HTMFS: Strong Consistency Comes for Free with Hardware Transactional Memory in Persistent Memory File Systems

Jifei Yi, Mingkai Dong, Fangnuo Wu, and Haibo Chen, *Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University*

https://www.usenix.org/conference/fast22/presentation/yi-htmfs

This paper is included in the Proceedings of the 20th USENIX Conference on File and Storage Technologies.

February 22–24, 2022 • Santa Clara, CA, USA

978-1-939133-26-7

Open access to the Proceedings of the 20th USENIX Conference on File and Storage Technologies is sponsored by USENIX.

HTMFS: Strong Consistency Comes for Free with Hardware Transactional Memory in Persistent Memory File Systems

Jifei Yi, Mingkai Dong, Fangnuo Wu, Haibo Chen
Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China

Abstract

File system designs are usually a trade-off between performance and consistency. A common practice is to sacrifice data consistency for better performance, as if high performance and strong consistency cannot be achieved simultaneously. In this paper, we revisit the trade-off and propose HOP, a lightweight hardware-software cooperative mechanism, to present the feasibility of leveraging hardware transactional memory (HTM) to achieve both high performance and strong consistency in persistent memory (PM) file systems. The key idea of HOP is to pick the updates visible to the file system interface and warp them into HTM. HOP adopts an FS-aware Optimistic Concurrency Control (OCC)-like mechanism to overcome the HTM capacity limitation and utilizes cooperative locks as fallbacks to guarantee progress. We apply HOP to build HTMFS, a user-space PM file system with strong consistency. In the evaluation, HTMFS presents up to 8.4× performance improvement compared to state-of-the-art PM file systems, showing that strong consistency can be achieved in high-performance persistent memory.

1 Introduction

File systems are the key cornerstones of many storage services such as key-value stores and databases and applications that persistently store data. In the early days, file systems are designed for performance with loose consistency guarantees. For example, FFS [47] relies on the clean unmount of the file system to avoid consistency issues. In case of crash or power shortages, file system users have to invoke and wait for the lengthy file system consistency checker, i.e., `fsck`, which will detect consistency issues and attempt to recover but with no guarantee [26].

Nowadays, with the speedup of storage devices and their widespread use in applications, performance is not the only feature that applications need. Applications also require strong consistency in order to provide reliable services. For example, key-value stores and databases need strong crash consistency to guarantee that all returned writes are persisted and can be correctly read after a system crash. Upon file systems with no or weak consistency guarantee, these applications have to either compromise on the consistency level or use complicated mechanisms to provide reliable storage. Programming efforts can also be reduced if the file system can provide strong consistency.

The strong consistency of the file system implies per-request sequential consistency, which consists of two aspects. First, for arbitrary file system requests, the modification to the file system states observed by concurrent tasks should be atomic. Most file systems use inode-level locks, which ensures the modification order of different requests to guarantee sequential consistency. Second, whenever the system crashes, after a reboot, all previous file system requests should satisfy the all-or-nothing semantics, i.e., all changes to the file system state by a single file system request should be applied or none of them should be applied. File systems do not necessarily guarantee strong crash consistency. For example, ZoFS [16] do not provide the atomicity of data modification. Suppose a writer crashes halfway through writing; it is possible for a reader to read the partially updated value after the system recovers.

At the same time, modern storage devices have become faster and different. The emerging persistent memory (PM) enforces memory with durability. Consequently, file systems can use load/store instructions to access PM storage with near-DRAM performance. Several PM file systems [11, 13, 16–18, 37, 42, 74, 80] are proposed to exploit the PM characteristics; many of them provide strong consistency.

However, existing PM file systems still require complicated and expensive mechanisms, such as journaling [6, 10] and shadow paging [7,64], for strong crash consistency. Journaling has the double writes problems, while shadow paging needs to propagate the changes to an atomic update, thus it only fits dedicated data structures. The write amplification is related to the data structure it uses and the pattern it writes.

Previous approaches are limited to atomicity unit of CPU writes. Intel's restricted transactional memory (RTM) [32] can provide atomicity of multiple updates. However, file system is incompatible with RTM naturally. Block device based

Table 1: Crash consistency mechanism comparison. The specific write amplification of shadow paging corresponds to the data structure and write locations. The write set size of RTM is evaluated with sequential writes on our platform.

Mechanism	Write Amplification	Write Set	Data Strucute	Crash Consistency
In-place Update	1	Unlimited	Any	No guarantee
Journaling	>2 (double writes)	Unlimited	Any	Strong
Shadow Paging	>1	Unlimited	Dedicated	Strong
Soft Updates	1	Unlimited	Dedicated	Weak
RTM	1	<16k	Any	Strong
HOP	Nearly 1	Unlimited	Any	Strong

file systems access data via IO, which will abort the RTM. Although persistent memory can be accessed directly by CPU load/store instructions, the PM write operations need to be persisted with the help of cache line flush instructions (such as `clflush`, `clflushopt`, and `clwb`) which will abort the RTM.

Recently, Intel proposes its second-generation Optane Persistent Memory products, which in cooperation with the new Xeon platforms enable enhanced asynchronous DRAM refresh (eADR) technique that embraces the CPU cache in the domain of persistence in case of crashes [29]. In particular, the platforms guarantee the persistence of memory writes once they become globally visible, which means that data modification to the persistent memory no longer requires cache line flush for persistence. This gives us the possibility of combining RTM and persistent memory to provide atomicity, concurrency, and persistence at the same time.

Although RTM can be used with PM, several challenges prevent RTM-PM from being used directly in PM file systems. At first, users use file systems to process large data storage and retrieval. However, RTM is limited in both read and write set size, thus can easily abort due to file data copy. Second, there are certain dependencies in the code paths of FS-related system calls. For example, path-related operations (such as `open` and `mkdir`) must be preceded by path lookups, and file indexing must be done before reading and writing a file. The operations can be lengthy and may include memory accesses that do not need to be tracked by RTM. Simply wrapping the entire operation within an RTM not only easily leads to capacity abort, but also increases the probability of conflict aborts.

In this paper, we propose HOP[1], a lightweight hardware-software cooperative mechanism for providing strong consistency in PM file systems. HOP builds on the recent eADR-compliant platforms and leverages Hardware Transactional Memory (HTM) to guarantee the atomic durability of file system updates. To address the capacity limitation of HTM, HOP adopts an OCC[2] [41]-like mechanism to chop a large file system request into smaller pieces, while retaining both concurrent consistency and crash consistency during the execution. To guarantee file system progress, HOP designs cooperative locks as the fallback of HTM. The comparison of HOP and other crash consistency mechanisms is shown in Table 1.

To illustrate HOP, we implement HTMFS, a user-space PM file systems base on ZoFS. Evaluation using FxMark [52], Filebench [72], LevelDB [24], and TPC-C [15] on SQLite [70] shows that HOP outperforms state-of-the-art PM file systems, achieving a similar performance to the weak consistency FS implemention while providing strong consistency. With carefully designed fine-grained concurrency control, HTMFS provides even better performance in competitive cases.

The contributions of the paper include:

- The design of HOP, a lightweight hardware-software cooperative mechanism to provide strong consistency in persistent memory file systems (§3);
- The implemention of HTMFS which provides both strong consistency and performance using HOP (§4);
- Comprehensive evaluation that shows that HTMFS outperforms state-of-the-art persistent memory file systems, proving the effectiveness of HOP (§5).

2 Background and Motivation

In this section, we introduce the background knowledge and motivation of our work.

2.1 File System Consistency and Performance

The original file systems are not built with consistency as the priority, e.g., FFS has no consistency guarantee if a crash happens before a clean unmount [26].

An ancient `fsck` tool simply makes the file system mountable [26], without any guarantees on data consistency or persistency. A `fsck` tool helps recover, repair, and refresh the system.

Without file systems providing consistency, applications need to take responsibility for guaranteeing consistency. For example, to guarantee that data are persisted to file A in atomic, applications need to do the following operations in sequence.

1. Create file B with the same content as file A;
2. Write new data to file B;

[1] HOP is short for Hardware-assisted Optimistic Persistence.
[2] Optimistic Concurrency Control

3. Flush file B to guarantee that the new data is persisted in storage;
4. Rename file B as file A;
5. Sync the directory change;

This obviously is costly for applications. Thus some file systems, such as Ext4 and NOVA [80], provide strong consistency as an optional feature.

However, strong consistency does not come for free. Ext4 uses data journal to provide atomic updates for the data, and therefore has the problem of double writes as shown in Table 1. NOVA can use the CoW (copy-on-write) approach to update data atomically. However, CoW may degrade NOVA's performance by up to more than 60% in our evaluation part.

2.2 Persistent Memory and PM File Systems

Persistent Memory (PM) is an emerging storage technology that enforces byte-addressable memory with persistence. With the same interfaces as volatile memory (i.e., DRAM), data written to PM are guaranteed to retain across power cycling. As a result, the storage hierarchy has changed.

Based on these changes, several PM file systems are proposed to better exploit the PM characteristics for better performance. These file systems revisit existing crash consistency mechanisms in the new scenarios brought by PM, rather than exploring fundamentally different (and more efficient) crash consistency mechanisms of file systems. The only difference would be leveraging atomic instructions to provide small updates up to a single cache line. BPFS [13] organizes the whole file system in a tree structure and provides strong consistency via shadow paging and atomic instructions. PMFS [18] introduces fine-grained journaling and combines atomic instructions and optional shadow paging for data consistency. NOVA [80] is a log-structured file system designed for PM, which combines all the atomic instructions, shadow paging, and journaling for strong consistency. SoupFS [17] is a revisit of the soft update technique on PM and provides no strong consistency guarantee.

