Finding and Fixing Performance Pathologies in Persistent Memory Software Stacks

Jian Xu*†
Google
andiryxu@google.com

Juno Kim*
University of California San Diego
juno@eng.ucsd.edu

Amirsaman Memaripour
University of California San Diego
amemarip@eng.ucsd.edu

Steven Swanson
University of California San Diego
swanson@eng.ucsd.edu

Abstract

Emerging fast, non-volatile memories will enable systems with large amounts of non-volatile main memory (NVMM) attached to the CPU memory bus, bringing the possibility of dramatic performance gains for IO-intensive applications. This paper analyzes the impact of state-of-the-art NVMM storage systems on some of these applications and explores how those applications can best leverage the performance that NVMMs offer.

Our analysis leads to several conclusions about how systems and applications should adapt to NVMMs. We propose *FiLe Emulation with DAX (FLEX)*, a technique for moving file operations into user space, and show it and other simple changes can dramatically improve application performance. We examine the scalability of NVMM file systems in light of the rising core counts and pronounced NUMA effects in modern systems, and propose changes to Linux's virtual file system (VFS) to improve scalability. We also show that adding NUMA-aware interfaces to an NVMM file system can significantly improve performance.

CCS Concepts • **Information systems** → **Key-value stores**; **Storage class memory**; **Phase change memory**; *Database transaction processing*; • **Software and its engineering** → **File systems management**; *Memory management*;

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

*The first two authors contributed equally to this work.
†Work done at University of California San Diego.

ASPLOS '19, April 13–17, 2019, Providence, RI, USA

ACM ISBN 978-1-4503-6240-5/19/04.
https://doi.org/10.1145/3297858.3304077

ACM Reference Format:
Jian Xu, Juno Kim, Amirsaman Memaripour, and Steven Swanson. 2019. Finding and Fixing Performance Pathologies in Persistent Memory Software Stacks. In *2019 Architectural Support for Programming Languages and Operating Systems (ASPLOS '19), April 13–17, 2019, Providence, RI, USA.* ACM, New York, NY, USA, Article 4, 14 pages. https://doi.org/10.1145/3297858.3304077

1 Introduction

Non-volatile main memory (NVMM) technologies like 3D XPoint [44] promise vast improvements in storage performance, but they also upend conventional design principles for the storage stack and the applications that use them. Software designed with conventional design principles in mind is likely to be a poor fit for NVMM due to its extremely low latency (compared to block devices) and its ability to support an enormous number of fine-grained, parallel accesses.

The process of adapting existing storage systems to NVMMs is in its early days, but important progress has been made: Researchers, companies, and open-source communities have built *native NVMM file systems* specifically for NVMMs[17, 21, 37, 63, 67, 68], both Linux and Windows have created *adapted NVMM file systems* by adding support for NVMM to existing file systems (e.g., ext4-DAX, xfs-DAX and NTFS), and some commercial applications have begun to leverage NVMMs to improve performance [2].

System support for NVMM brings a host of potential benefits. The most obvious of these is faster file access via conventional file system interfaces (e.g., `open`, `read`, `write`, and `fsync`). These interfaces should make leveraging NVMM performance easy, and several papers [19, 37, 67, 68] have shown significant performance gains without changing applications, demonstrating the benefits of specialized NVMM file systems.

A second, oft-cited benefit of NVMM is *direct access (DAX)* mmap, which allows an application to map the pages of an NVMM-backed file into its address space and then access it via load and store instructions. DAX removes all of the system software overhead

for common-case accesses enabling the fastest-possible access to persistent data. Using DAX requires applications to adopt an mmap-based interface to storage, and recent research shows that performance gains can be significant [15, 43, 52, 64].

Despite this early progress, several important questions remain about how applications can best exploit NVMMs and how file systems can best support those applications. These questions include:

1. How much effort is required to adapt legacy applications to exploit NVMMs? What best practices should developers follow?
2. Are sophisticated NVMM-based data structures necessary to exploit NVMM performance?
3. How effectively can legacy files systems evolve to accommodate NVMMs? What trade-offs are involved?
4. How effectively can current NVMM file systems scale to many-core, multi-socket systems? How can we improve scalability?

This paper offers insight into all of these questions by analyzing the performance and behavior of benchmark suites and full-scale applications on multiple NVMM-aware file systems on a many-core machine. We identify bottlenecks caused by application design, file system algorithms, generic kernel interfaces, and basic limitations of NVMM performance. In each case, we either apply well-known techniques or propose solutions that aim to boost performance while minimizing the burden on the application programmer, thereby easing the transition to NVMM.

Our results offer a broad view of the current landscape of NVMM-optimized system software. Our findings include the following:

- For the applications we examined, FiLe Emulation with DAX (FLEX) provides almost as much benefit as building complex crash-consistent data structures in NVMM.
- Block-based journaling mechanisms are a bottleneck for adapted NVMM file systems. Adding DAX-aware journaling improves performance on many operations.
- The block-based compatibility requirements of adapted NVMM file systems limit their performance on NVMM in some cases, suggesting that native NVMM file systems are likely to maintain a performance advantage.
- Poor performance in accessing non-local memory (NUMA effects) can significantly impact NVMM file system performance. Adding NUMA-aware interfaces to NVMM file systems can relieve these problems.

The remainder of the paper is organized as follows. Section 2 describes NVMMs, NVMM file system design issues, and the challenges that NVMM storage stacks face. Section 3 evaluates applications on NVMM and recounts the lessons we learned in porting them to NVMMs. Section 4 describes the scalability bottlenecks of NVMM file systems and how to fix them, and Section 5 concludes.

2 Background

This section provides a brief survey of NVMM technologies, the current state of NVMM-aware file systems, and the challenges of accessing NVMM directly with DAX.

2.1 Non-volatile memory technologies

This paper focuses on storage systems built around non-volatile memories attached to the processor memory bus that appear as a directly-addressable, persistent region in the processor's address space. We assume the memories offer performance similar to (but perhaps slightly lower than) DRAM.

Modern server platforms have supported NVMM in the form of battery-backed NVDIMMs [27, 45] for several years, and Linux and Windows include facilities to access these memories and build file systems on top of them.

Denser NVDIMMs that do not need a battery have been announced by Intel and Micron and use a technology termed "3D XPoint" [44]. There are several potential competitors to 3D XPoint, such as spin-torque transfer RAM (STT-RAM) [34, 48], phase change memory (PCM) [8, 13, 38, 53], and resistive RAM (ReRAM) [23, 58]. Each has different strengths and weaknesses: STT-RAM can meet or surpass DRAM's latency and it may eventually appear in on-chip, last-level caches [71], but its large cell size limits capacity and its feasibility as a DRAM replacement. PCM, ReRAM, and 3D XPoint are denser than DRAM, and may enable very large, non-volatile main memories. Their latency will be worse than DRAM, however, especially for writes. All of these memories suffer from potential wear-out after repeated writes.

2.2 NVMM File Systems and DAX

NVMMs' low latency makes software efficiency much more important than in block-based storage systems [4, 9, 65, 69]. This difference has driven the development of several NVMM-aware file systems [17, 19, 21, 37, 63, 66–68].

NVMM-aware file systems share two key characteristics: First, they implement direct access (DAX) features. DAX lets the file system avoid using the operating system buffer cache: There is no need to copy data from "disk" into memory since file data is always in memory (i.e., in NVMM). As a side effect, the `mmap()` system call

maps the pages that make up a file directly into the application's address space, allowing direct access via loads and stores. We refer to this capability as *DAX-mmap*. One crucial advantage of DAX-mmap is that it allows `msync()` to be implemented in user space by flushing the affected cachelines and issuing a memory barrier. In addition, the `fdatasync()` system call becomes a noop.

