A Highly Scalable and Efficient File System For Non-Volatile Memories

Fan Yang [1,3,*], Junbin Kang [2,*], Shuai Ma [1,3], Jinpeng Huai [1,3]

[1]SKLSDE Lab, Beihang University, China
[2]Tencent, China
[3] Beijing Advanced Innovation Center for Big Data and Brain Computing, Beijing, China

Abstract

With the rapid development of fast and byte addressable non-volatile memory (NVM) technologies, hybrid NVM/DRAM storage systems will soon be applied to the computer system. Existing NVM-based file systems have been highly optimized around the NVM properties such as byte-addressability. However, they still inherit some design choices of block-oriented storage such as scalability bottlenecks and data copying overhead for ensuring data consistency.

We present PLFS, a highly scalable, efficient and consistent file system for non-volatile memories. PLFS is designed to achieve high performance through a bundle of novel techniques: (1) a persistent, scalable and light-weight naming layer that substitutes the cumbersome VFS layer, (2) a fine-grained byte-unit file index that avoids the copy-on-write overhead while providing data consistency guarantee, (3) a distributed and light-weight journaling, which reduces the journaling overhead as well as contention on many-core platforms, (4) a light-weight *atomic-msync* mechanism, which supports atomic user-space data persisting with significant lower overhead compared to state-of-the-art file systems which provide the same atomic primitive.

Experimental results show that PLFS performs much better than the state-of-the-art file systems and achieves near-linear scalability on a 40-core machine.

1 Introduction

Emerging non-volatile memories (NVMs) such as Phase Change Memories (PCM) and memristors provide high performance comparable to DRAM and support fast accesses through memory bus. The advent of the fast non-volatile memory technologies are expected to radically revolutionize the landscape of existing storage systems and deliver a high level of parallelism and extremely low latency (nanosecond-level) for data accessing.

Designing a highly scalable and efficient file system for emerging non-volatile memories faces several challenges, and it requires the file system to fully exploit the characteristics of NVM such as byte-addressability and persistence. Meanwhile, it would certainly introduce some loss in performance if the file system provides the useful data consistency to applications.

To utilize the high performance offered by NVMs [5, 6, 13, 15, 17], existing NVM-aware file systems (NVMFS) sharp the traditional block-oriented file system designs around the advanced properties of NVMs. However, they still suffer from the following drawbacks.

Firstly, most of existing NVMFS improve the underlying file system performance within the confinement of the traditional I/O stack architecture. That is, they still need the traditional Virtual File System (VFS) layer, which is designed to abstract the underlying on-disk objects, and to offer in-memory cache, hierarchy naming and access protection, which introduces unnecessary costs and scalability bottlenecks when allocating, filling and freeing DRAM objects (e.g., dentry and inode) on fast NVMs. The VFS layer also has many notorious scaling bottlenecks on many-core platforms due to the use of global data structures [?].

Secondly, existing NVMFS usually employ copy-on-write (COW) and data journaling to provide data consistency guarantees, which comes at a price of double data writes. They adopt traditional block-based file data indexing. As a result, overwrites and unaligned appends still need to copy out the old data for COW although block-aligned appends only need one single data write.

Finally, although the recent proposed NOVA system [17] supports *atomic-mmap*, it needs to write back the entire mmaped area to maintain data consistency even if only a small portion is modified.

To address all the above issues, we propose the *Persistent memory Light-weight File System* (PLFS) in this s-

[*]Corresponding authors: Fan Yang, Junbin Kang

tudy. PLFS designs the light-weight naming layer named LVFS to integrate the traditional VFS with the namespace of underling file systems. By combining the naming functionality of the VFS and the underlying file system, LVFS merges double lookups launched by VFS and the underlying file system into one when looking up a file. By doing like this, LVFS also eliminates duplicated DRAM data structures in both VFS and the underling file system, and enables careful crafting of the DRAM data structures such as separating the global locks that cause scaling bottlenecks. In addition, LVFS introduces light-weight per-cpu meta-journal to guarantee the crash consistency and scalability.

As NVMs are byte addressable, PLFS adopts a fined-grained byte-unit file index to manage the file data, instead of the block-unit one adopted by others. Unlike the traditional block-unit extent management that writes data in a block aligned way, the byte-unit index allows to manage the extent in byte range, and write data starting from any position of a block, which enables PLFS to write application data without the need to block aligning the write and without the need to copying out the old data before performing block-unaligned write. Specifically, the traditional COW needs to copy the modification uncovered parts of a block out to the new allocated space, while PLFS just splits one extent to multiple extents recording the starting position and the cover size of data within these relevant updated blocks.

Meanwhile, PLFS uses light-weight per-inode journaling to achieve the atomicity of modifying the index tree, which provides the strong consistency, and maintains the efficiency and scalability as much as possible.

Further, in order to recognize the modified mmaped-data and to just write the dirty data back when persisting data by *msync*, PLFS tracks the process of page faults and records the modified pages.

The contributions of this paper are as follows:
(1) To fully exploiting the NVMs performance, we redesigns a lightweight naming layer *LVFS* to substitute the traditional VFS layer.
(2) We achieve *Non-Copy-On-Write* (NCOW) by using a fined-grained index to eliminate copies while keeping the data consistency.
(3) We propose the light-weight *atomic-msync* by tracking the dirty pages to eliminate unnecessary write backs.
(4) We finally implement a scalable and efficient file system PLFS, and experimentally demonstrate that it outperforms the existing NVM-based file systems ??? with references.

2 Background and Motivation

Existing file systems, even optimized for NVMs, are far from fully exploiting the high performance offered by NVMs. In this section, we highlight the inefficiencies of the existing file systems on NVMs.

VFS becomes cumbersome on NVMs. The VFS layer offers naming and in-memory cache for on-disk file system objects, which is designed around the assumption that the underlying storage device provides slow and block-aligned data accesses. However, certain design decisions for slow storage devices become the obstacles of exploiting the high performance of NVMs.

As the role of VFS within the I/O stack is to offer in-memory cache for on-disk file systems, cache-missed lookups cause extra lookups within the underlying file system such as the pathname lookup when creating a new file. For the on-disk lookup is slow, the double lookup problem has little impacts on the performance on disks. However, the overhead becomes significant on NVMs that provide near-DRAM performance.

