kelly, and Kai Shen. Failure-atomic msync(): A simple and efficient mechanism for preserving the integrity of durable data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13. ACM, 2013.

[51] Thanumalayan Sankaranarayana Pillai, Ramnatthan Alagappan, Lanyue Lu, Vijay Chidambaram, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Application Crash Consistency and Performance with CCFS. In *15th USENIX Conference on File and Storage Technologies*, FAST '17. USENIX Association, 2017.

[52] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. All file systems are not created equal: On the complexity of crafting crash-consistent applications. In *11th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '14. USENIX Association, 2014.

[53] Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Analysis and evolution of journaling file systems. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 8–8, Berkeley, CA, USA, 2005. USENIX Association.

[54] S. Qiu and A. L. N. Reddy. Exploiting superpages in a nonvolatile memory file system. In *2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–5, April 2012.

[55] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y. . Chen, R. M. Shelby, M. Salinga, D. Krebs, S. . Chen, H. . Lung, and C. H. Lam. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development*, 52(4.5):465–479, July 2008.

[56] Ohad Rodeh, Josef Bacik, and Chris Mason. Btrfs: The linux b-tree filesystem. *Trans. Storage*, 9(3):9:1–9:32, August 2013.

[57] Livio Soares and Michael Stumm. FlexSC: Flexible System Call Scheduling with Exception-less System Calls. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10. USENIX Association, 2010.

[58] Nae Young Song, Yongseok Son, Hyuck Han, and Heon Young Yeom. Efficient Memory-Mapped I/O on Fast Storage Device. *ACM Transactions on Storage*, 12(4):19:1–19:27, 2016.

[59] SQLite. https://www.sqlite.org.

[60] Michael M. Swift. Towards o(1) memory. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17. ACM, 2017.

[61] C. Villavieja, V. Karakostas, L. Vilanova, Y. Etsion, A. Ramirez, A. Mendelson, N. Navarro, A. Cristal, and O. S. Unsal. Didi: Mitigating the performance impact of tlb shootdowns using a shared tlb directory. In *2011 International Conference on Parallel Architectures and Compilation Techniques*, pages 340–349, Oct 2011.

[62] 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. ACM, 2014.

[63] Yang Wang, Manos Kapritsos, Zuocheng Ren, Prince Mahajan, Jeevitha Kirubanandam, Lorenzo Alvisi, and Mike Dahlin. Robustness in the salus scalable block store. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, page 357–370, USA, 2013. USENIX Association.

[64] David A. Wheeler. SLOCCount. https://dwheeler.com/sloccount/.

[65] Xiaojian Wu and A. L. Narasimha Reddy. SCMFS: A File System for Storage Class Memory. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11. ACM, 2011.

[66] Jian Xu, Juno Kim, Amirsaman Memaripour, and Steven Swanson. Finding and fixing performance pathologies in persistent memory software stacks. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 427–439, New York, NY, USA, 2019. Association for Computing Machinery.

[67] 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. USENIX Association, 2016.

[68] 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. ACM, 2017.

[69] Jisoo Yang, Dave B. Minturn, and Frank Hady. When poll is better than interrupt. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, pages 3–3, Berkeley, CA, USA, 2012. USENIX Association.

[70] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. Nv-tree: Reducing consistency cost for nvm-based single level systems. In *13th USENIX Conference on File and Storage Technologies*, FAST '15. USENIX Association, 2015.

[71] Mai Zheng, Joseph Tucek, Dachuan Huang, Feng Qin, Mark Lillibridge, Elizabeth S. Yang, Bill W Zhao, and Shashank Singh. Torturing databases for fun and profit. In *11th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '14. USENIX Association, 2014.