One small caveat is that the call to `mmap` must include the recently-added MAP_SYNC flag that ensures that the file is fully allocated and its metadata has been flushed to media. This is necessary because, without MAP_SYNC, in the disk-optimized implementations of `msync` and `mmap` that ext4 and xfs provide, `msync` can sometimes require metadata updates (e.g., to lazily allocate a page).

The second characteristic is that they make different assumptions about the atomicity of updates to storage. Current processors provide 8-byte atomicity for stores to NVMM instead of the sector atomicity that block devices provide.

We divide NVMM file systems into two groups. *Native* NVMM filesystems (or just "native file system") are designed especially for NVMMs. They exploit the byte-addressability of NVMM storage and can dispense with many of the optimizations (and associated complexity) that block-based file systems implement to hide the poor performance of disks.

The first native file system we are aware of is BPFS [17], a copy-on-write file system that introduced short-circuit shadow paging and proposed processor architecture extensions to make NVMM programming more efficient. Intel's PMFS [21], the first NVMM file system released publicly, has scalability issues with large directories and metadata operations.

NOVA [67, 68] is a log-structured file system designed for NVMM. It gives each inode a separate log to ensure scalability, and combines logging, light-weight journaling and copy-on-write to provide strong atomicity guarantees to both metadata and data. NOVA also includes snapshot and fault-tolerance features. NOVA is the only native DAX file system that is publicly available and supported by recent kernels (Intel has deprecated PMFS). It outperforms PMFS on all the workloads for which we have compared them.

Strata [37] is a "cross media" file system that runs partly in userspace. It provides strong atomicity and high performance, but does not support DAX[1].

[1]We have been working to include a quantitative comparison to Strata in this study, but we have run into several bugs and limitations. For example, it has trouble with multi-threaded workloads [36] and many of the workloads we use do not run successfully. Until we can resolve these issues, we have included qualitative discussion of Strata where appropriate.

Adapted NVMM file systems (or just "adapted file systems") are block-based file systems extended to implement NVMM features, like DAX and DAX-mmap. Xfs-DAX [12], ext4-DAX [66] and NTFS [25] all have modes in which they become adapted file systems. Xfs-DAX and ext4-DAX are the state-of-the-art adapted NVMM file systems in the Linux kernel. They add DAX support to the original file systems so that data page accesses bypass the page cache, but metadata updates still go through the old block-based journaling mechanism [10, 11].

So far, adapted file systems have been built subject to constraints that limit how much they can change to support NVMM. For instance, they use the same on-"disk" format in both block-based and DAX modes, and they must continue to implement (or at least remain compatible with) disk-centric optimizations.

Adapted file systems also often give up some features in DAX mode. For instance, ext4 does not support data journaling in DAX mode, so it cannot provide strong consistency guarantees on file data. Xfs disables many of its data integrity features in DAX mode.

2.3 NVMM programming

DAX-mmap gives applications the fastest possible access to stored data and allows them to build complex, persistent, pointer-based data structures. This usage model has the application create a large file in a NVMM file system, use `mmap()` to map it into its address space, and then rely on a userspace persistent object library [15, 52, 64] to manage it.

These libraries generally provide persistent memory allocators, an object model, and support for transactions on persistent objects. To ensure persistence and consistency, these libraries use instructions such as `clflushopt` and `clwb` to flush the dirty cachelines [28, 72] to NVMM, and non-temporal store instructions like `movntq` to bypass the CPU caches and write directly to NVMM. Enforcing ordering between stores requires memory barriers such as `mfence`.

Using mapped NVMM to build complex data structures is a daunting challenge. Programmers must manage concurrency, consistency, and memory allocation all while ensuring that the program can recover from an ill-timed system crash. Even worse, data structure corruption and memory leaks are persistent, so rebooting, the always-reliable solution to volatile data structure corruption and DRAM leaks, will not help. Addressing these challenges is the subject of a growing body of research [3, 15, 43, 52, 62, 64, 70].

3 Adapting applications to NVMM

The first applications to use NVMM in production are likely to be legacy applications originally built for block-based storage. Blithely running that code on a new, faster storage system will yield some gains, but fully exploiting NVMM's potential will require some tuning and modification. The amount, kind, and complexity of tuning required will help determine how quickly and how effectively applications can adapt.

We gathered the first-hand experience with porting legacy applications to NVMM-based storage systems by modifying five lightweight databases and key-value stores to better utilize NVMMs. The techniques we applied for each application depend on how it accesses the underlying storage. Below, we detail our experience and identify some best practices for NVMM programmers. Then, based on these findings, we propose a DAX-aware journaling scheme for ext4 that eliminates block IO overheads.

We use a quad-socket prototype HPE Scalable Persistent Memory server [26] to evaluate these applications. The server combines DRAM with NVMe SSDs and an integrated battery backup unit to create NVMM. The server hosts four Xeon Gold 6148 processors (a total of 80 cores), 300 GB of DRAM, and 300 GB of NVMM. We evaluate all the applications on Linux kernel 4.13.

3.1 SQLite

SQLite [57] is a lightweight embedded relational database that is popular in mobile systems. SQLite stores data in a B+tree contained in a single file.

To ensure consistency, SQLite uses one of four different techniques to log operations to a separate log file. Three of the techniques, DELETE, TRUNCATE and PERSIST, store undo logs while the last, WAL stores redo logs.

The undo logging modes invalidate the log after every operation. DELETE and TRUNCATE, respectively, delete the log file or truncate it. PERSISTS issues a write to set an "invalid" flag in log file header.

WAL appends redo log entries to a log file and takes periodic checkpoints after which it deletes the log and starts again.

We use Mobibench [29] to test the SET performance of SQLite in each journaling mode. The workload inserts 100 byte long values into a table. Figure 1 shows the result. DELETE and TRUNCATE incur significant file system journaling overhead with ext4-DAX and xfs-DAX. NOVA performs better because it does not need to journal operations that affect a single file. PERSIST mode performs equally on all three file systems.

WAL avoids file creation and deletion, but it does require allocating new space for each log entry. Ext4-DAX and xfs-DAX keep their allocator state in NVMM and keep it consistent at all times, so the allocation is expensive. Persistent allocator state is necessary in block-based file systems to avoid a time-consuming (on disk) media scan after a crash.

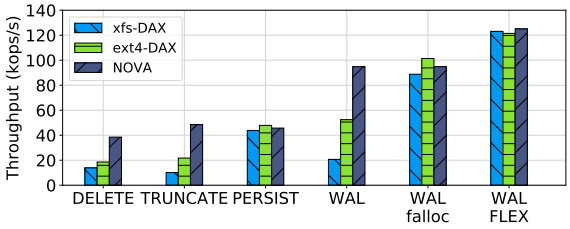

Figure 1. SQLite SET throughput with different journaling modes. Preallocating space for the log file using `falloc` avoids allocation overheads and makes write ahead logging (WAL) the clearly superior logging mode for SQLite running on NVMM file systems.

Scanning NVMM after crash is much less costly, so NOVA keeps allocator state in DRAM and only writes it to NVMM on a clean unmount. As a result, the allocation is much faster.

This difference in allocation overhead limits WAL's performance advantage compared to PERSIST to 9% for ext4-DAX, reduces performance by 53% for xfs-DAX, but improves NOVA's performance by 107%.

We modified SQLite to avoid allocation overhead by using `fallocate` to pre-allocate the WAL file. This is a common optimization for disk-based file systems, and it works here as well: The change closes the gap between the three file systems.