In addition, VFS adopts some global locks to protect the concurrent insertions and removals such as rename_lock [8], which could incur the scalability bottlenecks on many-core systems.

Data consistency is the performance killer. In additional to metadata consistency, offering data consistency is a key aspect of modern file systems. However, providing data consistency comes at the price of double data writes when adopting data journaling and copy-on-write.

For journaling, all updates have to be written twice: one to journal, and the other to file. When the update is larger than a COW block size, then journaling becomes costly because of the heavy double copying overhead.

For COW, if the application appends data to a file whose size is not exactly an integral multiple of the block size, the file system needs to firstly copy the old data in the last block out to the new allocated space, and then appends new data. If the application overwrites part of the file data, the file system needs to copy all the old data in the related blocks out to the new allocated space, and then modifies the copied data.

File system journaling is costly on NVMs. Modern file systems usually adopt journaling to maintain metadata consistency and even data consistency. Traditional journaling logs dirty data and/or metadata in the unit of blocks on block storage devices as they only provide block-aligned accesses.

As NVMs offer byte-addressable data access, emerging NVM-aware file systems such as PMFS [13] adopt fine-grained logging to remove unnecessary overhead during journaling. Nevertheless, this fine-grained logging approach still journals all the data/metadata fields to be modified to the log file before modifying the primary data/metadata on NVMs. The journaling overhead still grows with the size of the data/metadata to be modified.

On the other hand, in order to resolve the transaction dependencies between different transactions, current journaling file systems (e.g., Ext4 and XFS) usually adopts a centralized journaling design, which introduces scaling bottlenecks on many-core systems [9].

3 Design and Implementation

In order to address all the inefficiencies highlighted in Section 2, we propose PLFS, a highly scalable and efficient file system for NVMs on many-core. The key design of PLFS centers on LVFS, a light-weight naming layer that substitutes the cumbersome VFS, a Non-Copy-On-Write (NCOW) strategy realized by the novel fine-grained byte-unit file index structure which eliminates copying overhead for maintaining data consistency, and a light-weight *atomic-msync* providing atomic *mmap* and avoiding the unnecessary write back.

It should be noted that *pinode* is the only structure in LVFS which is organized by hash table and functionally like *inode* and *dentry* in VFS, *extent* is the leaf node of the file index tree in PLFS which manages the fragment of file data, and PLFS uses bitmap to display the usage of data space. There are some following design decisions to achieve above goals.

Remain memory-cache structures to ensure the highest efficiency. Although the read/write speed of NVM is comparable to DRAM, there are still a little gaps between them up till now, not to mention that the file system needs to look up and update the metadata frequently. Hence, keeping efficiency and consistency simultaneously would be difficult. Based on this, PLFS keeps the extend-copies of the NVM objects and the corresponding index structures in DRAM, and records the pointers to the NVM objects to ensure the highest efficiency.

For LVFS, PLFS builds the hash table to manage *pinodes* in DRAM. The hash table in DRAM is corresponding to the hash table in NVM while holding per-bucket locks to prevent concurrent modifications. The extend-copied *pinode* in DRAM points to the *pinode* in NVM and would be updated in-place without the consideration of consistency.

For file data management, PLFS keeps the file index tree and the extend *extent* in DRAM, only persists *extent* in NVM. Furthermore, PLFS adopts per-CPU link lists in DRAM to manage data space in NVM, just records the final space usage in bitmap after logging the file updates. What is worth mentioning is that the dram-structures enables the separation of *write* and *fsync*. And PLFS does not persist the modifications caused by the *write* operation until the invoking of *fsync*.

Propose a novel strategy to guarantee the data consistency. Although providing strong data consistency would incur extra overhead by copying and heavy metadata, it is still quit necessary for upper applications. Otherwise they need to implement their own complex and slow update protocols to ensure consistency of data on the file system [?]. PLFS proposes a set of techniques to guarantee the crash consistency and minimize the impact of the overhead. And the important one of them is Redirect-on-write (ROW), which keeps a cache of the NVM object pointers in DRAM, writes updates to new locations in NVM and only logs the modifications to the NVM object pointers before redirecting the pointers to new locations. The ROW strategy provides the atomic updates of NVM objects and reduces the log size which avoids the heavy overhead caused by journal cleaning and garbage collection.

Use distributed data structures to achieve the good scalability. In PLFS, nearly all of the data structures are distributed to achieve the high scalability. To be specific, the free *pinodes* and *extents* in PLFS are managed by link lists at each CPU in DRAM to allocate and free without contention. The data structures are protected by separated locks to avoid global locking. The metadata and data journals are organized by multiple log files to reduce scalability bottlenecks.

The key components of PLFS mainly include a light-weight, scalable and persistent naming layer called LVFS for the file system namespace management, a fine-grained byte-unit index tree for file data management and the light-weight and scalable metadata and data journalings for maintaining crash consistency.

3.1 LVFS

With the traditional I/O stack architecture, there are some in-memory cached structures like dentry designed for faster access on disk, which is unnecessary on fast NVMs and introduce extra overhead of their management, double lookup launched by VFS and the underlying file system, and scaling bottlenecks caused by global locks.

In addition, the NVM-based software usually uses *clflush* combined with *mfence* and *pcommit* to explicitly flush the data within the CPU cache to NVMs for ordering and durability [17, 13]. However, the flush overhead for persistence is expensive, and hence reducing the number of flush operations on NVMs is critical to achieving good performance [18].