Traditional file systems, such as Ext4 and XFS, introduce direct access mode (DAX) to bypass page cache in the data path, optimizing their performance when running on PM. However, this doesn't change the crash consistency level of these file systems.

The byte-addressability and persistence of PM also motivate several user-space file systems, e.g., Aerie [74], Strata [42], SplitFS [37], ZoFS [16], and Libnvmmio [11]. Strata and Libnvmmio use logs to guarantee consistency. SplitFS relies on the underlying Ext4 for metadata processing. ZoFS takes the soft update approach to protect data modification, thus only providing weak consistency. It first updates the data in place and then modifies the size of the file to complete the operation. However, if the system crashes before the file size is changed, partial updates may be read by the next read operation.

In summary, PM brings new opportunities in the design of file systems; while existing new file systems still stick to existing mechanisms for crash consistency, leaving the trade-off between performance and strong consistency a lasting barrier towards fast and reliable file systems.

2.3 Hardware Transactional Memory

Transactional memory provides programmers with an easy (and sometimes efficient) approach to implementing concurrent applications. Hardware transactional memory technologies, such as Intel's Restricted Transactional Memory (RTM) in TSX [1] and ARM's TME [46], provide hardware support of transactional memory. Programmers only need to identify the critical section that wraps the shared memory resources and mark it with `xbegin` and `xend` instructions. The transactional memory mechanism will guarantee that the execution of critical sections can be serialized so that no data race occurs. Executing transactions failing to meet the serializable requirements will be aborted by the hardware, which is detected via the cache coherence protocol. Specifically, data writes of an uncommitted transaction are kept in the private cache of that CPU core and only become globally visible when the transaction successfully commits.

Due to the strong affiliation to the cache implementation, the following limitations will cause HTM to abort, which the users should take care of.

Conflict aborts. HTM uses read/write set to track accesses to memory. Cache lines read in the HTM are added to the read set, and cache lines written are added to the write set. Before HTM is successfully committed, if a cache line in the read set is modified or the write set is accessed by another core, this transaction will be aborted due to conflicts. This type of abort may succeed by retrying the transactions.

Capacity aborts. CPU's private cache size is limited; thus, HTM has limited read and write sets. Any transaction that exceeds the read or write set will inevitably be aborted, no matter how many times the transaction is retried.

Other aborts. Besides conflict and capacity aborts, some other sources could abort a transaction, such as interrupts and HTM-incompatible instructions. It depends on the specific scenario to tell whether a simple retry will make the transaction succeed. For example, if a page fault occurs during the transaction, the interrupt will cause the transaction to abort. In this case, a prefault (trigger the page fault in advance) is necessary before retrying the transaction.

2.4 HTM in PM File Systems

HTM was never an option for file system consistency in the era of block-based storage devices. The emergence of byte-addressable persistent memory gives a chance to use HTM in file systems. However, the volatility of CPU cache forces the use of cache line flush instructions for durability, which intrinsically conflicts with the HTM mechanism that stashes in-flight transaction data within the CPU cache. Until January

2021, Intel's new platform included the CPU cache in the persistence domain, meaning that data that reaches the CPU cache can be guaranteed to be durable even in case of crashes and power shortage, it becomes possible to use HTM upon persistent memory. And it goes beyond that. HTM becomes a good companion to be used with persistent memory. According to Intel [29], only globally visible data will be made durable if a power shortage occurs. In other words, HTM in-flight data modifications will be discarded, making HTM a good alternative approach to enforce atomic updates for crash consistency in file systems. HTM seems promising to be used in file systems to provide both crash consistency and concurrency guarantees at the same time.

3 Design

At first sight, it seems straightforward to equip file systems with HTM: simply wrapping each file system request in a pair of `xbegin` and `xend` can guarantee the ACID of the file system request. However, the reality proves that this is far from enough. Due to the long code path and complicated operations in file system requests, wrapping the entire file system request directly within a hardware transaction will frequently (if not always) result in transaction aborts. Directly adopting HTM in a file system will lead to the following three problems:

1. The long code path may permanently cause capacity aborts;
2. The long code path make the transaction easier to abort due to data conflicts;
3. More works need to be repeated in the retry of the abort transaction.

To resolve the above problems, we designed a lightweight hardware-software cooperative mechanism named HOP. Next, we will first introduce what HOP is and then describe how we use HOP to build an RTM-compliant file system, namely HTMFS.

3.1 HOP

To shorten the code path in the HTM, we split a single file system operation into multiple small pieces. When joined together, they will perform similarly to a single huge transaction. This idea is similar to transaction chopping [68].

All memory accesses in file system operations can be classified into three types:

1. Reads;
2. Invisible writes: updates that cannot be observed via the file system interface (such as memory allocation and updates to the shadow pages);
3. Visible writes: updates that can be observed by the file system interface (like timestamp modification, in-place updates, and the change of file size).

To alleviate the capacity aborts caused by complex file system operations, HOP only wraps visible writes (3) in the transactions. Invisible updates are designed to be able to roll back with minimal overhead, while critical reads are protected by sequence counts.

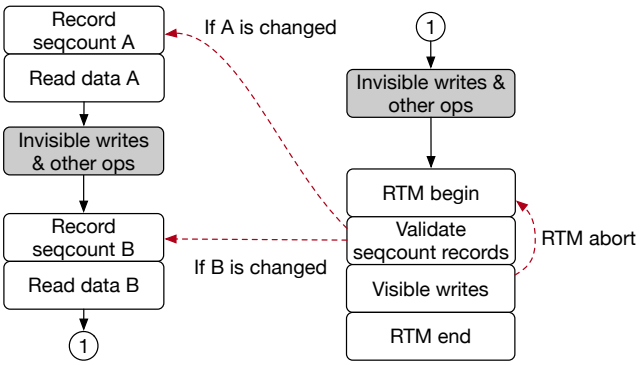


Figure 1: HOP: A transaction wants to read critical data A and B, and write something in atomic. It can read the seqcount and the data in sequence, and validate them in the same RTM with the visible writes to ensure A and B do not change during the whole execution; i.e., the whole process can be considered as an atomic transaction.

In more detail, we first perform all reads and invisible writes outside the RTM. Then we wrap the visible writes to persistent memory using an RTM to complete them atomically. However, not applying any protection to the first part may lead to concurrency errors. HOP ensures concurrent consistency by protecting the fields that may cause concurrency errors by sequence counts. As shown in Figure 1, when we want to access the protected fields outside an RTM, we will first record the corresponding sequence count and then access the persistent memory. These sequence numbers will be validated when entering the RTM-protected region to ensure that the rest remains unchanged throughout the process as long as the RTM commits successfully. If the validation fails (i.e., there is a sequence count that has been changed), HOP will roll back to the first changed point to restart the transaction. For example, if we find that A's seqcount has not changed, but B's seqcount has been modified, we will take the red dotted line "B is changed" to re-record B's seqcount and re-read B.

Besides a modified seqcount, many reasons (introduced in §2.3) may also cause the aborts. If it is an accidental abort caused by an interruption or something else, retrying the RTM transaction again ("RTM abort" in Figure 1) is enough, as going back to the very beginning would cause unnecessary overhead.

Discussion of concurrency correctness. Next, we will discuss all concurrency scenarios (read-read, write-write, write-read, and read-write) in the HOP. Read-read will not bring problems anytime. Since potentially conflicting writes in the HOP are protected by RTM, two conflicting writes will cause each other to conflict abort until one of them succeeds (or keep aborting each other, making it impossible to move forward, which we will avoid by other methods).

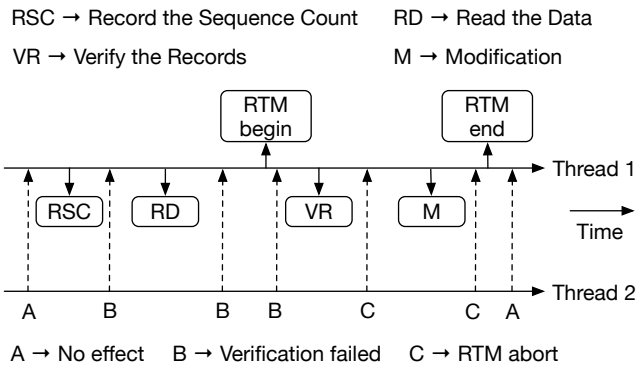


Figure 2: HOP: Thread 2 modifies the sequence count recorded (or to be recorded) by thread 1 at different times, leading to three results. A has no effect on thread 1, while B and C both cause thread 1 to redo, thus guaranteeing that there will be no concurrency errors.

Write-read/read-write, however, will cause an RTM to abort if they conflict, given that potentially confliting write operations are all executed inside the RTM. Any abort will trigger a redo in Figure 1, so a successfully committed transaction guarantees that no concurrency errors exist.

For the read-write scenario, as shown in Figure 2, thread 1 first reads some variables protected by the sequence count, then begins the RTM, validates the sequence numbers, and performs all visible write operations. Thread 2 is simplified to modify the conflict variable at some point in time. The time point modified by thread 2 can be divided into three ranges, resulting in three consequences.

- Result A: If Thread 2 modifies seqcount before Thread 1 reads it or after Thread 1 finishes all operations, it has no effect on the result of Thread 1.
- Result B: If Thread 2 modifies seqcount between Thread 1 reading seqcount and verifying seqcount, it causes Thread 1 to fail validation and thus redo the whole task.
- Result C: If Thread 2 modifies seqcount after Thread 1 verifies successfully (while before the RTM ends), it causes an RTM abort in Thread 1 as it modified Thread 1's read set, thus redoing the whole task.