To improve performance further, we use a technique we call FiLe Emulation with DAX (FLEX) to avoid the kernel completely for writes to the WAL file. To implement FLEX, SQLite DAX-mmaps the WAL file into its address space and uses non-temporal stores and `clwb` to ensure the log entries are reliably stored in NVMM. We study FLEX in detail in Section 3.4. Implementing these changes required changing just 266 lines of code but improved performance by between 15% and 38%, and further narrows the performance gap between the three file systems.

This final DAX-aware version of SQLite outperforms the PERSIST version by between 2.5× and 2.8×.

Other groups have adapted SQLite to solid-states storage as well. Jeong *et al.* [30] and WALDIO [39] investigate SQLite I/O access patterns and implement optimizations in ext4's journaling system or SQLite itself to reduce the cost of write-ahead logging. Our approach is similar, but it leverages DAX to avoid the file system and leverage NVMM. SQLite/PPL [49], NVWAL [35] use slotted paging [55] to make SQLite run efficiently on NVMM. A comparison to these systems would be

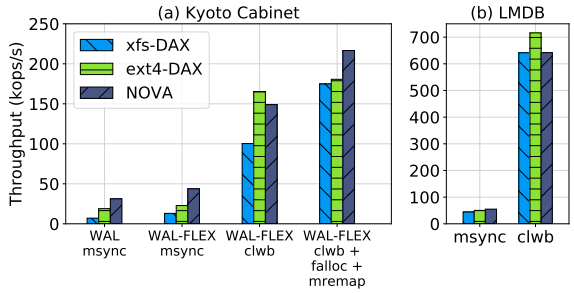

Figure 2. Kyoto Cabinet (KC) and LMDB SET throughput. Applications that use `mmap` can improve performance by performing `msync` in userspace.

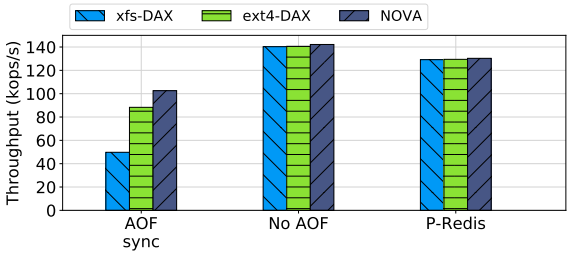

Figure 3. Redis MSET throughput. Making Redis' core data structure persistent in NVMM (P-Redis) improves performance by 27% to 2.6×.

interesting, but unfortunately, none of them is publicly available.

3.2 Kyoto Cabinet and LMDB

Even without DAX, some applications access files via `mmap`, and this makes them a natural match for DAX file systems. However, maximizing the benefits of DAX still requires some changes. We select two applications to explore what is required: Kyoto Cabinet and LMDB.

Kyoto Cabinet Kyoto Cabinet [24] (KC) is a high performance database library. It stores the database in a single file with database metadata at the head. Kyoto Cabinet memory maps the metadata region, uses load/store instructions to access and update it, and calls `msync` to persist the changes. Kyoto Cabinet uses write-ahead logging to provide failure atomicity for SET operations.

Figure 2 shows the impact of applying optimizations to KC's database file and its write-ahead log. First, we change KC to use FLEX writes to update the log ("WAL-FLEX msync" in the figure). The left two sets of bars in Figure 2 (a) show the impact of these changes. The graph plots throughput for SET operation on Kyoto Cabinet HashDB. The key size is 8 bytes and value size is 1024 bytes. FLEX write improves performance by 40% for NOVA, 20% for ext4-DAX, and 84% for xfs-DAX.

Kyoto Cabinet calls `msync` frequently on its data file to ensure that updates to memory-mapped data are persistent. DAX-mmap allows userspace to provide these guarantees using a series of `clwb` instructions followed by a memory fence. Flushing in userspace is also more precise since `msync` operates on pages rather than cache lines. Avoiding `msync` improves performance further by 3.4× for NOVA, 7.2× for ext4-DAX, and 7.7× for xfs-DAX ("WAL-opt clwb").

By default, Kyoto Cabinet only mmaps the first 64 MB of the file, which includes the header and ∼63 MB of data. It uses `write` to append new records to the file. Our final optimization uses `fallocate` and `mremap` to resize the file ("WAL-opt clb + falloc + mremap"). It boosts the throughput for all the file systems by between 7× to 25×, compared to the baseline implementation that issued `msync` system calls without WAL optimization.

Implementing all of these optimizations for both files required changing just 181 lines of code.

LMDB Lightning Memory-Mapped Database Manager (LMDB) [59] is a Btree-based lightweight database management library. LMDB memory-maps the entire database, so that all data accesses directly load and store the mapped memory region. LMDB performs copy-on-write on data pages to provide atomicity, a technique that requires frequent `msync` calls.

For LMDB, using `clwb` instead of `msync` improves the throughput by between 11× to 14× (Figure 2 (b)). Ext4-DAX out-performs xfs-DAX and NOVA by about 11% because ext4-DAX supports super-page (2 MB) `mmap` which reduces the number of page faults. These changes entailed changes to 101 lines of code.

3.3 RocksDB and Redis

Since disk is slow, many disk-based applications keep data structures in DRAM and flush them to disk only when necessary. To provide persistence, they also record updates in a persistent log, since sequential access is most efficient for disks. We consider two such applications, Redis and RocksDB, to understand how this technique can be adapted to NVMM.

Redis Redis [54] is an in-memory key-value store widely used in web site development as a caching layer and for message queue applications. Redis uses an "append only file" (AOF) to log all the write operations to the storage device. At recovery, it replays the log. The frequency at which Redis flushes the AOF to persistent storage allows the administrator to trade-off between performance and consistency.

Figure 3 measures Redis MSET (multiple set) performance. As we have seen with other applications, xfs-DAX's journaling overheads hurt append performance. The graph also shows the potential benefit of eliminating

AOF (and giving up persistence): It improves throughput by 2.8×, 59%, and 38% for xfs-DAX, ext4-DAX, and NOVA, respectively.

The hash table Redis uses internally is an attractive target for NVMM conversion, since making it persistent would eliminate the need for the AOF. We created a fully-functional persistent version of the hash table in NVMM using PMDK [52] by adopting a copy-on-write mechanism for atomic updates: To insert/update a key-value pair, we allocate a new pair to store the data, and replace the old data by atomically updating the pointer in the hashtable. We refer to the resulting system as Persistent Redis (P-Redis).

The throughput with our persistent hash table is 27% to 2.6× better than using synchronous writes to the AOF, and ∼9% worse than skipping persistence altogether. Implementing the persistent version of the hash table took 1529 lines of code.

Although Redis is highly-optimized for DRAM, porting it to NVMM is not straightforward and requires large engineering effort. First, Redis represents and stores different objects with different encodings and formats, and P-Redis has to be able to interpret and handle the various types of objects properly. Second, Redis stores virtual addresses in the hashtable, and P-Redis needs to either adjust the addresses upon restart if the virtual address of the `mmap`'d hashtable file has changed, or change the internal hashtable implementation to use offset instead of absolute addresses [16]. Neither option is satisfying, and we choose the former solution for simplicity. Third, whenever Redis starts, it uses a random seed for its hashing functions, and P-Redis must make the seeds constant. Fourth, Redis updates the hashtable entry before updating the value, and P-Redis must persist the key-value pair before updating the hashtable entry for consistency. Finally, P-Redis hashtable does not support resizing as it requires journaling mechanism to guarantee consistency.