To this end, LVFS is designed to substitute for VFS within the I/O stack for NVMs. Similar to VFS, LVFS offers naming and permission checking functionality. However, the most important difference from VFS is that LVFS also provides durability and crash consistency on NVMs, which is responsible for namespace management of PLFS. LVFS is crafted to pack multiple key data structures (e.g., *inode* and *dentry*) together as a single object and then to compact the object into fewer number of cache lines for better performance on NVMs. We call such object a node (*pinode*) in LVFS. Each *pinode* repre-

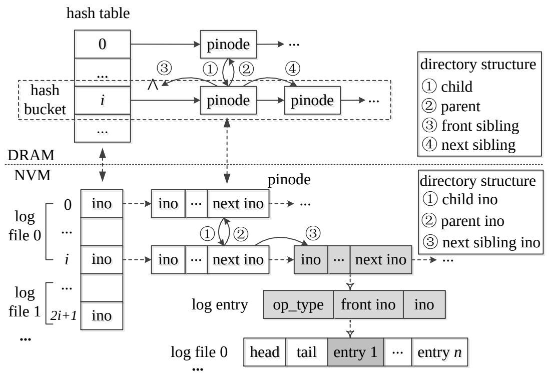


Figure 1: **The LVFS data structures and layout in N-VM.** *This figure shows*

sents a file system entity (i.e., file or directory) and stores the necessary metadata information such as the name, size and pointer to its file tree root (for file) or pointer to its first child nodes (for directory).

By combining the naming functionality of VFS and that of the underlying file system together through LVF-S, PLFS eliminates the double lookup performance problem so as to speed up the file system lookup operations, lowers the flush overhead for persistence, and enables separated locks to reduce contention.

3.1.1 The LVFS data structures

As shown in the Figure 1, LVFS adopts a global multi-buckets hash table to organize the namespace of PLFS like VFS. And each bucket links the *pinodes* in NVM with the same hash value. Not like the traditional I/O stack that the underling file system would build additional structures such as dentry tree to organize the directory, the hash table is the only structure in LVFS and we just record the directory information including pointers to parent, first child (for directory) and siblings to *pinode*. As the length and function of the hash table in LVF-S is same as VFS, the length of each bucket would not be too long to lookup.

To quickly rebuild the hash table in DRAM when restart or recovery, LVFS keeps a corresponding one in NVM. To minimize the write size on NVM, all of the pointers to *pinodes* are the 4-bytes-id of the *pinode*, which is calculated by the relative address on NVM.

When the application requires a directory operation, LVFS just lookup in DRAM. For example, when we do the pathname lookup, we first calculate the hash value of the name, and then traverse the corresponding hash bucket link list to find the exact *pinode* in DRAM according to the file name and its parent.

To protect the concurrent directory operations, LVFS adopts per-bucket locks of the hash table to guarantee the good scalability on many-core platforms. As pathname lookup is extremely frequent compared to insertion and removal, LVFS employs RCU lock for each bucket like VFS. For the bucket lookup, LVFS holds the rcu-lock. While for the insertion or removal, LVFS uses spinlock to ensure the correctness. Except that, for the operation involves multi-buckets such as rename, LVFS uses sequence-lock to prevent the special situation that one bucket removes the old *pinode* and the other one does not add the new *pinode* yet.

3.1.2 The LVFS crash consistency

The data structures of LVFS are persisted on NVMs and hence could survive system power down or crashes. One key challenge to PLFS is how to maintain crash consistency of these data structures while providing scalability and minimizing the overhead. LVFS adopts the distributed journals to scalably log the updates. As shown in Figure 1, LVFS evenly maps the hash buckets to multiple log files on NVM. Each log file is simply a reserved continuous NVM area used to log updates and maintains the head and tail of the log records within the start of the log file.

Logging For each log entry, it contains the operation type flag to distinguish *create*, *delete* and *rename*, as each operation has various handling and log entry numbers. As mentioned before, the log file records the updated *pinode* and the ROW strategy provides the crash consistency guarantee of the *pinode* in NVM. LVFS still needs to record the hash bucket link list and the directory link list to rebuild the relationship.Therefore, the log entry also records the front *pinode*. The front one could be the front *pinode* in bucket or the front sibling in directory. And the next *pinode* in hash bucket or in directory has been recorded in the *pinode* in NVM.

When adding a *pinode*, LVFS would insert it to the head of the link list. As the hash value and parent address are recorded in *pinode*, LVFS only logs the updated *pinode* without the front one. While for the removal, the deleted *pinode* could be anywhere in the link list, then both the front node in bucket and the front sibling in directory would be logged. Updates to each link list are first recorded in the log file, and then are applied to the bucket and directory persistently.

Journal mapping and Crash consistency At any time, each hash bucket can only be assigned to one single log file. And all the update records of each hash bucket should maintain the update order, as the update order is critical to resolving the dependencies between update records when recovering the hash table using the log files upon failure. For example, provided that there are already two nodes *A* and *B* linked into one single bucket in the given order, a node removal operation *O1* which tries to remove *B* makes the next pointer of node *A* be NULL. Then a node insertion operation *O2* lets bucket node *A* point to the newly added node *B*. If the log record for operation *O2* persists early than the update log record of *O1*, the newly added node *B* would disappear after redoing the log updates in case of system crashes.

For file *create* and *delete* operations which only manipulate one single bucket at one time, the distributed journals can ensure the crash consistency of the hash table by redoing the update records in order within each log file to recover each bucket to a consistent state.

However, for the *rename* operation which may manipulate two buckets, we should resolve the update dependency spanning two mapped log files. Specifically, the rename operation involves a node removal to the source bucket and a node insertion to the destination bucket, the update records of which may be logged separately in two log files. In order to ensure the crash-atomicity of the rename operation, we should encapsulate the two update records together as one transaction. To this end, LVFS locks the two log files simultaneously, and then records the two updates together in one log file.

This approach would introduce recovery mistake as the transaction involves two log files but only record in one of them, which would be the same situation as the cross update order mentioned above. Therefore, we propose a log record barrier approach to addressing this problem. Specifically, LVFS would clean the target log file after applying the update records residing within the log file to the file system in sequence. LVFS performs log record barriers can not only make the normal journaling activities parallel, but also save recovery time and avoid garbage collection because of the immediate cleanup. Furthermore, the journaling in LVFS adopts circular logging pattern to save storage space.

3.2 File Data Management

PLFS adopts NCOW and light-weight journaling to keep data consistency. As the file system needs to copy the modification uncovered parts of a block out to the new allocated space, the traditional COW introduces redundant copies when the write is not block-aligned,. The byte-addressable NVMs give us new chance to solve the problem. Instead of remaining the block-grained file index like the block-aligned file systems, PLFS proposes fine-grained byte-unit index to manage the file tree, which could avoid the unnecessary copying completely but still guarantee data consistency.