With HOP, we can break the RTM capacity limit. Then we will introduce how HOP helps to build HTMFS through some specific operations in the file system.

3.2 File Operations

3.2.1 Data Read

For data reads, we use a seqcount-based method to make it atomic. Specifically, the structure of a file is shown in Figure 3, for each page, we first record the persistent pointer (with the sequence count) of the last page, and then read its content. After finishing reading, we verify that the pointers to all records and their sequence counts are unchanged. We will re-read the page that changed and then verify all the sequence

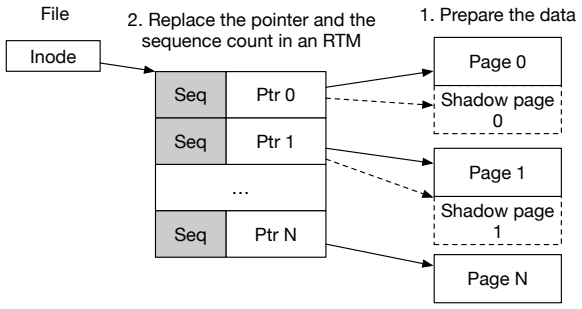


Figure 3: A file is organized in a page table like manner. A single-page update is performed directly wrapped in a transaction. Multi-page updates need to allocate new pages (the shadow pages) for the data, and then copy the new data to the shadow pages. Pointers and the corresponding sequence counts are updated atomically in an RTM.

counts again until all records in this progress stay stable. As only data writes modify the sequence counts or the persistent pointers, we can ensure that there are no changes to the pages we read throughout the entire read operation.

3.2.2 Data Write

Data updates are the foundation of file system operations. One of the major challenges that HTMFS faces is the conflicts between RTM's capacity limitation and the large amount of data involved in file system operations. As a result, directly wrapping the whole file system operation in an RTM transaction will inevitably cause capacity aborts that prevent the operations from being completed.

To address this issue, we propose a hybrid approach that combines the copy-on-write and journalling to convert data updates to metadata updates that can be embedded in the RTM transactions.

Small writes, which fit in a single PM page, are wrapped in an RTM directly. For large data writes, as shown in Figure 3, our strategy first writes data to the persistent memory so that large bulk of data can be represented by pointers, enabling it to be easily embedded in the limited RTM transaction. To explain in detail, we first allocate PM space to store the data. Note that the allocation information is in DRAM, which will not be persisted after a crash. But the data is in PM. Then we start an RTM transaction, in which file system metadata is modified, including the modification of allocation metadata. The persistence point is the RTM commit. Upon a successful commit, the file data and metadata are persistent in an atomic approach. Upon a transaction abort, no changes to the file systems are visible after reboots, with the only exception that the file data are written to the unallocated PM, which is benign most of the time. But the blocks may have leaked after a system crash. Time-consuming scanning of the whole persistent memory can help retrieve the leaked space. To eliminate the recovery process, we design a new allocator based on the free list (as shown in Figure 4) to prevent a memory leak.

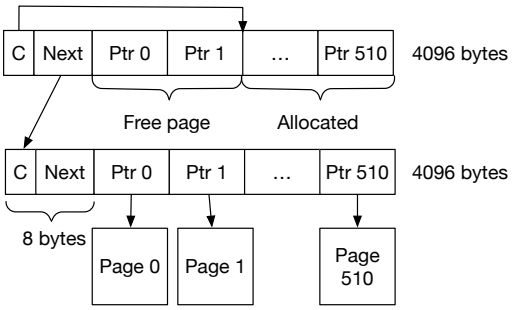


Figure 4: The atomical allocator. `C` stands for `Current`. The pages before `Current` are free pages, while the after is allocated.

3.2.3 Allocation

We split the allocation into two parts: first we move the allocated blocks into a temporal allocating list, which has the same structure as the unallocated space list. Then we simply discard the temporal list inside a transaction to persist the allocation. If a crash happens, we add the temporal list into the unallocated space list to prevent a memory leak.

To ensure that the file system does not reference any unallocated data block, usually the file system modifies (or removes) the reference to the data block before releasing the block. A memory leak may also occur if the file system crashes after a reference to a block of data has been removed (when this block of memory has not yet been freed). When we need to free multiple data blocks, we may also have a crash halfway through the release. An easier way to ensure atomicity is wrapping all these operations in a transaction, but RTM is likely to have a capacity abort. To solve this, we adopt a method similar to the allocation for the free operations. Only operations that must be completed atomically are placed inside RTM, thus avoiding the probability of a capacity abort.

3.3 Directory Operations

3.3.1 Path Walk

File systems usually use a tree structure to maintain the directory hierarchy. Path walking is a quite common scenario in file systems. Many file-system-related system calls require path walking, such as `open`, `mkdir`, `unlink`, etc. These functions will first do a path walk, where the file system will split the full path by slash. It then looks for each level of pathname in turn, starting from the root directory, until it reaches the last level. Then the specified operation (e.g. `open`) is performed on the last file name in the last directory.

All operations that need to walk the path will record the sequence numbers of the directory entries (dentry) it visits, and validate these sequence numbers in the same RTM with the data writes (as shown in Figure 1). Take `touch /a/b` as an example, this operation will first search the dentry `a` in the root directory (`/`). When it finds the matching dentry, it will first record the sequence number of the dentry (`Dseq` of the dentry `a`) and then read the inode number of the directory `a`.

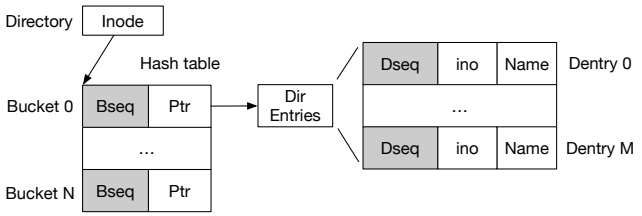


Figure 5: HTMFS uses hash tables to manage the directory entries. `Bseq` is used to serialize changes to directory entries within the same bucket. This prevents the insertion of two files with the same name, etc.

Then it will begin a transaction, validate the sequence number read previously, insert the new dentry `b` into the directory `/a`, and finally commit the transaction.

3.3.2 Directory Updates

We do not use locks on the directory inode to protect updates to the same directory (add/remove a dentry). Instead, we use a separate seqcount in each bucket, and all insert operations need to modify the seqcount of the corresponding bucket in the hash table (Bseq in Figure 5). When inserting multiple different directory entries into a directory simultaneously, the competition will result in only one directory insertion operation succeeding. At the same time, the other will have to redo the whole operation because the sequence number has been modified. In the process of redoing the operation, the operation will find that a directory entry with the same name already exists in the directory and return the error code `EEXIST`.

In file systems, directories can be removed by the system call `rmdir`. However, only empty directories can be removed to avoid deleting useful data accidentally. The utility `rm` can be used to remove a non-empty directory with a parameter `-r`, which will remove directories and their contents recursively. In the implementation, it will remove all the children of the directory first and then delete the empty directory from the file system tree by `rmdir`. This process does not break the restriction that only empty directories can be removed in file systems.

We need to consider the situation that process `A` tries to touch a new file `/a/b/c` into an empty directory `/a/b` while process `B` attempts to delete this empty directory `/a/b`.

As shown in Figure 1, `A` will first walk the path and record the sequence count `Dseq` of `/a/b`'s dentry. Then it will validate the sequence number in the same RTM with the insertion of `/a/b/c`. If the sequence number has been changed before the validation, then `A` will fail to validate it and rollback (lookup the path again). If the sequence number is changed after `A` succeeds in validation, the modification of this sequence number (`B` changes it in another transaction) will cause the transaction of `A` to abort. Then `A` will be rolled back and do again. In the new round of path lookup, it will find that the directory `/a/b` does not exist, which has the same results as if `B`'s entire operation had finished before `A`, will not cause

any problem.

B also needs to validate and modify this sequence number in the same RTM with the operation that deletes this empty directory. Once successfully committed, the insert operation that has not finished the path walking will not be able to find this directory (/a/b), the others will fail to validate the Dseq or be aborted by the modification of the Dseq, thus protecting the correctness of this case. If B is aborted by A because of a conflict, B will find that the directory is not empty when it retries, thus returning ENOTEMPTY as if it is trying to delete a non-empty directory, which is the same as if A operation is finished atomically before B.

This Dseq guarantees that the results of both operations in this case are consistent with a serial execution. So it is no longer necessary to use locks to protect its concurrent correctness.

3.4 Other File Types

Symbolic links. Symbolic links are first expanded to a normal path, and the new path will be returned to the dispatcher, which will re-dispatch the file request. The rest of the operation is just like a normal file.

3.5 The Timestamps

There are several timestamps in file systems to record some information about a file.

- Access timestamp (atime): the last time the file was accessed.
- Modified timestamp (mtime): the last time the file's contents were modified.
- Changed timestamp (ctime): the last time the metadata of the file was changed.

Many file system operations (even read operations such as read, stat, etc.) will modify some of the timestamps. Modifying the timestamp should theoretically happen at the same time as accessing the file, so they need to be done atomically. We need to modify the timestamps in the same transaction as the other operations. Here we observe that placing accesses and modifications to critical variables at the end of a transaction significantly reduces the probability of an abort due to conflicts.