RocksDB RocksDB [22] is a high-performance embedded key-value store based on log-structured merge trees (LSM-trees). When applications write data to a LSM-tree, RocksDB inserts the data to a skip-list in DRAM, and appends the data to a write-ahead log (WAL) file. When the skip-list is full, RocksDB writes it to disk and discards the log file.

Figure 4 measures RocksDB SET throughput with 20-byte keys and 100-byte values. RocksDB's default settings perform poorly on xfs-DAX and ext4-DAX, because each append requires journaling for those file systems. NOVA performs better because it avoids this cost.

RocksDB benefits from FLEX as well. It improves throughput by 2.2× - 18.7× and eliminates the performance gap between file systems.

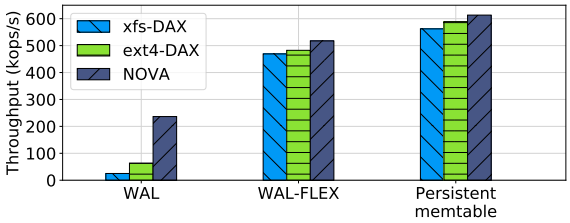

Figure 4. RocksDB SET throughput. Appends to the write-ahead log (WAL) file limit RocksDB throughput on NVMM file systems. Using FLEX writes improves performance by 2.2× to 18.7×. Replacing the skip-list and the log with a crash-consistent, persistent skip-list improves throughput by another 19% on average.

Since the skip-list contains the same information as the WAL file, we eliminate the WAL file by making the skip-list a persistent data structure, similar to NoveLSM [32] based on LevelDB. The final bars in Figure 4 measure the performance of RocksDB with a crash-consistent skip-list in NVMM. Performance improves by 11× compared to the RocksDB baseline but just 19% compared to optimizing WAL with FLEX.

3.4 Evaluating FLEX

In general, FLEX involves replacing conventional file operations with similar DAX-based operations to avoid entering the kernel. We have applied FLEX techniques by hand to the SQLite, RocksDB, and Kyoto Cabinet, but they could easily be encapsulated in a simple library.

FLEX replaces `open()` with `open()` followed by DAX-`mmap()` to map the file into the application's address space. Then, the application can replace `read()` and `write()` system calls with userspace operations.

A FLEX write first checks if the file will grow as a result of the write. If so, the application can expand the file using `fallocate()` and `mmap()` or `mremap()` to expand the mapping. To amortize the cost of `fallocate()`, the application can extend the file by more than the write requires.

Once space is available, a FLEX write uses non-temporal stores to copy data into the file. If the write needs to be synchronous the application issues an `sfence` instruction to ensure the stores have completed. FLEX also uses an `sfence` instruction to replace `fsync()`.

FLEX reads are simpler: They simply translate to `memcpy()`.

FLEX requires the application to track a small amount of extra state about the file, including its location in memory, its current write point, and its current allocated size.

FLEX operations provide semantics that are similar to POSIX, but there are important and potentially subtle differences. First, operations are not atomic. Second, POSIX semantics for shared file descriptors are lost. We

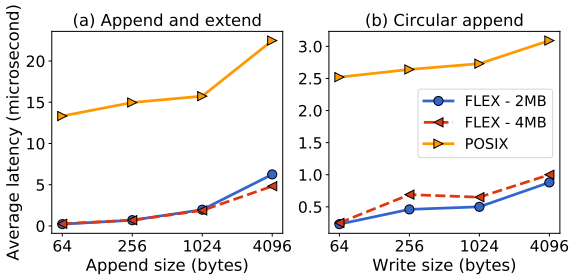

Figure 5. The Impact of FLEX File Operations. Emulating file accesses in user space can improve performance for a wide range of access patterns. Note that the Y axes have different scales. "-2 MB" and "-4 MB" denote different `fallocate()` sizes.

have not found these differences to be relevant for the performance-critical file operations in the workloads we have studied. We elaborate this point in Section 3.4.1.

To understand when FLEX improves performance, we constructed a simple microbenchmark that opens a file, and performs a series of reads or writes each followed by `fsync()`. We vary the size and number of operations and the amount of file space we pre-allocate with `fallocate()`. Figure 5 shows results for two different cases: "Append and extend" uses FLEX to emulate append operations that always cause the file to grow. "Circular append" reuses the same file area and avoids the need to allocate more space. The applications we studied use both models to implement logging: RocksDB uses "append and extend" whereas SQLite and Kyoto Cabinet use "circular append."

The data show that FLEX outperforms normal write operations by up to 61× for append and extend and up to 11× for circular append. The larger speedup for append and extend is due to the NVMM allocation overhead. Performance gains are especially large for small writes, a common case in the applications we studied.

For use cases that must extend the file, minimizing the cost of space allocation is critical. The results in the figure use 2 MB pages to minimize paging overheads. With 4 KB pages, FLEX only provides speedups for transfers under 4 KB.

Our experience with applying FLEX to RocksDB, SQLite, and Kyoto Cabinet shows that it can provide substantial performance benefits for very little effort. In contrast to re-implementing data structures to be crash-consistent, FLEX requires little to no changes to application logic and requires no additional logging or locking protocols. The only subtleties lie in determining that strict POSIX semantics are not necessary.

These results show that FLEX can provide an easy, incremental, and high-value path for developers creating new applications for NVMM or migrating existing code. It also reduces the importance of using a native NVMM file system, further easing migration, since FLEX performance depends little on the underlying file system.

The Strata file system [37] provides some of the same advantages as FLEX through userspace logging through a library that communicates with the in-kernel file system. Their results show that coupling the user space interface to the underlying file system leads to good performance. Their interface makes strong atomicity guarantees while FLEX lets the application enforce the semantics it requires.

3.4.1 Correctness

Since FLEX is not atomic, applying it to applications that assume atomic writes is likely to cause a correctness problem. To our knowledge, SQLite, RocksDB, and Kyoto Cabinet do not assume the atomicity of write system calls [51], thereby applying FLEX does not break their application logic. Only LMDB assumes that 512 bytes sector writes are atomic [1]. Therefore, running it on NVMM file systems introduces the correctness problem since only 8 bytes are atomic on NVMM. To solve this problem, we added a checksum for the LMDB metadata: When a checksum error is detected, LMDB falls back to the previous header.

3.5 Best Practices

Based on our experiences with these five applications, we can draw some useful conclusions about how applications can profitably exploit NVMMs.

Use FLEX Emulating file operations in user space provides large performance gains for very little programmer effort.

Use fine-grained cache flushing instead of `msync` Applications that already use `mmap` and `msync` to access data and ensure consistency, can improve performance significantly by flushing cache lines rather than `msync`'ing pages. However, ensuring that all updated cache lines are flushed correctly can be a challenge.

Use complex persistent data structure judiciously For both of the DRAM data structures we made persistent, the programming effort required was significant and likely performance gains were relatively small relative to FLEX. This finding leads us to two conclusions: First, it is critical to make building persistent data structures in NVMM as easy as possible. Second, it is wise to estimate the potential performance impact the persistent data structure will have before investing a large amount of programmer effort in developing it [41].

Preallocate files on adapted NVMM file systems Several of the performance problems we found with

	Native techniques		Optimizations (Lines changed)		
	WAL	mmap+msync	FLEX	CLWB+fence	Persistent Objects
SQLite	×	-	266	-	-
Kyoto Cabinet	×	×	133	48	-
LMDB	-	×	-	101	-
Redis	×	-	-	-	1326
RocksDB	×	-	56	-	380

Table 1. Application Optimization Summary The applications we studied used a variety of techniques to reliably store persistent state. All the optimizations we applied improved performance, but the amount of programmer effort varied widely. The data Figures 1, 2, 3, and 4 show that programmer effort does not correlate with performance gains.