In additional, PLFS offers transaction semantics to upper-layer applications. We add atomic semantics to the POSIX *fsync* system call, which is used to offer durable semantic to applications. The atomic and durable *fsync*

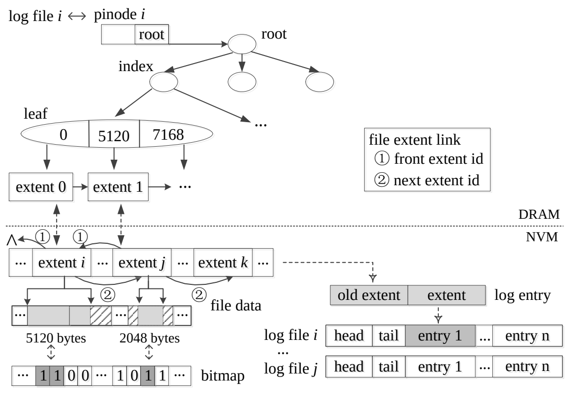


Figure 2: **The file data structures and layout in NVM.** *This figure shows*

semantic guarantees that the updates to a single file between two adjacent *fsync* system calls would be applied to NVMs atomically and durably.

3.2.1 File data structures and NVM layout

As shown in Figure 2, PLFS builds B+ tree to manage the file data. The leaf node is the byte-unit *extent* containing the start offset and cover size of a file data fragment, and the key of the index is set as the start offset of the *extent*. As the tree structures are difficult to implement correctly and efficiently in NVM [3, 14, 18], PLFS keeps the whole B+ tree structure in DRAM and only persists the leaf node in NVM. The *extent* in NVM are linked corresponding to B+ tree in DRAM so that the file tree could be rapidly rebuild when restart or recovery.

File tree The B+ tree adopts the fine-grained byte-unit index and byte-unit *extent* to manage the file data. For the block-unit index tree, the *extent* takes the block as the unit to records a continuous data space, and the modification of file data must in the whole block unit no matter how the I/O size required. However, the byte-unit index tree in PLFS records the start offset and continuous byte size of each data fragment and write the exact size of data by splitting one *extent* into multiple extents. Then the key of index would be set as the exact offset in byte-unit, not the block-unit one.

When the application write data to a file, firstly PLFS search the start *extent* of the write position in B+ tree. Then it would allocate new data space according to the write size to write the update. If the operation is appending, PLFS inserts the *extent* which records the new data fragment to the B+ tree. And for overwrite, PLFS splits the old *extent* to multiple new *extents* which manage the multiple byte-unit data fragments including the updated one, and inserts them to the B+ tree. In this way, we could completely avoid the unnecessary copy introduced by block-aligned write and we call this Non-Copy-On-write (NCOW).

Inevitably, this approach still exists the drawback that when the overwrite is random and in large quantity, there would be plenty of the leaf nodes which would lead the B+ tree too heavy to search, insertion or removal. The good news is that experiments in Section 4 has demonstrated that the overhead imported by B+ tree in this special situation is lighter than redundant copies, and then we can easily employ the NCOW in NVM.

Data space management For the management of file data space in NVM, PLFS remains the block-unit allocation and release just like other file systems. To save space, PLFS uses bitmap to display the usage of file data space in NVM. One bit in bitmap presents the mode of occupation of the corresponding data block. The setting of one bit express the corresponding data block is used, and the clearance indicates freedom. As the allocation and release of data space is frequent in *write* operation, to guarantee the efficiency and scalability, PLFS uses per-CPU link lists in DRAM to manage the file data space. Each link node records a continuous space presented by start block id and continuous block number.

In order to allocate the continuous free space as large as possible to reduce the number of leaf nodes in file tree, PLFS links the whole data space no matter used or freedom. Specially, each *extent* in file tree is also regarded as the link node which presents the used data space by recording the start block id and continuous block number of the data it occupies. And PLFS links the *extents* and the free space link nodes together according to the relative position of the data space in NVM they organize. When release a space being used, PLFS would merge the the free data space if the front or the next neighbor space is free, so that it could make the continuous free space as large as possible.

3.2.2 Data consistency

As mentioned above, keeping data consistency is still one of the most important thing to the file system. PLFS choose to use NCOW and distributed light-weight journaling to guarantee the data consistency, meanwhile providing high efficiency and scalability. The data journaling in PLFS is similar to the metadata journaling in LVFS, which is organized by multiple log files in NVM. Each log file is simply a continuous NVM area used to log updates and maintains the log head and tail within the start of the log file. And each one attaches to one single *pinode*. If the log file is full, PLFS would allocate another log file and stores the next log pointer in the end of it.

Because of the ROW strategy, each log entry in data journaling contains old *extent* and new allocated *extent* in NVM of the updated *extent*. For both of the insertion and update of the *extent*, PLFS need to firstly allocate and initialize a new *extent* in NVM. After logging to the log file, PLFS updates the link list of *extent* in NVM and set the corresponding bits in bitmap. For insertion, PLFS just logs the new *extent* in NVM to the log file. While for update, PLFS also need to records the old *extent* in NVM to the log file so that the bitmap in NVM and file tree in DRAM could be recovered when doing crash recoveries. And for the deletion, there is no need to log it, as the neighbor *extents* would update the adjacent pointers to maintain the link list of *extent* in NVM and PLFS adopts the redo journaling.

As the upper-layer applications don't always need to persist the update for each write, PLFS adds atomic semantics to the POSIX *fsync* system call, which guarantees the updates between two adjacent *fsync* to a single file would be atomical and durable. And the *write* system call would not persist the updates but only update the file tree in DRAM until the *fsync* system call is invoked. In this way, if the *fsync* operation to a file is not too frequent, the total write and persistent performance could be improved.

To guarantee the updates of the file is atomic, PLFS updates the log tail after all of updates are logged in the log file. And when finishing the updates of the *extent* link list and bitmap in NVM, PLFS would updates the log head to clean the log file. If the system crashes before the logging finished, there is nothing to do when recovery, as PLFS adopts the redo journaling and all the objects in NVM still stay the last consistent state. If it crashes after the log tail updated, PLFS would read the log entries to redo the updates. And if the crash happens after the cleaning of the log file, all updates have been persisted and nothing should be done to recovery.