3.6 The Special Case: Rename

Both unlink and rmdir can only remove leaf nodes (files and empty directories) from the file system. Rename, however, has no such limitation and can move a filesystem subtree to another location.

Rename is a special operation that requires atomically removing a directory or a file from the file system tree and adding it to another directory. Usually we will hold locks on both directories to ensure the correctness. However, it may happen that two rename operations both hold a lock and wait to take each other's lock, resulting in a deadlock. This problem can be solved by comparing the two locks and taking the locks in a certain order. However, taking only the locks of the two directories modified cannot prevent the occurrence of a cycle. As shown in Figure 6, there are two path /A/B/C and /X/Y/Z in the directory tree. There are two rename operations; one wants to rename /A/B to /X/Y/Z/B and another wants to rename /X/Y to /A/B/C/Y.

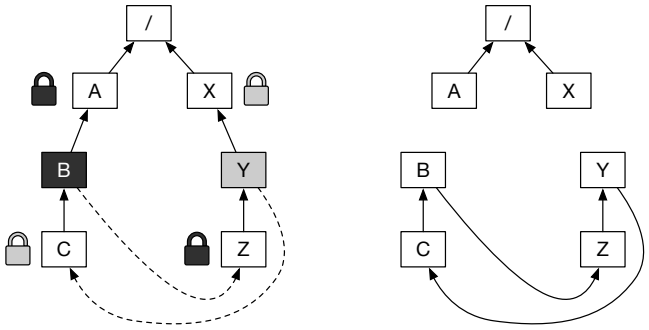


Figure 6: Rename cycle. If rename only locks the parent inode of the source and destination, the rename cycle (outside of the directory tree) may occur.

Take the first operation as an example. 1. First it will walk the path and find the source directory A/B (lock the parent inode A) and the destination /X/Y/Z (lock the inode Z). 2. Then it tries to delete the directory entry B from the directory A and insert a new directory entry B to the directory Z. 3. Finally it will release the two held locks. However, between step 1 and 2, another operation may also finish the path walking and get the two inodes (source Y and destination C). Without other protection, both operations can succeed, thus resulting in a rename cycle. So we need to take extra steps to avoid the cycle, for example, by adding a global rename lock to serialize all rename operations.

We still adopt a lock-free design (HOP) for the rename process. In the path walking (namex), we record the sequence count of all the directories we traversed (as described in § 3.3.1), and finally check if all the sequence counts have changed in one RTM. If there is a change, the namex operation will be executed again from the point of change; if there is no change, the operation of deleting the directory entry and adding it is continued. Since all of the above operations (checking for path changes and modifying directory entries) are done within the same RTM, a successful RTM commit guarantees that the entire rename operation completes atomically. In the preceding example, if both operations complete the path walking and enter the directory modification step (step 2), then when one operation completes, the other operation will abort as its read set is modified, thus re-validating the sequence number and failing because the sequence count has been modified. It then rolls back, redos the path walking and finds the directory tree has changed finally.

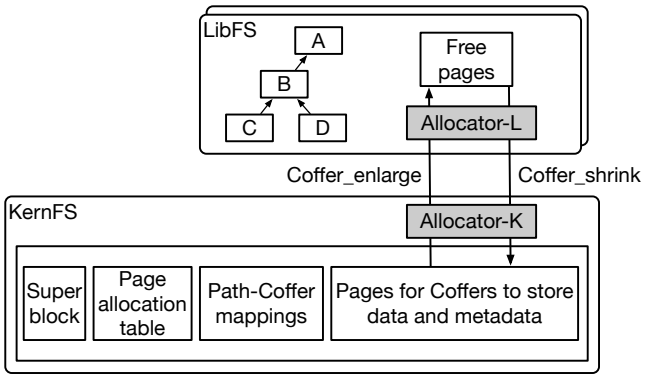


Figure 7: HTMFS consists of a KernFS and a user-space library (LibFS). LibFS calls `Coffer_enlarge` to ask for more PM space from the kernel. When there are too many free pages, LibFS return some to KernFS via `Coffer_shrink`.

4 Implementation

To illustrate the effectiveness of HTMFS, we implement a new file system. After comparing several file systems, we decide to implement HTMFS based on ZoFS [16] because all operations in ZoFS are in user space, thus avoiding the possibility of transaction abort due to system calls.

The overall architecture is shown in Figure 7. ZoFS consists of a kernel-state KernFS and one (or more) user-space file system libraries. HTMFS also consists of two parts, the original KernFS of ZoFS and a new LibFS. In ZoFS, the entire file system tree is divided into multiple zones according to permissions.

4.1 KernFS

KernFS is responsible for maintaining information about all the zones in the entire file system, and the attribution of all persistent memory pages. Each zone has a root page that stores the metadata for that zone. KernFS uses a persistent hash table to store all the zones, where the key is the path prefix of each zone and the value is the relative address of the root page of each zone. When a user-space filesystem library needs to access a path, KernFS uses this hash table to find the root page of that zone and further access that zone.

KernFS manages all PM space globally at page granularity. ZoFS uses a two-level allocation. KernFS allocates PM pages to zones in bulk, and each zone further allocates its pages to store data and metadata. KernFS keeps track of the allocation status of each page, i.e., which zone each page belongs to and which pages are free and can be allocated. In this process, ZoFS uses a global volatile red-black tree to track all free spaces in the allocation table, and another red-black tree [5] to track all allocated spaces and the root page address of the corresponding zone. These volatile data structures can be easily recreated after a system crash.

4.2 LibFS

LibFS is responsible for managing all the metadata and data inside a zone, including mainly files and directories. It contains all the designs in § 3. The file structure, shown in Figure 3, is a three-level structure similar to a page table, supporting files up to 512 GB. Of course, it can be easily extended to support larger files. The directory structure is a hash table as shown in Figure 5.

Since KernFS uses a free list to manage free space, when a zone issues a system call to the kernel to get more free space (`Coffer_enlarge`), KernFS returns a free list. Thus, our LibFS needs to convert the free list to a version recognized by HTMFS (as shown in Figure 4). This prevents modifications to the kernel side.

Fallback path. When RTM fails, we choose to retry or fallback path depending on the return value. We also walk the fallback path when the number of failed retries exceeds the threshold (We choose 60 in the implementation as it gives the best performance when varing the maximum retry number from 10 to 100.). In the fallback path we use inode-level read/write locks for concurrency control and use RTM for crash consistency. When RTM still fails in the fallback path, we use journal as a last resort.

Operations on the normal path will first check if the write lock is held by someone after RTM begins. If the write lock is held by another task, the operation will rollback to the fallback path and try to hold the write lock. If the lock is not held by others during the check, but someone else gets the write lock before the RTM commit, the operation will abort because it's read set has been modified, then retry the RTM operation, and re-check the lock state.

4.3 Prevent RTM abort

There are many causes of RTM abort, starting with RTM capacity abort. The simplest implementation is to wrap the entire file system call in an RTM, and after experimenting we find that most directory and file operations yield capacity abort. After using HOP, HTMFS solve this type of problem.

In our implementation we find that one common cause of RTM abort is page fault, which cannot be predicted. So we prevent page fault failures by first accessing the memory that needs to be accessed and preloading the code to be executed after an RTM abort.

Lastly, the failure is due to conflict, which returns a specific value. In that case HTMFS tries to retry first, which can resolve these conflicts if there is not much competition. If the retries fail a certain number of times, HTMFS fallbacks to the fallback path, i.e., using locks to protect critical code for concurrency control. In fallback path we will first take locks to prevent concurrent accesses and then use HOP to ensure its crash consistency. So it can still meet the strong consistency requirement.

There are some other reasons, such as intermixing AVX and SSE instructions in an RTM, long strings in `REP-MOV*` instructions, etc., which can cause RTM abort [31]. In practice, we found that the `REP-MOV*` instruction used by memcpy will

cause RTM abort in a high probability. So we use cyclic assignments (`SSE2-MOV*`) to replace the memcpy inside RTM.

5 Evaluation

In this section, we evaluate HTMFS against state-of-the-art file systems using different data consistency mechanisms to answer the following questions.

- Can HTMFS's HTM-based hybrid strong crash consistency techniques provide almost as good performance as weak consistency?
- Can HTMFS improve the applications' performance?
- How does HTM improve the performance of file systems?

5.1 Platform Setup

Experiments are conducted on a twenty-eight-core Intel® Xeon® Gold 6330 CPU server. Hyper-threading is disabled, and the CPU frequencies are set to 2.0GHz to get stable results during the evaluation. The server is equipped with 512GB DDR4 DRAM and 1024GB Intel® Optane™ Persistent Memory 200 series.

To evaluate the performance of HTMFS, we compare it against state-of-the-art file systems. ZoFS [16] is evaluated as the baseline. We also evaluate three state-of-art PM-aware file systems (NOVA [80], SplitFS [37], and Libnvmmio [11] on NOVA) to compare. Inode-level locks are used in these file systems. We remove all clflush, clflushopt, clwb, and fence instructions in all of these file systems to improve their performance because these operations are not needed on the eADR platform.

We use FxMark [52], filebench [72], TPC-C [15] on SQLite [70], and LevelDB [24] to evaluate the performance of HTMFS.

5.2 Micro-benchmarks

FxMark includes a set of micro-benchmarks that stress the performance of FS-related system calls. We use FxMark to evaluate the performance and scalability of HTMFS.