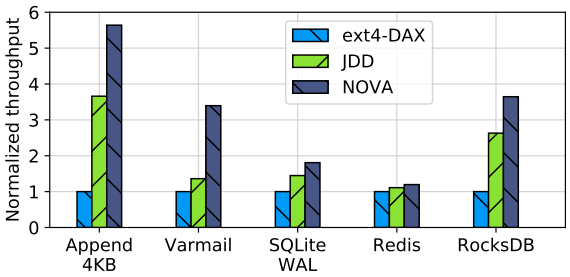

Figure 6. JDD performance. Fine-grained, DAX-optimized journaling on NVMM improves performance for metadata-intensive applications.

adapted NVMM file systems stemmed from storage allocation overheads. Using `fallocate` to pre-allocate file space eliminated them.

Avoid meta-data operations Directory operations (e.g., deleting files) and storage allocation incurred journaling overheads in both xfs and ext4. Avoiding them improves performance, but this is not always possible.

3.6 Reducing journaling overhead

Several of the best practices we identify above focus on avoiding metadata operations since they are often slow. This can be awkward and some metadata operations are unavoidable, so improving their performance would make adapting to NVMMs easier and improve performance.

NOVA's mechanism for performing consistent metadata updates is tailored specifically for NVMMs, but ext4 and xfs' journaling mechanisms were built for disk, and this legacy is evident in their poorer metadata performance.

Ext4 uses the journaling block device (JBD2) to perform consistent metadata updates. To ensure atomicity, it always writes entire 4 KB pages, even if the metadata change affects a single byte. Transactions often involve multiple metadata pages. For instance, appending 4 KB data to a file and then calling `fsync` writes one data page and eight journal pages: a header, a commit block, and up to six pages for inode, inode bitmap, and allocator.

JDB2 also allows no concurrency between journaled operations, so concurrent threads must synchronize to join the same running transaction, making the journaling a scalability bottleneck [56]. Son *et al.* [56] and iJournaling [50] have tried to fix ext4's scalability issues by reducing lock contention and adding per-core journal areas to JBD2.

Previous works [10, 11] has identified the inefficiencies of coarse-grain logging and proposed solutions in the context of block-based file systems. FSMAC [11] maintains data in disk/SSD and metadata in NVMM, and uses undo log journaling for metadata consistency. The work in [10] journals redo log records of individual metadata fields to NVMM during transaction commit, and applies them to storage during checkpointing.

To understand how much of ext4's poor metadata performance is due to coarse-grain logging, we apply these fine-grain logging techniques to develop a journaling DAX device (JDD) for ext4 which performs DAX-style journaling on NVMM and provides improved scalability.

JDD makes three key improvements to JBD2. First, it journals individual metadata fields rather than entire pages. Second, it provides pre-allocated, per-CPU journaling areas so CPUs can perform journaled operations in parallel. Third, it uses undo logging in the journals: It copies the old values into the journal and performs updates directly to the metadata structures in NVMM. To commit an update it marks the journal as invalid. During recovery from a crash, the file system rolls back partial updates using the journaled data. These changes provide for very lightweight transaction commit and make checkpointing unnecessary.

JDD differs from the previous works by focusing on NVMM file systems. FSMAC aims to accelerate metadata updates for disk-based file systems by putting the metadata separately in NVMM. To handle the performance gap between NVMM and disk, FSMAC maintains

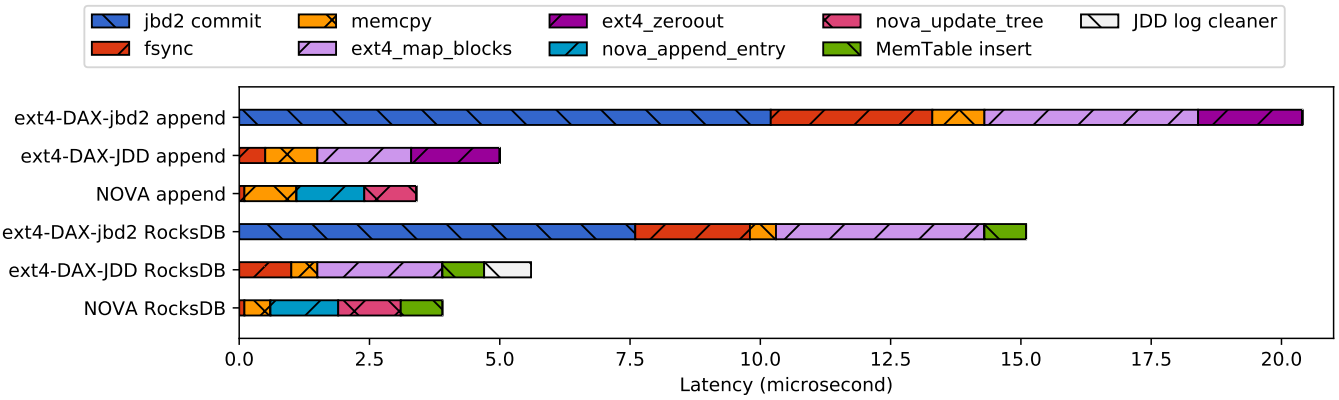

Figure 7. Latency break for 4KB append and RocksDB SET. JDD significantly reduces journaling overhead by eliminating JBD2 transaction commit, but still has higher latency than NOVA's metadata update mechanism.

multiple versions of metadata. The work in [10] optimizes ext4 using fine-grained redo logging on NVMM journal. We built JDD to improve the performance of adapted NVMM file systems using fine-grained undo logging, avoiding the complexity of previous works – managing versions in FSMAC or transaction committing and checkpointing in [10].

Strata [37] and Aerie [63] take a more aggressive approach and log updates in userspace under the control of file system-specific libraries. Metadata updates occur later and off the critical path. This approach should offer better performance than the techniques described above since it avoids entering the kernel for metadata updates. However, it also involves more extensive changes to the application.

Figure 6 shows JDD's impact on a microbenchmark that performs random 4 KB writes followed by `fsync`, Filebench [60] Varmail (which is metadata-intensive), and the three databases and key value stores we evaluated earlier that perform frequent metadata operations as part of WAL. The JDD improves the microbenchmark performance by 3.7× and varmail by 40%. For applications that use write-ahead logging, the benefits range from 11% to 2.6×.

We further analyze the latency of JDD for 4 KB appends and RocksDB SET operation and show the latency breakdown in Figure 7. In ext4-DAX, JBD2 transaction commit (jbd2_commit) occupies 50% of the total latency. JDD eliminates this overhead by performing undo logging. JDD also reduces ext4 overheads such as block allocation (ext4_map_blocks). The remaining performance gap between ext4 and NOVA (46%) is due to ext4's more complex design and its need to keep more persistent states in storage media. In particular (as discussed in Section 3.1) ext4 keeps its data block and inode allocator state continually up-to-date on disk.

The performance improvement on Redis and SQLite are smaller, because they have higher internal overheads. Redis spends most of its time on TCP transfers between the Redis server and the benchmark application, and SQLite spends over 40% of execution time parsing SQL and performing B-tree operations.

4 File System Scalability

We expect NVMM file systems to be subject to more onerous scalability demands than block-based filesystems due to the higher performance of the underlying media and the large amount of parallelism that modern memory hierarchies can support [5]. Further, since NVMMs attach to the CPU memory bus, the capacity of NVMM file systems will tend to scale with the number sockets (and cores) in the systems.