3.3 atomic-msync

The byte-addressable NVMs make the DAX-mmap come true that applications could access NVM directly via load and store instructions by mapping the physical address in NVM into the user address space. However, it could not provide the data consistency when the system is crashed. NOVA [17] has realized the atomic-mmap to commit the changes atomically by COW and *msync*. The *atomic-mmap* has higher overhead than DAX-mmap because of the write back of the whole mmaped-pages, but providing stronger consistent guarantee. Then we want to know whether we could reduce the overhead, as there exists unnecessary copies that the *atomic-mmap* would write back all of the mmaped-pages even if part of them are just read.

PLFS proposes the *atomic-msync* which remains the strong data consistency but improves the performance by recording the dirty pages. To be specific, PLFS tracks the process of the page fault interrupt and records the mmaped dirty pages. If the *msync* system call is invoked, PLFS just atomically writes the dirty pages back and sets the pte clean. And if there is no write to the persisted page before the next *msync*, it would not be wrote back any more. Then the *atomic-msync* in PLFS could not only guarantee the strong consistency but also remain higher the efficiency.

3.4 Implementation

According to the design described above, we now present the implementation of the common file system operations in PLFS.

Creating a file For the *create* operation, firstly PLFS allocates and initializes the *pinode* in DRAM and NVM respectively, inserts the new one in DRAM to the hash bucket and the directory link list. Then PLFS appends a log entry which is related to the insertion to log file and updates the log tail. Finally, PLFS inserts the *pinode* to the hash bucket and the directory link list in NVM, and updates the log head to clean the log file.

Deleting a file When deleting a file, PLFS firstly removes the deleted *pinode* in DRAM from the hash bucket and the directory link list. Then it appends the log entries to log file and updates the log tail. As mentioned above, PLFS adds two log entries. One records the front node in hash bucket and the deleted *pinode*, the other records the front sibling in the directory link list and the deleted *pinode*. Finally, PLFS removes the *pinode* from the hash bucket and the directory link list in NVM, updates the log head to clean the log file.

Renaming a file Renaming a file involves two operations, one is the removal of the old *pinode* from the old hash bucket and directory, the other is the insertion of the new *pinode* to the new hash bucket and directory. PLFS firstly allocates a new *pinode* in NVM and initializes it with the new information. Then PLFS removes the *pinode* from the old hash bucket and directory, and inserts it with new name to the new hash bucket and directory in DRAM. Here comes the appending of multiple log entries to the single log file mapped with the new hash bucket. After the update of the log tail, PLFS removes the *pinode* from the old hash bucket and directory in NVM, and adds the new one to the new hash bucket and directory in NVM. Finally, it cleans the log file by updating the log head.

write and fsync As shown in Figure 3, the original part of the file tree consists of *extent0*, *extent1* and *extent2*, and we want to write new data from offset 9216 with 2048 bytes. Firstly PLFS searches the file tree and finds that the covered data space is organized by *extent1* and *extent2*. As there are still some parts of the data s-

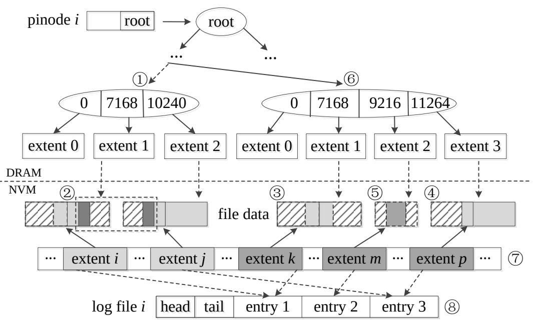


Figure 3: **The file write operation.** *This figure shows*

pace not modified in *extent1* and *extent2*, PLFS splits the old *extent1* and *extent2* in DRAM to *extent1* and *extent3* with data not modified. Then it allocates one new block to fill the new data and insert the new *extent2* to file tree in DRAM. Now all of the updates in DRAM has finished and the normal *write* operation is finished.

If the user application wants to persist the updates immediately, the system call *fsync* should be invoked. That is after all of the updates in DRAM, PLFS allocates and initializes three new *extents* in NVM with the updated metadata. Specifically, the new *extent* k, m, p record the whole metadata including the pointer to adjacent *extents*, while the adjacent pointers in their neighbors whose data are not modified are not updated until the log entries added to the log file. After the update of the log tail, PLFS updates the adjacent pointers of the affected neighbors and updates the corresponding bits in bitmap. Finally, PLFS updates the log head to clean the log file.

4 Evaluation

In this section we evaluate the performance and scalability of PLFS using a set of micro- and macro-benchmarks to answer the following questions:

(1) How does LVFS performs against VFS combining with the underling file system on the common directory operations?

(2) How efficient is PLFS block-unaligned write operations compared to the state-of-the-art file systems?

(3) Does the design of fine-grained byte-unit index degrade the performance of block-aligned write and file read operations?

(4) How is the scalability of the common file system operations in PLFS?

4.1 Experimental Setup

All of our experiments are conducted on a Intel Xeon E7 server and the processors run at 2.2GHz with 40 cores. The emulation of NVM is based on 150G DRAM memory, that is seen by OS as Persistent Memory region.

We evaluate PLFS on Linux kernel 4.6.5 against four file systems. Two of them, NOVA and PMFS are the only available open source file systems based on NVM that we know of. Two others, Ext4-DAX(ordered) and XFS-DAX are traditional file systems designed for block devices that could run directly on NVM, bypassing the page cache. NOVA is the state-of-the-art NVM-based file system providing the same strong data consistency guarantees as PLFS. While the others only journal metadata and perform in-place updates for file data.

4.2 Micro-benchmarks

We create two micro-benchmarks to evaluate the throughout and scalability of PLFS. The first one is called *p-fops*, which consists of three common directory operations: *create*, *rename* and *unlink*. The second one is called *p-frwrite* that contains *write* and *read* operations. And each micro-benchmark creates a number of threads performing the corresponding operations in parallel and we vary the number of threads from 1 to 40.