Figure 8 shows the performance of file data and metadata operations as the number of threads increases. HTMFS outperforms other file systems in most workloads, including data writes(8(a)(b)) and metadata operations(8(e)(g)(h)). For some workloads, the results of SplitFS and Libnvmmio are not fully displayed, as they get stuck or encounter self-contained errors with an increasing number of threads.

Figure 8a shows the performance for data overwrite operations when different threads overwrite the first 4KB block of different files (DWOL). HTMFS is slower than ZoFS because we replace memcpy (which uses `REP-MOV*` instructions) with `SSE2-MOV*` instructions, which takes more time. If we use `SSE2-MOV*` instructions in ZoFS, the degradation disappears, as ZoFS-SSE2 in the figure shows. Other workloads do not suffer from this degradation because `REP-MOV*`-based memcpy (in ZoFS) only outperforms `SSE2-MOV*`-based memcpy (in HTMFS) when hitting the cache. In DWOL, almost all writes hit the cache, while in other workloads, throughputs are dominated by writes to PM.

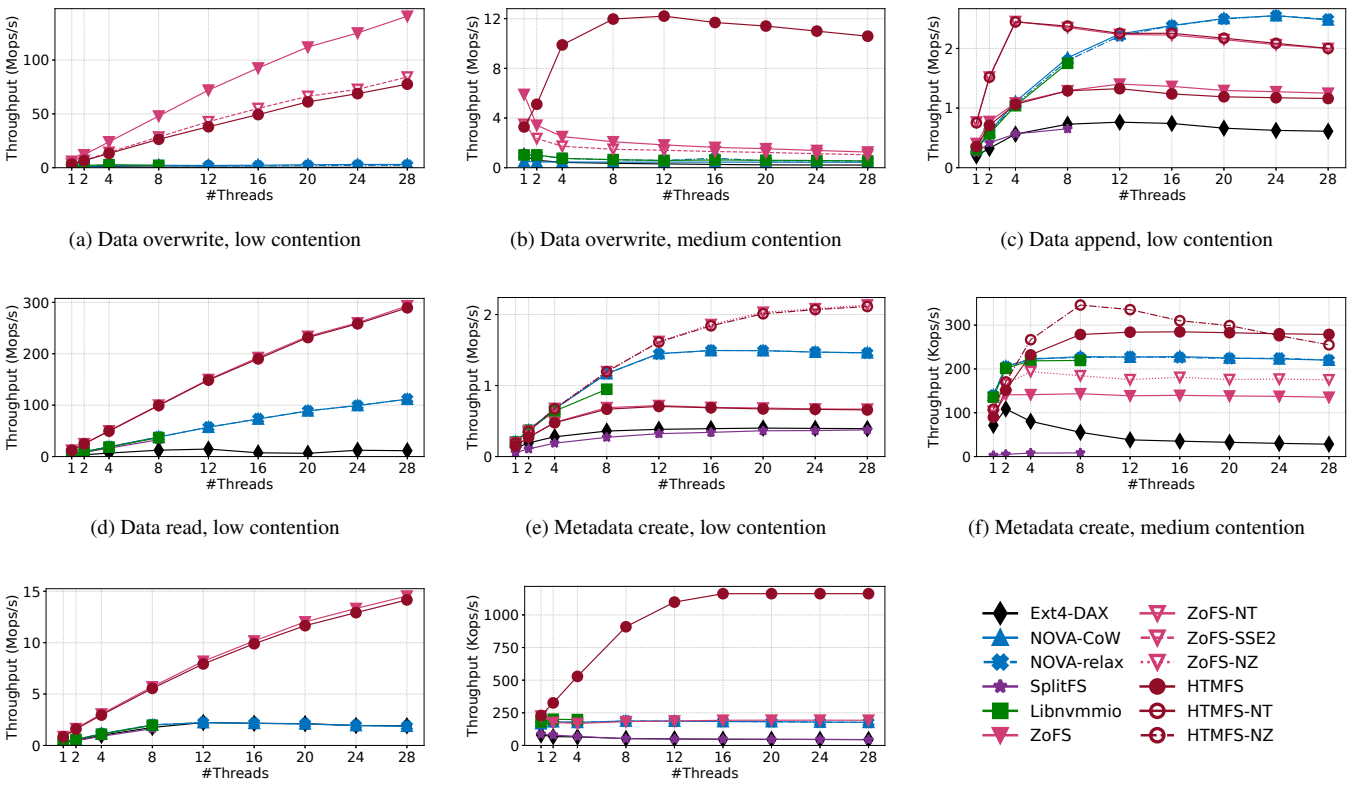
With the medium sharing level, where different threads overwrite different blocks in a shared file (DWOM), HTMFS shows the best scalability. In contrast, the throughputs of other file systems drop as the number of threads increases, as shown in Figure 8b. When there are 28 threads, the throughput of HTMFS is 8.4× of ZoFS. The good scalability of HTMFS mainly comes from the HTMFS's lock-free design. For tests like DWOL and DWOM where the write operations only write to the cache, this part of the difference is magnified to become obvious.

For data append (DWAL, Figure 8c), HTMFS fails to scale after 12 threads. NOVA scales best in this workload, thanks to its per-core allocator. The performance gap between NOVA and HTMFS mainly comes from the different write instructions they use. NOVA uses non-temporal write (NT-write) instructions to store data, which bypass the cache and directly write to PM. It only occupies the write bandwidth of the PM. In contrast, HTMFS uses normal write instructions to store data. In case of a cache miss, HTMFS first reads the data into a cache line, then writes to the cache line, occupying both read and write bandwidth of the PM. However, the reads and writes to the PM interfere with each other, causing a decline in the total bandwidth [81]. Therefore, in DWAL, where most writes miss the cache, HTMFS has lower throughput than NOVA. We replace the write instructions in ZoFS and HTMFS with non-temporal ones and name them ZoFS-NT and HTMFS-NT. They show similar good scalability as NOVA. However, the performance begins to degrade after four threads because ZoFS and HTMFS have reached the upper limit of the PM write bandwidth, which keeps decreasing as the number of threads increases [34, 81].

For data read workloads, when different threads read a block in their respective private file (DRBL, Figure 8d), all file systems scale nearly linearly. Reading a private block in the shared file (DRBM) and reading the same block (DRBH) show similar performance, so these results are not shown here.

For metadata creation workloads, Figure 8e shows the performance when different threads create files in different directories (MWCL). HTMFS and ZoFS stop to scale after eight threads. This is because HTMFS and ZoFS are bounded by the limited PM write bandwidth resource.

However, NOVA performs better than HTMFS with less PM write bandwidth. The reason is that ZoFS uses zero indexes to indicate a non-exist page (as shown in Figure 3). It needs to initialize the file index to zero when creating a file, occupying significant PM bandwidth. ZoFS cannot use file size to indicate whether a page exists in a file because when a crash happens before an update operation completes, the file size will be inconsistent with the file index after reboot. However, the file size and the file index are consistent at any time in HTMFS, which makes it feasible for HTMFS to remove the



(a) Data overwrite, low contention (b) Data overwrite, medium contention (c) Data append, low contention

(d) Data read, low contention (e) Metadata create, low contention (f) Metadata create, medium contention

(g) Metadata rename, low contention (h) Metadata rename, medium contention

Figure 8: Results of FxMark workloads. HTMFS outperformaces ZoFS in DWOM(8b), MWCM(8f), MWRM(8h) and achieves similar performance in most cases. The worse performance of HTMFS (compared with ZoFS, e.g., DWOL(8a)) mainly comes from the gap between a certain instruction we replace.

file index initialization when creating a file. After removing the zero operation from HTMFS and ZoFS, the scalability becomes better than NOVA, as shown by HTMFS-NZ (No Zero) and ZoFS-NZ.

When creating files in a shared directory (MWCM, Figure 8f), HTMFS still scales well while other file systems exhibit poor scalability as the number of threads increases. The good scalability of HTMFS mainly comes from our lock strategy. Instead of locking the parent directory before every create operation, we only need hash table related lock, which avoids a lot competition. HTMFS-NZ performs better than HTMFS because it removes unnecessary memset from code path. It achieves maximum throughput when using eight threads, and degrades as threads increases [34, 81].

For metadata rename workloads, when different threads rename files in different directories (MWRL, Figure 8g), all file systems scale nearly linearly and HTMFS performs best among them. When moving files into a shared directory (MWRM, Figure 8h), like MWCM, HTMFS performs best among them. Thanks to the fine-grained concurrency control provided by HOP, HTMFS outperforms ZoFS by up to 6×.

Abort rate. Since we will retry for up to 60 times, an operation may trigger abort up to 60 times, being counted as 60 aborts. The average abort count is calculated by dividing the number of aborts by the total number of operations completed. After an operation has failed for all the 60 times, it will give up and walk the fallback path. We count the number of times the fallback path is executed and obtain the fallback rate as shown in Table 2. In the several tests both the number of aborts and the number of fallback path executions are negligible.

Table 2: Abort rate. The abort operation accounts for a small proportion of total operations, as well as the fallback path.

Operation	Average Abort Count	Fallback Rate
DWAL-8threads	0.002	0%
DWOL-1thread	0	0%
Varmail-1thread	0.004	0%
Varmail-28threads	0.303	0.17%
TPC-C SQLite	0.001	0%

The latency of the fallback path. For the write operation, we evaluate the operation latency of the normal path (RTM succeeds), the fallback path (RTM fails), and the journal-based path. The results are shown in Table 3. Although the scalability of the normal path is better than the fallback path, their latency is about the same. For writing 4KB files, the latency

Table 3: Operation latency. Each path is independent. e.g., Fallback path latency does not contain that of running a normal path. The journal-based path is slower than others because it requires writing more additional data.