Many-core scalability is also a concern for conventional block-based file systems, and researchers have proposed potential solutions. SpanFS [31] shards file and directories across cores at a coarse granularity, requiring developers to distribute the files and directories carefully. ScaleFS [5] decouples the in-memory file system from the on-disk file system, and uses per-core operation logs to achieve high concurrency. ScaleFS was built on xv6, a research prototype kernel, which makes impossible to perform a good head-to-head comparison with our changes. However, we expect that applying its techniques and the Scalable Commutativity Rule [14] systematically to NVMM file systems (and the VFS layer) might yield further scaling improvements.

This section first describes the FxMark [46] benchmark suite. Then, we identify several operations that have scalability limitations and propose solutions.

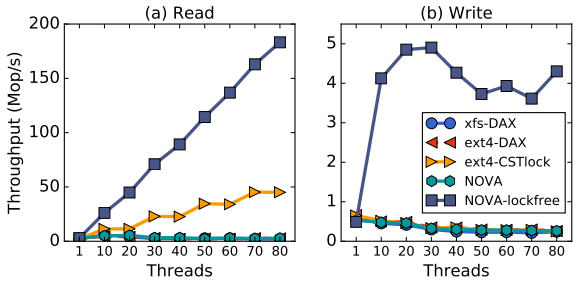

Figure 8. Concurrent 4KB read and write throughput. By default, Linux uses a non-scalable reader/writer lock to coordinate access to files. Using finer-grain, more scalable locks improves read and write scalability.

4.1 FxMark scalability test suite

Min *et al.* [46] built a file system scalability test suite called FxMark and used it to identify many scalability problems in both file systems and Linux's VFS layer. It includes nineteen tests of performance for data and metadata operations under varying levels of contention.

Min *et al.* use FxMark to identify scalability bottlenecks across many file systems. Interestingly, it is their analysis of tmpfs, a DRAM-based pseudo-file system that reveals the bottlenecks that are most critical for ext4-DAX, xfs-DAX, and/or NOVA.

We repeat their experiments and then develop solutions to improve scalability. The solutions we identify are sufficient to give good scalability with NVMM, but would probably also help disk-based file systems too.

FxMark includes nineteen workloads. Below, we only discuss those that show poor scalability for at least one the NVMM file systems we consider.

4.2 Concurrent file read/write

Concurrent `read` and `write` operations to a shared file are a well-known sore spot in file system performance. Figure 8 shows scalability problems for both reads and writes across ext4-DAX, xfs-DAX, and NOVA. The root cause of this poor performance is Linux's read/write semaphore implementation [6, 7, 33, 40]: It is not scalable because of the atomic update required to acquire and release it.

The semaphore protects two things: The file data and the metadata that describes the file layout. To remove this bottleneck in NOVA, we use separate mechanisms to protect the data and metadata.

To protect file data, we leverage NOVA's logs. NOVA maintains one log per inode. Many of the log entries correspond to write operations and hold pointers to the file pages that contain the data for the write. Rather than locking the whole inode, we use reader/writer locks on each log entry to protect the pages to which it links. Although this lock resides in NVMM, its state is not necessary for recovery and is cleared before use after a restart, so hot locks will reside in processor caches and not usually be subject to NVMM access latency.

NOVA's approach to tracking file layout makes protecting it simple. NOVA uses an in-DRAM radix tree to map file offsets to write entries in the log. Write operations update the tree and both reads and writes query it. Instead of using a lock we leverage the Linux radix tree implementation that uses read-copy update [42] to provide more scalable, concurrent access to a file.

Figure 8 shows the results (labeled "NOVA-lockfree") on our 80-core machine. 4 KB read performance scales from 2.9 Mops/s for one thread to 183 Mops/s with 80 threads (63×). The changes improve write performance as well, but write bandwidth saturates at twenty threads because our NVMM is attached to one of four NUMA nodes and each node has twenty threads.

Adding fine-grain locking for ranges of a file is possible for ext4-DAX and xfs-DAX, and it would improve performance when running on any storage device.

Using the radix tree to store file layout information would be more challenging since ext4 and xfs make updates to file layout information immediately persistent in the file's inode and indirect blocks. This is necessary to avoid reading the data from disk when the file is opened, which would be slow on block device. Since NVMM is much faster, NOVA can afford to scan the inode's log on `open` to construct the radix tree in DRAM.

An alternative solution for ext4 and xfs would be to replace VFS's per-inode reader/write semaphore with a CST semaphore [33] (or some other more scalable semaphore). The ext4-CSTlock line in the figure shows the impact on ext4-DAX: Performance scales from 2.1 Mops/s for one thread to 45 Mops/s for eighty threads (21×). The gains are not as large as the approach we implemented in NOVA, and they only apply to reads. Both of these approaches could coexist.

4.3 Directory Accesses

Scalable directory operations are critical in multi-program, data intensive workloads. Figure 9 shows that creating files in private directories only scales to twenty cores. Min *et al.* identify the root cause, but do not offer a solution: VFS takes a spinlock to add the new inode to the superblock's inode list and a global inode cache. The inode list includes all live inodes, and the inode cache provides a mapping from inode number to inode addresses.

We solve this problem and improve scalability for the inode list by breaking it into per-CPU lists and protecting each with a private lock. The global inode cache is an open-chaining hash table with 1,048,576 slots. We modify NOVA to use a per-core inode cache table. The table is distributed across the cores, each core maintains

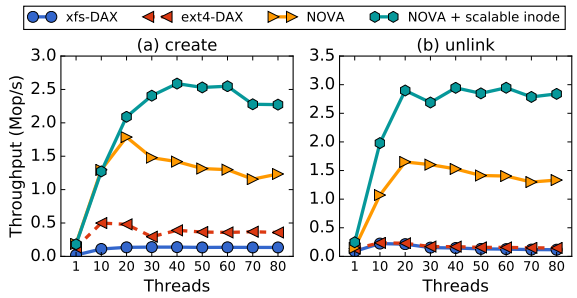

Figure 9. Concurrent `create` and `unlink` throughput. The `create` and `unlink` operations are not scalable even if performed in isolated directories, because Linux protects the global inode lists and inode cache with a single spinlock. Moving to per-cpu structures and fine-grain locks improves scalability above 20 cores.

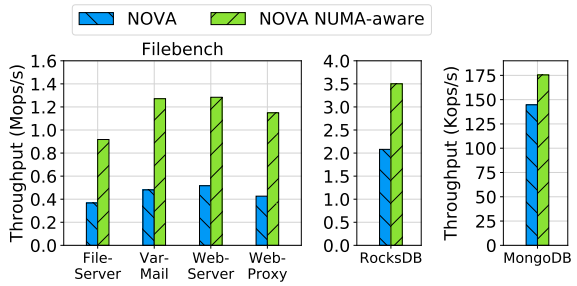

Figure 10. NUMA-awareness in the file system. Since NVMM is memory, NUMA effects impact performance. Providing simple controls over where the file system allocates NVMM for a file lets application run threads near the data they operate on, leading to higher performance.

a radix tree that provides lock-free lookups, and threads on different cores can perform inserts concurrently. In Figure 9, the "NOVA + scalable inode" line shows the resulting improvements in scaling.

Updates to shared directories also scale poorly due to VFS locking. For every directory operation, VFS takes the inode mutexes of all the affected inodes, so operations in the shared directories are serialized. The `rename` operation is globally serialized at a system level in the Linux kernel for consistent updates of the dentry cache. Fixing these problems is beyond the scope of this paper, but recent work has addressed them [5, 61].