4.2.1 *p-fops* micro-benchmarks

Single Thread The micro-benchmark *p-fops* aims to compare the performance of LVFS with VFS combining with the underlying file system on the common directory operations. It creates a set number of files under the same directory, then renames these files to the same directory, and finally deletes all of them. We change the number of files from 100 to 1000000. Figure 4(a) to 4(c) show the throughput of the three operations evaluated by *p-fops*.

PLFS provides the highest throughput for each operation, outperforms NOVA by 27.5% to 46.2% on *create* operation, 59.4% to 1X on *rename* operation, and 81.1% to 1.65X to *unlink* operation. We attribute the performance improvement to the integration of VFS layer with the naming space of the underlying file system.

For *create* operation, the traditional I/O stack needs to firstly do double lookup on VFS and the underlying file system, then creates the new *inode* and *dentry*, finally inserts the *inode* and *dentry* to the *inode_hashtable*, *dentry_hashtable* and directory structures such as radix tree in NOVA respectively. While for LVFS, it just need to lookup the pathname iteratively in hash table, then create the new *pinode* and insert it to the head of the hash table and the directory link list. The overhead of the double lookup and the allocation, initialize and insertion of the two structures are larger than LVFS which compacts the objects to a single one. In addition, the organization of directory structures also leads to the gap of the performance of these two mechanisms that O(1) is the lowest time complexity to insert and remove provided by link list and radix tree is certainly larger than that. For *rename* and *unlink* operations, the extra overheads are similarly introduced by the double lookup, the heavy insertion and removal to radix tree and the delete of both *dentry* and *inode* for *unlink*.

Multi-Threads We then evaluate the scalability of the three operations using multi-threaded *p-fops*, each thread handles 10000 files, and vary the number of threads from 1 to 40. Figure 5(a) to 5(c) show the average throughput of each thread under the different number of parallel threads tests.

As we can see in Figure 5(a) and 5(c), the scalability of *create* and *unlink* are good for both PLFS and NOVA. For *rename* operation, PLFS still remains the high scalability while all of the other file systems show the scalable bottlenecks. For the traditional I/O stack, the general lock protections are provided by VFS. As the *dentry_hashtable* adopts RCU lock to protect each hash bucket, there is few competition between multiple threads for *create* and *unlink*. But for *rename* operation which may involve operations of two hash bucket, VFS provides the global sequence lock (*rename_lock*) to guarantee the correctness which would introduce scalable bottlenecks. The crafted structures of LVFS solve the problem that it gives each bucket a sequence lock to protect the *rename* operation.

4.2.2 *p-frwrite* micro-benchmarks

Single Thread of *write* Operation To evaluate the performance benefit of *write* operation, *p-frwrite* appends data to a file in 16GB with a single thread, then rewrite and random write the file respectively. Especially the *random write* operation means the start position of each write operation is random with byte-unit. To observe how the performance impacted by the write size, we alter the I/O size from 256B to 256KB. As NOVA is the only file system evaluated provides the strong data consistency, we just compare the *re-write* and *random-write* performance with NOVA. Figure 6(a) to 6(c) show the throughput of the three write operations with difference write sequence.

PLFS provides the highest throughput for all of the three situations. It outperforms NOVA by 19.6% to 10.4X on *append*, 19.4% to 6.27X on *rewrite*, and 16.2% to 1.16X on *random write*. The improvement of the performance is contributed by the fine-grained byte-unit index tree which completely avoids the copying introduced by the COW of the block-unit file index tree.

As can be seen in Figure 6(a) to 6(c), the block-unaligned *write* operation with the 256B and 1KB I/O size bring higher performance improvement than the other block-aligned *write* operations because of the NCOW strategy. The different sequence of *write* can lead to different scales of the B+ tree in PLFS and *random write* would finally build a huge file tree that each *extent* in it only manages few data less than the block size. The result showed by Figure 6(c) can demonstrate that the overhead of redundant copying is larger than the heavy overhead introduced by the B+ tree which with quantity of leaf nodes as mentioned in Section 3.2.1.

Multi-Threads of *write* Operation Then we experiment the write scalability with two groups: one creates a number of threads from 1 to 40 and each thread makes 4KB write to the 1GB file, another creates a number of threads from 1 to 40 and each thread makes 1KB write to the 512MB file to obviously show the non block-aligned case. Figure 7(a) to 7(c) show the throughput of the 4KB write operations, and Figure 8(a) to 8(c) show the 1KB write results.

Both of PLFS and NOVA have good scalability for the three write operations. For the 4KB write to 1GB file, the append and rewrite operation are block-aligned operations, and PLFS performs sightly better than NOVA. For the random write shown in Figure 7(c), PLFS performs better than NOVA even if the B+ tree is largest among the three situations. And for the 1KB write to 512MB file shown in Figure 8, PLFS performs much better than NOVA in all of the three situations, as all of the them are block-unaligned.

The performance of *read* Operation To guarantee that the design of the fine-grained byte-unit index does not affect the *read* operation, we also evaluate the *read* performance. And we first evaluate the *read* operation performance of PLFS in different file status by changing write size and write sequence. Figure 9(a) to 9(c) show the results. We could acknowledge that no matter how we modify the write size in *append* and *rewrite*, the read performance remains stable in all kinds of read I/O size. Although the *random write* situation waves sightly, it can be explained that increasing write size could greatly reduces the *extent* numbers when we do *random write*.

As the 4KB write size is the worst case of PLFS, we then compare the *read* performance in different read size with NOVA in this case. And the results are shown in Figure 10(a) to 10(c). It can be seen that the read performance of PLFS is basically same as NOVA. Although the throughput of PLFS is sightly lower than NOVA under the *random write* situation which is caused by the heavy B+ tree, the gap between them is so small that it can be ignored.

4.3 Macro-benchmarks

Filebench is a file system and storage benchmark that can simulate a variety of complex workloads by specifying different models. We use Varmail and Fileserver, which are write-intensive workloads, to evaluate the performance of PLFS for real world applications. Figure 11 summarizes the characteristics of the workloads. As shown in Figure 12, We run these benchmarks in a single thread and there are 1 to 40 threads concurrently.