Latency/cycles	Write (4KB)	Mkdir	Rename
Normal path	620.77	12360.67	3378.03
Fallback path	654.47	12486.87	3390.23
Journal-based path	4924.00	13627.00	3548.53
CoW path (NT write)	2293.00	/	/

Table 4: The throughput under high abort rate. We write DWOH that multiple threads write to a shared block in a shared file. HTMFS can fallback to lock rapidly to avoid dramatic performance drops.

Throughput (Kops/s)/#Thread	1	8	28
HTMFS	3732	1309	1111
ZoFS	6148	1241	983
NOVA-CoW	517	416	408
NOVA-relax	1036	1026	992
Libnvmmio	520	416	413

of the journal-based path should be twice the normal path theoretically. However, it is much higher than the theoretical value since the normal path writes are not all written to the PM (reside in the cache). To verify that, we add the latency of the CoW path (using non-temporal write, where the writes fall into the PM directly). The latency of the journal-based path is slightly higher than twice that of the CoW path because the former needs to record some metadata updates.

The latency difference between the different paths is not significant for other metadata operations. The journal-based path requires logging metadata updates, so the latency is slightly higher than the others.

The performance under high abort rate. We design a workload with strong competition for fxmark that multiple threads write to a shared block in a shared file to evaluate how HTMFS's fallback path performs. As shown in Table 4, HTMFS is able to fall back to locks quickly. As the number of threads increases, HTMFS's performance becomes better than ZoFS. The performance of HTMFS is weaker than ZoFS with a single thread, the reason of which is still because we use a slower memcpy to avoid RTM abort.

5.3 Macro-benchmarks

We select two filebench [72] workloads to evaluate the performance of HTMFS. Table 5 summarizes the characteristics of these workloads and the results are shown in Figure 9. We can observe that HTMFS performs well in all chosen workloads.

Webproxy is a read-dominated workload, HTMFS achieves similar performance with ZoFS and shows slightly higher throughput than NOVA and Libnvmmio.

Varmail emulates an email server with a large number of small files and involves both read and write operations. HTMFS is a good fit for this workload as Varmail involves more metadata operations. Besides, NOVA and Libnvmmio also show good scalability.

Table 5: Filebench workload characteristics.

Workload	# Files	Dir Width	File Size	R/W Ratio
Webproxy	10,000	1,000,000	16KB	5:1
Varmail	1,000	1,000,000	16KB	1:1

In both workloads, SplitFS is also tested but not shown here as it fails to scale after 8 threads and not outperforms HTMFS.

5.4 Crash Consistency Correctness

Correctness is difficult to be proven without formal verification. To show the crash consistency correctness of HTMFS, we design a simple experiment to show the difference between HTMFS and ZoFS (a weak crash consistency file system).

We first create a 4KB-file filled with character 'a'. Then we open it and write 8KB 'b' into it with a file system call `write(fd, data, 8192)`. In this process, the file system 1) first allocates a new free page as the second page (4KB–8KB), 2) overwrites the first page (0–4KB) with 'b', 3) fills the allocated page with 'b', 4) then links it to the file data index, 5) and finally updates the file size from 4KB to 8KB and updates both ctime and mtime in the file's metadata to the current time.

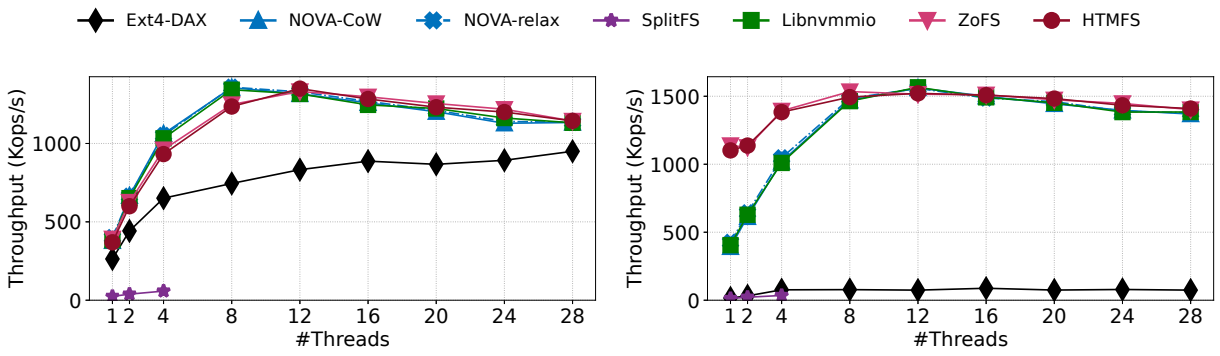
We inject several system crashes during the file system call `write(fd, data, 8192)` and then check some characteristics after rebooting the file system. As all PM writes in the RTM are guaranteed to be persisted atomically [66], rather than injecting crash in the RTM, we insert crash begin/after the RTM. The results are shown in Table 6.

The first two rows show the characteristic of consistent (all-or-nothing) states, respectively. If no change is applied, we should see 4KB "a" in the file, the length of the freelist being 249 (measured in the experiment), the ctime and mtime both unchanged. If the whole operation is finished, the file size, the content, the length of freelist (which should be 248 since a new page will be allocated) and the ctime and mtime should be updated altogether.

Row 3–8 (from ZoFS-1 to HTMFS-3) show the characteristic when ZoFS and HTMFS crash at different points. For every crash point, ZoFS has some difference with both consistent states. For example, at crash point 1, ZoFS is inconsistent for its freelist is reduced by 1, which means there is a persistent memory leak in ZoFS. At the same time, HTMFS is consistent with "nothing" or "all" state, proving HTMFS has stronger crash consistency than ZoFS.

5.5 Application Benchmarks

TPCC on SQLite. SQlite is a widely used lightweight yet full-featured SQL database engine. We drive SQLite with TPC-



(a) Webproxy (b) Varmail

Figure 9: Filebench. HTMFS achieves similar throughput as ZoFS in these workloads.

Table 6: Crash consistency states of ZoFS and HTMFS. We insert three crash points when writing 8KB 'b' into a file of 4KB 'a'. HTMFS is consistent with all-or-nothing states after crash, while ZoFS fail to restore consistent state.

Crash Point	File Size	Content[0]	Len(freelist)	Ctime&Mtime
Nothing	4KB	'a'	249	Not changed
All	8KB	'b'	248	Changed
ZoFS-1	4KB	'a'	248	Not changed
HTMFS-1	4KB	'a'	249	Not changed
ZoFS-2	4KB	'b'	248	Not changed
HTMFS-2	4KB	'a'	249	Not changed
ZoFS-3	8KB	'b'	248	Not changed
HTMFS-3	8KB	'b'	248	Changed

Table 7: TPC-C transaction mix.

Transaction	NEW	PAY	OS	DLY	SL
Ratio	44%	44%	4%	4%	4%

C [15], which is an online transaction processing benchmark that simulates an order processing application.

TPC-C involves five types of transactions: New-Order (NEW), Payment (PAY), Order-Status (OS), Delivery (DLY), and Stock-Level (SL). We use the mixed workload in the experiment and run it with a single thread. Table 7 gives the ratio of different transactions.

Figure 10 summarizes the throughput of different file systems. HTMFS achieves the second highest throughput, which is 2% lower than ZoFS. While NOVA-CoW is 67% slower than NOVA. This demonstrates the low overhead of HTMFS in achieving strong consistency.

LevelDB. LevelDB [24] is a key-value storage library developed by Google. We use LevelDB's `db_bench` benchmarks to prove that we can achieve strong consistency with little overhead. SplitFS and NOVA provide both strong and weak consistent modes. However, we cannot run this benchmark on SplitFS, so we only compare HTMFS with NOVA.

For the read operations, NOVA-CoW and NOVA-relax perform almost the same. For the update operations (fill, overwirte, and delete), NOVA-CoW is obviously slower than NOVA-relax, while HTMFS always performs as well as ZoFS.

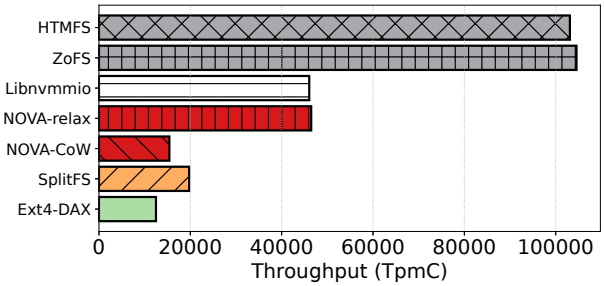


Figure 10: TPC-C SQLite. HTMFS provides stronger consistency with acceptable performance reduction compared to ZoFS, while NOVA sacrifices much more to get the same consistency.

Table 8: Latency of LevelDB. HTMFS and ZoFS perform almost identically, while we can observe a clear latency gap between NOVA-CoW and NOVA-relax. This indicates that HTMFS efficiently achieves data consistency guarantees.

Latency/μs	NOVA-CoW	NOVA-relax	ZoFS	HTMFS
Fill sync.	6.605	5.262	3.190	3.134
Fill seq.	4.605	3.284	2.071	2.039
Fill rand.	31.528	25.142	24.125	24.313
Overwrite.	39.662	31.641	42.128	42.207
Read seq.	1.020	1.004	2.111	2.136
Read rand.	7.357	7.029	11.027	10.600
Read hot.	1.373	1.373	1.289	1.281
Delete rand.	3.169	2.120	1.335	1.281

6 Discussion

6.1 Other File System Features

Previous sections mainly focus on the common file system interfaces, like read/write. We suggest that our design can be further combined with other features.