4.4 NUMA Scalability

Intelligently allocating memory in NUMA systems is critical to maximizing performance. Since a key task of NVMM file systems is allocating memory, these file systems should be NUMA-aware. Otherwise, poor data placement decisions will lead to poor performance [20].

We have added NUMA-aware features to NOVA to understand the impact they can have. We created a new `ioctl` that can set and query the preferred NUMA node for the file. A NUMA node represents a set of processors and memory regions that are close to one another in terms of memory access latency. The file system will try to use that node to allocate all the metadata and data pages for that file. A thread can use this `ioctl` along with Linux's CPU affinity mechanism to bind itself to the NUMA node where the file's data and metadata reside.

Figure 10(left) shows the result of Filebench workloads running with fifty threads. The NVMM is attached to NUMA node 0. Without the new mechanism, threads are spread across all the NUMA nodes, and some of them are accessing NVMM remotely. Binding threads to the NUMA node that holds the file they are accessing improves performance by 2.6× on average.

The other two graphs in Figure 10 measure the impact on RocksDB and MongoDB [47]. We modified RocksDB to schedule threads on the same NUMA node as the `SSTable` files using our `ioctl`, and ran `db_bench readrandom` benchmark with twenty threads. Similarly, we modified MongoDB to enable NUMA-aware thread scheduling, and ran read-intensive (95% read, 5% update) YCSB benchmark [18] with twenty threads. For both workloads, the data set size is 30 GB. The graphs show the result: NUMA-aware scheduling improves RocksDB and MongoDB performance by 68% and 21%, respectively.

5 Conclusion

We have examined the performance of NVMM storage software stacks to identify the bottlenecks and understand how both applications and the operating system should adapt to exploit NVMM performance.

We examined several applications and identified several simple techniques that provide significant gains. The most widely applicable of these use FLEX to move writes to user space, but implementing `msync` in userspace and assiduously avoiding metadata operations also help, especially on adapted NVMM file systems. Notably, our results show that FLEX can deliver nearly the same level of performance as building crash-consistent data structures in NVMM but with much less effort.

On the file system side, we evaluated solutions to the problems of inefficient logging in adapted NVMM file systems, multicore scaling limitations in file systems and the Linux's VFS layer, and the novel challenge of dealing with NUMA affects in the context of NVMM storage.

Overall, we find that although there are many opportunities for further improvement, the efforts of systems designers over the last several years to prepare systems for NVMM have been largely successful. As a result, there are a range of attractive paths for legacy applications to follow as the migrate to NVMM.

Acknowledgments

This work was supported in part by CRISP, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. We are also thankful to Anton Gavriliuk and Thierry Fevrier from HP for their support and hardware access.

References

[1] LMDB app-level crash consistency. https://www.openldap.org/lists/openldap-devel/201410/msg00004.html.

[2] M. Andrei, C. Lemke, G. Radestock, R. Schulze, C. Thiel, R. Blanco, A. Meghlan, M. Sharique, S. Seifert, S. Vishnoi, D. Booss, T. Peh, I. Schreter, W. Thesing, M. Wagle, and T. Willhalm. SAP HANA Adoption of Non-volatile Memory. *Proc. VLDB Endow.*, 10(12):1754–1765, Aug. 2017.

[3] J. Arulraj, J. Levandoski, U. F. Minhas, and P.-A. Larson. Bztree: A High-performance Latch-free Range Index for Non-volatile Memory. *Proc. VLDB Endow.*, 11(5):553–565, Jan. 2018.

[4] M. S. Bhaskaran, J. Xu, and S. Swanson. Bankshot: Caching Slow Storage in Fast Non-volatile Memory. In *Proceedings of the 1st Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, INFLOW '13, pages 1:1–1:9, New York, NY, USA, 2013. ACM.

[5] S. S. Bhat, R. Eqbal, A. T. Clements, M. F. Kaashoek, and N. Zeldovich. Scaling a File System to Many Cores Using an Operation Log. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 69–86, New York, NY, USA, 2017. ACM.

[6] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An Analysis of Linux Scalability to Many Cores. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–16, Berkeley, CA, USA, 2010. USENIX Association.

[7] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich. Non-scalable locks are dangerous. In *Proceedings of the Linux Symposium*, pages 119–130, 2012.

[8] M. J. Breitwisch. Phase change memory. *Interconnect Technology Conference, 2008. IITC 2008. International*, pages 219–221, June 2008.

[9] A. M. Caulfield, T. I. Mollov, L. A. Eisner, A. De, J. Coburn, and S. 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, pages 387–400, New York, NY, USA, 2012. ACM.

[10] C. Chen, J. Yang, Q. Wei, C. Wang, and M. Xue. Optimizing File Systems with Fine-grained Metadata Journaling on Byte-addressable NVM. *ACM Trans. Storage*, 13(2):13:1–13:25, May 2017.

[11] J. Chen, Q. Wei, C. Chen, and L. Wu. FSMAC: A file system metadata accelerator with non-volatile memory. In *Mass Storage Systems and Technologies (MSST), 2013 IEEE 29th Symposium on*, pages 1–11. IEEE, 2013.

[12] D. Chinner. xfs: updates for 4.2-rc1, 2015. http://oss.sgi.com/archives/xfs/2015-06/msg00478.html.

[13] Y. Choi, I. Song, M.-H. Park, H. Chung, S. Chang, B. Cho, J. Kim, Y. Oh, D. Kwon, J. Sunwoo, J. Shin, Y. Rho, C. Lee, M. G. Kang, J. Lee, Y. Kwon, S. Kim, J. Kim, Y.-J. Lee, Q. Wang, S. Cha, S. Ahn, H. Horii, J. Lee, K. Kim, H. Joo, K. Lee, Y.-T. Lee, J. Yoo, and G. Jeong. A 20nm 1.8V 8Gb PRAM with 40MB/s program bandwidth. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2012 IEEE International*, pages 46–48, Feb 2012.

[14] A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. *ACM Trans. Comput. Syst.*, 32(4):10:1–10:47, Jan. 2015.

[15] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '11, pages 105–118, New York, NY, USA, 2011. ACM.

[16] N. Cohen, D. T. Aksun, and J. R. Larus. Object-oriented Recovery for Non-volatile Memory. *Proc. ACM Program. Lang.*, 2(OOPSLA):153:1–153:22, Oct. 2018.

[17] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. 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.

[18] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[19] M. Dong and H. Chen. Soft Updates Made Simple and Fast on Non-volatile Memory. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 719–731, Santa Clara, CA, 2017. USENIX Association.

[20] M. Dong, Q. Yu, X. Zhou, Y. Hong, H. Chen, and B. Zang. Rethinking Benchmarking for NVM-based File Systems. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '16, pages 20:1–20:7, New York, NY, USA, 2016. ACM.

[21] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. 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.

[22] Facebook. RocksDB, 2017. http://rocksdb.org.

[23] R. Fackenthal, M. Kitagawa, W. Otsuka, K. Prall, D. Mills, K. Tsutsui, J. Javanifard, K. Tedrow, T. Tsushima, Y. Shibahara, and G. Hush. A 16Gb ReRAM with 200MB/s write and 1GB/s read in 27nm technology. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2014 IEEE International*, pages 338–339, Feb 2014.

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

[25] R. Harris. Windows leaps into the NVM revolution, 2016. http://www.zdnet.com/article/windows-leaps-into-the-nvm-revolution/.

[26] Hewlett Packard Enterprise. HPE Scalable Persistent Memory, 2018. https://www.hpe.com/us/en/servers/persistent-memory.html.

[27] Intel. NVDIMM Namespace Specification, 2015. http://pmem.io/documents/NVDIMM_Namespace_Spec.pdf.