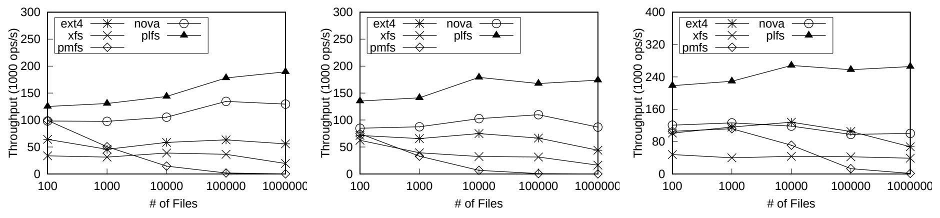
Fileserver PLFS provides the highest throughput for all of the four situations and stays high scalability simultaneously. As the average file size in Fileserver is 128KB, it writes more data and the data write speed dominates. For the results shown in Figure 12(a), the throughput of PLFS is same as NOVA in all of the tests with different parallel threads. This is because the write is block-aligned, and there is no extra copying for the append to the workload. For the results shown in Figure 12(b), PLFS outperforms NOVA by 3.7X to 5.4X with 1KB write size. And this is mainly contributed to the fine-grained byte-unit index tree which enables the NCOW strategy to avoid copying overhead.

Varmail The Varmail acts as a mail server and both directory operations and file I/O contribute the performance. Therefore, even if the *write* is block-aligned shown in Figure 12(c), PLFS still outperforms NOVA by 27% to 43% because of the faster *create* and *unlink* operations. And for the block-unaligned situation shown in Figure 12(d), PLFS outperforms NOVA by 68.9% to 80.8%. As the total file size is only 16KB in Varmail, the benefit of NCOW is not so obviously as Fileserver.

4.4 Mmap I/O

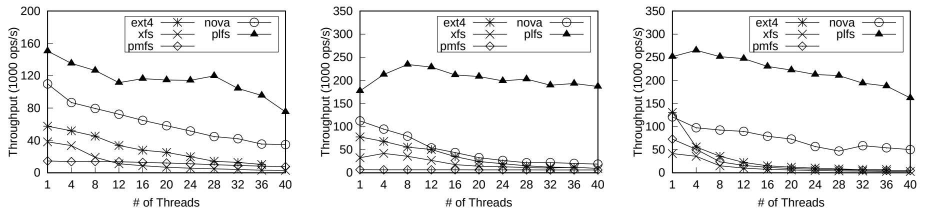
In this section, we use *fio* to evaluate the performance of *mmap*. We change the file size from 1M to 12M, and finally mmap a 16GB file to see the worse case of PLFS and NOVA. For each file, we use *fio* to make the 4KB write size for the mmapped data and *mfsync* the whole file after each *write* and the corresponding *msync*. Figure 13 shows the throughput of the operations.

PLFS provides 26.9X to 389.6X throughput improvement compared to NOVA, the state-of-art NVMFS which provides equally *mmap* data consistency guarantees as PLFS. As we add the dirty page tracing when page fault occurs, it sightly increases the overhead when we just *mmap*-write and *msync* each write section. However, it perfectly solves the problem that we could not know the dirty pages after *mmap*, then we write back all of the mmaped pages to NVM even if we do not modify them. Based on this, PLFS just write the dirty page back when the system call *msync* invoked, while NOVA needs to write the whole mmapped data back which leads to the significant write amplification.



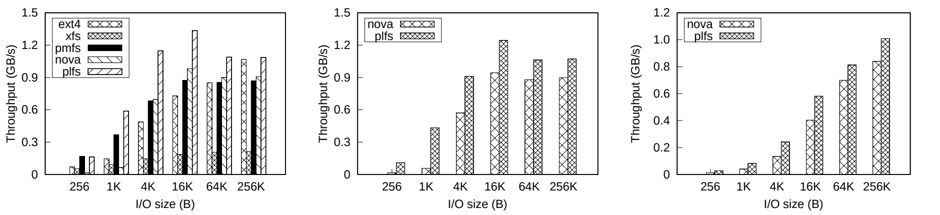
(a) create (b) rename (c) unlink

Figure 4: **Evaluation of common directory operations including *create*, *rename* and *unlink*.** *This figure depicts*



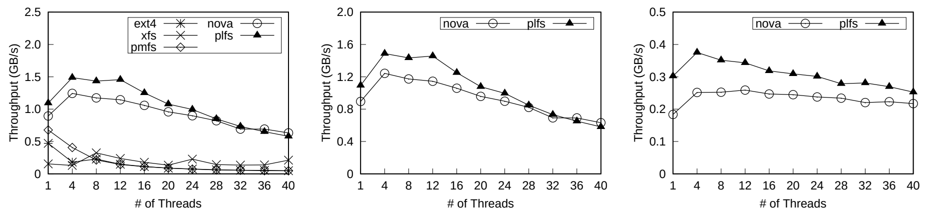
(a) create (b) rename (c) unlink

Figure 5: **Evaluation of the scalability of the three common directory operations including *create*, *rename* and *unlink*.***This figure depicts*



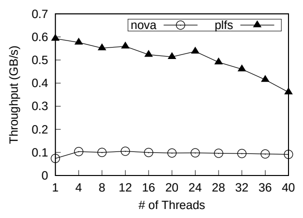
(a) append (b) rewrite (c) random-write

Figure 6: **Evaluation of *append*, *rewrite* and *random-write* in different I/O size.** *This figure shows*

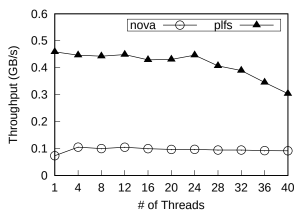


(a) append (b) rewrite (c) random-write

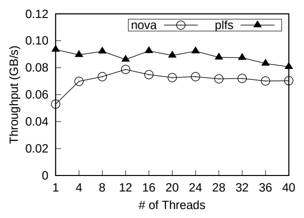
Figure 7: **Evaluation of the scalability of 4KB *append*, *rewrite* and *random-write* to the 1GB file.** *This figure shows*