Compression Some file systems support data compression features to reduce space on storage devices. The compression procedure can be viewed as normal read (read the data and apply the compression algorithm) plus write (write the compressed data), which falls into the scope of our HOP design.

Deduplication Deduplication features the ability to reduce redundancy in stored data to reclaim disk space. It involves scanning all data at intervals to find duplicate blocks and remove them. The scanning part does not introduce any conflicts, and the removing part is no different from common file operations, which can also be handled by the HOP design to ensure consistency.

Checksumming/Encrypting Checksumming and Encrypting features are supported for error-detection and security considerations, which works by checksumming/encrypting the data before write operations and verifying the checksum during read operations. This procedure can be easily wrapped in the original read and write operations protected by HOP.

To summarize, these advanced features are orthogonal to our work and can be implemented in further works.

6.2 HOP in Key-Value Stores

Since key-value stores have a fixed access interface (e.g. `put/get/scan`) like the file system, it is relatively easy to use HOP for key-value stores. Like applying HOP to the file system, when using it for key-value stores, we need to consider how each API needs to be modified to reduce the transaction size.

7 Related Work

To our best knowledge, no prior work has discussed using HTM to improve the performance of strong consistent file systems. We discuss related work in this section.

Persistent Transactions. As PM adds durability to memory, researchers study how to facilitate the PM programming via transaction semantics. Some of these studies [9, 12, 14, 23, 25, 27, 38–40, 44, 45, 48, 49, 56, 61, 62, 75, 82, 83] use software approaches, such as undo and/or redo logs, to guarantee transaction semantics on PM; while the others [3, 4, 8, 21, 33, 35, 36, 44, 54, 63, 71] leverage modified hardware mechanisms. All these existing persistent transaction systems targets on user-space applications or data structures, while our work focuses on using HTM in PM file systems. Compared with the data structures, the file systems put extra challenges due to the FS's inevitable large memory footprint and complex operations.

Before eADR [29] is available, many HTM implementations or modifications are proposed to facilitate PM with HTM [3, 4, 22, 35, 60, 77]. The design of HTMFS is orthogonal to these HTM hardware implementation. Furthermore, HTMFS can be simplified if transaction suspend and resume are supported on the platform, as what is planned in the next-generation Intel's server platform [30].

System Transactions and Transactional FS. Researchers have studied to use transactions in an operating system [58, 59]. TxLinux [65] is the first operating system that leverages MetaTM [60], an interrupt-compatible HTM model, in a cooperative synchronization approach that combines HTM with software locks. TxOS [57] proposes and implements system transactions to provide system-wide transactional support. A transactional Ext3 is implemented in TxOS.

A set of file systems provide transactional APIs to applications so that multiple file operations could be finished in an ACID transaction. Examples include Microsoft TxF [2,50], Inversion [55], OdeFS [20], DBFS [53], TFFS [19], Stasis [67], Amino [78], Valor [69], CFS [51], and TxFS [28]. Unlike these prior studies, our work focuses on leveraging HTM to enforce the performance and strong consistency within each single file system operation. We think it possible to extend HTMFS to implement cross-operation transactions and we leave further exploration as future work.

HTM-assisted OCC. Several prior work has explored to combine HTM with OCC-like mechanisms. DBX [76] first use an OCC-like mechanism to address the limited working set of HTM. Leis et al. [43] proposes to split a database transaction into small pieces, each of which is protected by an HTM transaction. These pieces are then glued together via timestamps to guarantee the atomicity of the whole database transaction. HTCC [79] combines fine-grained locks and HTM-assisted OCC. It uses HTM-assisted OCC only on cold data to reduce the database transaction abort rates and leverages delta-restoration to minimize the overhead of transaction restarts. In contrast to this work that focuses on concurrent consistency of database transactions, our work focuses on providing both concurrent consistency and crash consistency with a combination of HTM and FS-aware OCC-like mechanism.

Page fault in RTM. PfTouch [73] efficiently solves the problem of RTM abort due to page fault by modifying the RTM hardware to recognize page fault and triggering page fault in the abort handler. With RTM hardware support, HTMFS can use these methods to reduce the performance loss due to page fault abort.

8 Conclusion

We provide HTMFS, the first HTM-based PM file system. HTMFS provides strong crash consistency and fine-grained concurrency control with HTM support. Evaluation shows that the performance of HTMFS is as good as file systems that only provide weak crash consistency guarantees while providing strong consistency guarantees. In some competitive scenarios, the performance improvements are significant.

Acknowledgements

We sincerely thank our shepherd Changwoo Min, the anonymous reviewers, Nian Liu, Jingwei Xu, and Qing Wang for their constructive comments and insightful suggestions. This work is supported in part by the National Natural Science Foundation of China (No. 61925206), the High-Tech Support Program from Shanghai Committee of Science and Technology (No. 19511121100), and Huawei. Mingkai Dong (mingkaidong@sjtu.edu.cn) is the corresponding author.

References

[1] *Intel® 64 and IA-32 Architectures Software Developer's Manual*, volume 1, chapter 16. Intel, 2021.

[2] Christian Allred. Understanding windows file system transactions. https://www.snia.org/sites/default/orig/sdc_archives/2009_presentations/tuesday/ChristianAllred_UnderstandingWindowsFileSystemTransactions.pdf, 2009.

[3] Hillel Avni and Trevor Brown. Persistent hybrid transactional memory for databases. *Proc. VLDB Endow.*, 10(4):409–420, November 2016.

[4] Hillel Avni, Eliezer Levy, and Avi Mendelson. Hardware transactions in nonvolatile memory. In *Proceedings of the 29th International Symposium on Distributed Computing - Volume 9363*, DISC 2015, pages 617–630, Berlin, Heidelberg, 2015. Springer-Verlag.

[5] Rudolf Bayer. Symmetric binary b-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1:290–306, 1972.

[6] S. Best. Jfs overview.

[7] Rev C, Dave Hitz, James Lau, and Michael Malcolm. File system design for an nfs file server appliance. 10 2000.

[8] Daniel Castro, Paolo Romano, and João Barreto. Hardware transactional memory meets memory persistency. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 368–377, 2018.

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

[10] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Optimistic crash consistency. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 228–243, New York, NY, USA, 2013. Association for Computing Machinery.

[11] Jungsik Choi, Jaewan Hong, Youngjin Kwon, and Hwansoo Han. Libnvmmio: Reconstructing software IO path with failure-atomic memory-mapped interface. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 1–16, 2020.

[12] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven 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 XVI, pages 105–118, New York, NY, USA, 2011. Association for Computing Machinery.

[13] 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*, pages 133–146, 2009.

[14] Andreia Correia, Pascal Felber, and Pedro Ramalhete. Romulus: Efficient algorithms for persistent transactional memory. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, SPAA '18, pages 271–282, New York, NY, USA, 2018. Association for Computing Machinery.

[15] The Transaction Processing Council. Tpc-c benchmark v5.11. 2021.

[16] Mingkai Dong, Heng Bu, Jifei Yi, Benchao Dong, and Haibo Chen. Performance and protection in the zofs user-space nvm file system. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 478–493, 2019.

[17] Mingkai Dong and Haibo Chen. Soft updates made simple and fast on non-volatile memory. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '17, page 719–731. USENIX Association, 2017.

[18] 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*, pages 1–15, 2014.

[19] Eran Gal and Sivan Toledo. A transactional flash file system for microcontrollers. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, page 7, USA, 2005. USENIX Association.

[20] Narain H. Gehani, H. V. Jagadish, and William D. Roome. Odefs: A file system interface to an object-oriented database. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 249–260, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[21] Kaan Genç, Michael D. Bond, and Guoqing Harry Xu. Crafty: Efficient, htm-compatible persistent transactions. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, pages 59–74, New York, NY, USA, 2020. Association for Computing Machinery.

[22] Ellis Giles, Kshitij Doshi, and Peter Varman. Hardware transactional persistent memory. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS '18, pages 190–205, New York, NY, USA, 2018. Association for Computing Machinery.

[23] Ellis R. Giles, Kshitij Doshi, and Peter Varman. Softwrap: A lightweight framework for transactional support of storage class memory. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–14, 2015.

[24] Google. Leveldb. https://github.com/google/leveldb, 2021.

[25] Jinyu Gu, Qianqian Yu, Xiayang Wang, Zhaoguo Wang, Binyu Zang, Haibing Guan, and Haibo Chen. Pisces: A scalable and efficient persistent transactional memory. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, pages 913–928, USA, 2019. USENIX Association.

[26] Valerie Henson. The many faces of fsck. https://lwn.net/Articles/248180/, Sep. 2007.

[27] Qingda Hu, Jinglei Ren, Anirudh Badam, Jiwu Shu, and Thomas Moscibroda. Log-structured non-volatile main memory. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '17, pages 703–717, USA, 2017. USENIX Association.

[28] Yige Hu, Zhiting Zhu, Ian Neal, Youngjin Kwon, Tianyu Cheng, Vijay Chidambaram, and Emmett Witchel. Txfs: Leveraging file-system crash consistency to provide acid transactions. *ACM Trans. Storage*, 15(2), May 2019.

[29] Intel. eadr: New opportunities for persistent memory applications. https://software.intel.com/content/www/us/en/develop/articles/eadr-new-opportunities-for-persistent-memory-applications.html, Jan. 2021.