[28] Intel. Intel Architecture Instruction Set Extensions Programming Reference, 2017. https://software.intel.com/sites/default/files/managed/0d/53/319433-022.pdf.

[29] S. Jeong, K. Lee, J. Hwang, S. Lee, and Y. Won. AndroStep: Android Storage Performance Analysis Tool. In *Software Engineering (Workshops)*, volume 13, pages 327–340, 2013.

[30] S. Jeong, K. Lee, S. Lee, S. Son, and Y. Won. I/O Stack Optimization for Smartphones. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 309–320, San Jose, CA, 2013. USENIX.

[31] J. Kang, B. Zhang, T. Wo, W. Yu, L. Du, S. Ma, and J. Huai. SpanFS: A Scalable File System on Fast Storage Devices. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 249–261, Santa Clara, CA, 2015. USENIX Association.

[32] S. Kannan, N. Bhat, A. Gavrilovska, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. Redesigning LSMs for Nonvolatile Memory with NoveLSM. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 993–1005, Boston, MA, 2018. USENIX Association.

[33] S. Kashyap, C. Min, and T. Kim. Scalable numa-aware blocking synchronization primitives. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 603–615, Santa Clara, CA, 2017. USENIX Association.

[34] T. Kawahara. Scalable Spin-Transfer Torque RAM Technology for Normally-Off Computing. *Design & Test of Computers, IEEE*, 28(1):52–63, Jan 2011.

[35] W.-H. Kim, J. Kim, W. Baek, B. Nam, and Y. 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, pages 385–398, New York, NY, USA, 2016. ACM.

[36] Y. Kwon. Personal communication, 2018.

[37] Y. Kwon, H. Fingler, T. Hunt, S. Peter, E. Witchel, and T. 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.

[38] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable DRAM alternative. In *ISCA '09: Proceedings of the 36th Annual International Symposium on Computer Architecture*, pages 2–13, New York, NY, USA, 2009. ACM.

[39] W. Lee, K. Lee, H. Son, W.-H. Kim, B. Nam, and Y. Won. WALDIO: Eliminating the Filesystem Journaling in Resolving the Journaling of Journal Anomaly. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 235–247, Santa Clara, CA, 2015. USENIX Association.

[40] R. Liu, H. Zhang, and H. Chen. Scalable Read-mostly Synchronization Using Passive Reader-Writer Locks. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 219–230, Philadelphia, PA, 2014. USENIX Association.

[41] V. J. Marathe, M. Seltzer, S. Byan, and T. Harris. Persistent Memcached: Bringing Legacy Code to Byte-Addressable Persistent Memory. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*, Santa Clara, CA, 2017. USENIX Association.

[42] P. E. McKenney, J. Appavoo, A. Kleen, O. Krieger, R. Russell, D. Sarma, and M. Soni. Read-copy update. In *AUUG Conference Proceedings*, page 175. AUUG, Inc., 2001.

[43] A. Memaripour, A. Badam, A. Phanishayee, Y. Zhou, R. Alagappan, K. Strauss, and S. Swanson. Atomic In-place Updates for Non-volatile Main Memories with Kamino-Tx. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 499–512, New York, NY, USA, 2017. ACM.

[44] Micron. 3D XPoint Technology, 2017. http://www.micron.com/products/advanced-solutions/3d-xpoint-technology.

[45] Micron. Hybrid Memory: Bridging the Gap Between DRAM Speed and NAND Nonvolatility, 2017. http://www.micron.com/products/dram-modules/nvdimm.

[46] C. Min, S. Kashyap, S. Maass, and T. Kim. Understanding Manycore Scalability of File Systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 71–85, Denver, CO, 2016. USENIX Association.

[47] MongoDB, Inc. MongoDB, 2017. https://www.mongodb.com.

[48] H. Noguchi, K. Ikegami, K. Kushida, K. Abe, S. Itai, S. Takaya, N. Shimomura, J. Ito, A. Kawasumi, H. Hara, and S. Fujita. A 3.3ns-access-time 71.2uW/MHz 1Mb embedded STT-MRAM using physically eliminated read-disturb scheme and normally-off memory architecture. In *Solid-State Circuits Conference (ISSCC), 2015 IEEE International*, pages 1–3, Feb 2015.

[49] G. Oh, S. Kim, S.-W. Lee, and B. Moon. SQLite Optimization with Phase Change Memory for Mobile Applications. *Proc. VLDB Endow.*, 8(12):1454–1465, Aug. 2015.

[50] D. Park and D. 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, 2017. USENIX Association.

[51] T. S. Pillai, V. Chidambaram, R. Alagappan, S. Al-Kiswany, A. C. Arpaci-Dusseau, and R. 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)*, pages 433–448, Broomfield, CO, Oct. 2014. USENIX Association.

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

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

[54] redislabs. Redis, 2017. https://redis.io.

[55] J. Seo, W.-H. Kim, W. Baek, B. Nam, and S. H. Noh. Failure-Atomic Slotted Paging for Persistent Memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 91–104, New York, NY, USA, 2017. ACM.

[56] Y. Son, S. Kim, H. Y. Yeom, and H. Han. High-performance transaction processing in journaling file systems. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 227–240, Oakland, CA, 2018. USENIX Association.

[57] SQLite. SQLite, 2017. https://www.sqlite.org.

[58] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams. The missing memristor found. *Nature*, 453(7191):80–83, 2008.

[59] Symas. Lightning Memory-Mapped Database (LMDB), 2017. https://symas.com/lmdb/.

[60] V. Tarasov, E. Zadok, and S. Shepler. Filebench: A flexible framework for file system benchmarking. *USENIX; login*, 41, 2016.

[61] C.-C. Tsai, Y. Zhan, J. Reddy, Y. Jiao, T. Zhang, and D. E. Porter. How to Get More Value from Your File System Directory Cache. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 441–456, New York, NY, USA, 2015. ACM.

[62] S. Venkataraman, N. Tolia, P. Ranganathan, and R. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies*, FAST '11, pages 5–5, San Jose, CA, USA, February 2011. USENIX Association.

[63] H. Volos, S. Nalli, S. Panneerselvam, V. Varadarajan, P. Saxena, and M. 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.

[64] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight Persistent Memory. In *ASPLOS '11: Proceeding of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, New York, NY, USA, 2011. ACM.

[65] D. Vučinić, Q. Wang, C. Guyot, R. Mateescu, F. Blagojević, L. Franca-Neto, D. L. Moal, T. Bunker, J. Xu, S. Swanson, and Z. Bandić. DC Express: Shortest Latency Protocol for Reading Phase Change Memory over PCI Express. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, FAST '14, pages 309–315, Santa Clara, CA, 2014. USENIX.

[66] M. Wilcox. Add support for NV-DIMMs to ext4, 2014. https://lwn.net/Articles/613384/.

[67] J. Xu and S. 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, Feb. 2016. USENIX Association.

[68] J. Xu, L. Zhang, A. Memaripour, A. Gangadharaiah, A. Borase, T. B. Da Silva, S. Swanson, and A. 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.

[69] J. Yang, D. B. Minturn, and F. 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.

[70] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He. NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems. In *13th USENIX Conference on File and Storage Technologies*, FAST '15, pages 167–181, Santa Clara, CA, Feb. 2015. USENIX Association.

[71] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi. Kiln: Closing the Performance Gap Between Systems With and Without Persistence Support. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 421–432, New York, NY, USA, 2013. ACM.

[72] R. Zwisler. Add support for new persistent memory instructions. https://lwn.net/Articles/619851/.