[30] Intel. Intel® architecture instruction set extensions and future features programming reference. https://software.intel.com/content/dam/develop/external/us/en/documents-tps/architecture-instruction-set-extensions-programming-reference.pdf, May 2021.

[31] Intel. Intel® transactional synchronization extensions (intel® tsx) programming considerations. https://software.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/compiler-reference/intrinsics/intrinsics-for-intel-advanced-vector-extensions-2/intrinsics-for-intel-transactional-synchronization-extensions-intel-tsx/intel-transactional-synchronization-extensions-intel-tsx-programming-considerations.html, June 2021.

[32] Intel. Restricted transactional memory overview. https://software.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/compiler-reference/intrinsics/intrinsics-for-intel-advanced-vector-extensions-2/intrinsics-for-intel-transactional-synchronization-extensions-intel-tsx/intrinsics-for-restricted-transactional-memory-operations/restricted-transactional-memory-overview.html, 2021.

[33] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. Failure-atomic persistent memory updates via justdo logging. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 427–442, New York, NY, USA, 2016. Association for Computing Machinery.

[34] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic performance measurements of the intel optane dc persistent memory module, 2019.

[35] Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. Dhtm: Durable hardware transactional memory. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, pages 452–465. IEEE Press, 2018.

[36] Arpit Joshi, Vijay Nagarajan, Stratis Viglas, and Marcelo Cintra. Atom: Atomic durability in non-volatile memory through hardware logging. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 361–372, 2017.

[37] 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*, pages 494–508, 2019.

[38] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. High-performance transactions for persistent memories. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 399–411, New York, NY, USA, 2016. Association for Computing Machinery.

[39] R Madhava Krishnan, Jaeho Kim, Ajit Mathew, Xinwei Fu, Anthony Demeri, Changwoo Min, and Sudarsun Kannan. Durable transactional memory can scale with timestone. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 335–349, 2020.

[40] R. Madhava Krishnan, Wook-Hee Kim, Xinwei Fu, Sumit Kumar Monga, Hee Won Lee, Minsung Jang, Ajit Mathew, and Changwoo Min. TIPS: Making volatile index structures persistent with DRAM-NVMM tiering. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 773–787. USENIX Association, July 2021.

[41] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, June 1981.

[42] 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*, pages 460–477, 2017.

[43] Viktor Leis, Alfons Kemper, and Thomas Neumann. Exploiting hardware transactional memory in main-memory databases. In *2014 IEEE 30th International Conference on Data Engineering*, pages 580–591, 2014.

[44] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. Dudetm: Building durable transactions with decoupling for persistent memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 329–343, New York, NY, USA, 2017. Association for Computing Machinery.

[45] Youyou Lu, Jiwu Shu, and Long Sun. Blurred persistence in transactional persistent memory. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–13, 2015.

[46] Berenice Mann. New technologies for the arm a-profile architecture. https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/new-technologies-for-the-arm-a-profile-architecture, April 2019.

[47] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for unix. *ACM Trans. Comput. Syst.*, 2(3):181–197, August 1984.

[48] Amirsaman Memaripour, Anirudh Badam, Amar Phanishayee, Yanqi Zhou, Ramnatthan Alagappan, Karin Strauss, and Steven 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. Association for Computing Machinery.

[49] Amirsaman Memaripour, Joseph Izraelevitz, and Steven Swanson. Pronto: Easy and fast persistence for volatile data structures. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, pages 789–806, New York, NY, USA, 2020. Association for Computing Machinery.

[50] Microsoft. Transactional ntfs (txf). https://docs.microsoft.com/en-us/windows/win32/fileio/transactional-ntfs-portal, May 2018.

[51] Changwoo Min, Woon-Hak Kang, Taesoo Kim, Sang-Won Lee, and Young Ik Eom. Lightweight Application-Level crash consistency on transactional flash storage. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 221–234, Santa Clara, CA, July 2015. USENIX Association.

[52] 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, 2016.

[53] Nicholas Murphy, Mark Tonkelowitz, and Mike Vernal. The design and implementation of the database file system. 2002.

[54] Matheus Almeida Ogleari, Ethan L. Miller, and Jishen Zhao. Steal but no force: Efficient hardware undo+redo logging for persistent memory systems. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 336–349, 2018.

[55] Michael A. Olson. The design and implementation of the inversion file system. In *USENIX Winter 1993 Conference (USENIX Winter 1993 Conference)*, San Diego, CA, January 1993. USENIX Association.

[56] pmem.io. Persistent memory development kit. https://pmem.io/pmdk/.

[57] Donald E. Porter, Owen S. Hofmann, Christopher J. Rossbach, Alexander Benn, and Emmett Witchel. Operating system transactions. In *Proceedings of the ACM*

SIGOPS 22nd Symposium on Operating Systems Principles, SOSP '09, pages 161–176, New York, NY, USA, 2009. Association for Computing Machinery.

[58] Donald E. Porter and Emmett Witchel. Understanding transactional memory performance. In *2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*, pages 97–108, 2010.

[59] Hany E. Ramadan, C. Rossbach, and E. Witchel. The linux kernel: A challenging workload for transactional memory. 2006.

[60] Hany E. Ramadan, Christopher J. Rossbach, Donald E. Porter, Owen S. Hofmann, Aditya Bhandari, and Emmett Witchel. Metatm/txlinux: Transactional memory for an operating system. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 92–103, New York, NY, USA, 2007. Association for Computing Machinery.

[61] Pedro Ramalhete, Andreia Correia, and Pascal Felber. Efficient algorithms for persistent transactional memory. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '21, pages 1–15, New York, NY, USA, 2021. Association for Computing Machinery.

[62] Pedro Ramalhete, Andreia Correia, Pascal Felber, and Nachshon Cohen. Onefile: A wait-free persistent transactional memory. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 151–163, 2019.

[63] Jinglei Ren, Jishen Zhao, Samira Khan, Jongmoo Choi, Yongwei Wu, and Onur Mutlu. Thynvm: Enabling software-transparent crash consistency in persistent memory systems. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, pages 672–685, New York, NY, USA, 2015. Association for Computing Machinery.

[64] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, feb 1992.

[65] Christopher J. Rossbach, Owen S. Hofmann, Donald E. Porter, Hany E. Ramadan, Bhandari Aditya, and Emmett Witchel. Txlinux: Using and managing hardware transactional memory in an operating system. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 87–102, New York, NY, USA, 2007. Association for Computing Machinery.

[66] Andy Rudoff. Questions about eadr, sfence and intel tsx. https://groups.google.com/g/pmem/c/_DJCFGylfVE/m/L0oyltg8BAAJ, 2020.

[67] Russell Sears and Eric Brewer. Stasis: Flexible transactional storage. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 29–44, USA, 2006. USENIX Association.

[68] Dennis Shasha, Francois Llirbat, Eric Simon, and Patrick Valduriez. Transaction chopping: Algorithms and performance studies. *ACM Trans. Database Syst.*, 20(3):325–363, sep 1995.

[69] Richard P. Spillane, Sachin Gaikwad, Manjunath Chinni, Erez Zadok, and Charles P. Wright. Enabling transactional file access via lightweight kernel extensions. In *Proccedings of the 7th Conference on File and Storage Technologies*, FAST '09, pages 29–42, USA, 2009. USENIX Association.

[70] SQLite. Sqlite transactional sql database engine. https://www.sqlite.org/, 2021.

[71] Long Sun, Youyou Lu, and Jiwu Shu. Dp2: Reducing transaction overhead with differential and dual persistency in persistent memory. In *Proceedings of the 12th ACM International Conference on Computing Frontiers*, CF '15, New York, NY, USA, 2015. Association for Computing Machinery.

[72] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A flexible framework for file system benchmarking. *USENIX; login*, 41(1):6–12, 2016.

[73] Rubén Titos-Gil, Ricardo Fernández-Pascual, Alberto Ros, and Manuel E Acacio. Pftouch: Concurrent page-fault handling for intel restricted transactional memory. *Journal of Parallel and Distributed Computing*, 145:111–123, 2020.

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

[75] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight persistent memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 91–104, New York, NY, USA, 2011. Association for Computing Machinery.

[76] Zhaoguo Wang, Hao Qian, Jinyang Li, and Haibo Chen. Using restricted transactional memory to build a scalable in-memory database. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, New York, NY, USA, 2014. Association for Computing Machinery.

[77] Zhaoguo Wang, Han Yi, Ran Liu, Mingkai Dong, and Haibo Chen. Persistent transactional memory. *IEEE Comput. Archit. Lett.*, 14(1):58–61, January 2015.

[78] Charles P. Wright, Richard Spillane, Gopalan Sivathanu, and Erez Zadok. Extending acid semantics to the file system. *ACM Trans. Storage*, 3(2):4–es, June 2007.

[79] Yingjun Wu and Kian-Lee Tan. Scalable in-memory transaction processing with htm. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '16, pages 365–377, USA, 2016. USENIX Association.

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

[81] Jifei Yi, Benchao Dong, Mingkai Dong, Ruizhe Tong, and Haibo Chen. MT^2: Memory bandwidth regulation on hybrid NVM/DRAM platforms. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, Santa Clara, CA, February 2022. USENIX Association.

[82] Pantea Zardoshti, Tingzhe Zhou, Yujie Liu, and Michael Spear. Optimizing persistent memory transactions. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 219–231, 2019.

[83] Lu Zhang and Steven Swanson. Pangolin: A fault-tolerant persistent memory programming library. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, pages 897–911, USA, 2019. USENIX Association.