(a) append

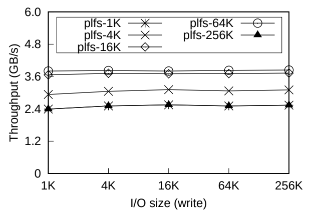


(b) rewrite

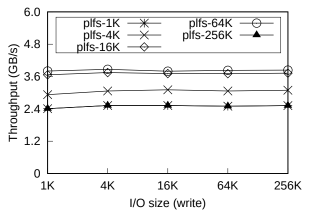


(c) random-write

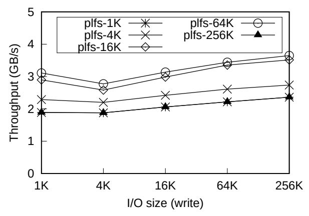
Figure 8: **Evaluation of the scalability of 1KB *append*, *rewrite* and *random-write* to the 512MB file.** *This figure shows*



(a) append

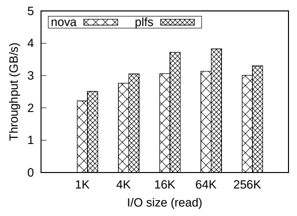


(b) rewrite

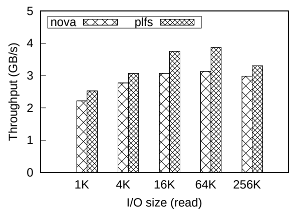


(c) random-write

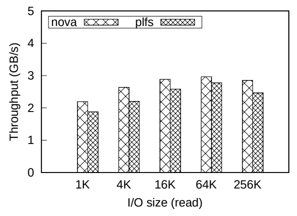
Figure 9: **Evaluation of *read* performance of PLFS in different file status.** *This figure shows*



(a) append



(b) rewrite

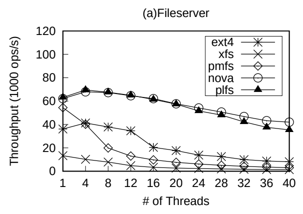


(c) random-write

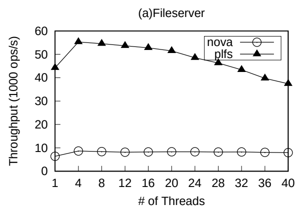
Figure 10: **Evaluation of *read* performance with 4KB-write-size file.** *This figure shows*

Workload	Average file size	number of files	I/O size(r/w)	R/W ratio	Run time
Fileserver	128KB	5K	1MB/1MB	1:2	60s
Fileserver-1K	128KB	5K	1MB/1KB	1:2	60s
Varmail	16KB	50K	1MB/16KB	1:1	60s
Varmail-1K	16KB	50K	1MB/1KB	1:1	60s

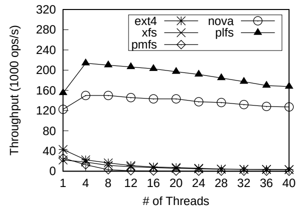
Figure 11: **Filebench workload characteristics (with multi-threads).** *This figure shows*



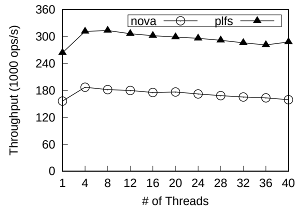
(a) FileServer



(b) FileServer-1K



(c) Varmail



(d) Varmail-1K

Figure 12: **Evaluation of Filebench performance.** *This figure shows*

	1M	4M	8M	12M
nova	7.351	2.041	1.031	0.694
plfs	204.8	250.98	303.907	271.087

Figure 13: **The throughput of *mmap* (in MB/s) evaluated by *fio*.** *This figure shows*

5 Related Work

NVM-aware file systems. Emerging fast non-volatile memories exhibit significantly different properties from traditional disk-based persistent storage in terms of both performance and access interfaces. The advantages provided by such fast storage open up many opportunities to boost application I/O performance. However, existing file systems designed for disk-based storage incur unnecessary software overhead on fast NVMs. Recent research work proposes new file systems highly optimized around the advanced properties of NVMs to improve performance. BPFS [5] proposes a short-circut shading page technique to reduce the copy-on-write overhead for maintaining consistency on NVMs, and an elegant epoch commit strategy that decouples ordering from durability to mitigate the cache flush overhead. PMFS [13] advocates to fully leverage the characteristics of NVMs and modern processors to optimize the file system performance. Specifically, PMFS [13] proposes bypassing the page-cache layer to support direct NVM access, thus avoiding double data copying, and adopts a fine-grained logging technique combined with atomic word updating supported by modern processors to reduce the journaling overhead for consistency. Aerie [15] argues that the overhead of traditional software abstractions will squander the performance provided by NVMs, and hence proposes a user-space direct access architecture for NVM store. Aerie offers each application a library to allow direct, user-space access to NVMs, and provides a global, consistent namespace, and data protection among different applications and users by adopting a centralized trusted service running in the user-space mode. The most recent work on NVM-aware file systems are NOVA [17] and HiFS [11]. NOVA [17] is a log-structured file system on NVMs, which adopts per-inode logs to provide high parallelism and incorporates DRAM to speed up metadata operations. NOVA [17] is a log-structured file system, which adopts light-weight journaling to journal the update of the log tail instead of the metadata/data to be modified and employs per-CPU journals to ensure good scalability on many-core. However, their light-weight per-CPU journal approach is only suited for log-structured file systems, which need to garbage collect stale objects in the logs. HiFS argues that using DRAM to temporarily store the continuous writing data in DRAM until a synchronous action occurs can improve file system performance when the write speed of NVMs is significantly slow than that of DRAM.

NVM logging. As the emerging NVMs are expected to offer DRAM-like performance and data persistence, many recent work proposes to adopt NVMs to perform database logging and sharps the traditional block-oriented logging on NVMs to reduce the logging overhead [16, 2, 7, 1, 10]. Wang et al. [16] proposes a distributed logging to provide fast, scalable logging performance on multi-core systems incorporating NVMs. The proposed distributed logging uses logical clocks to track and resolve transaction dependencies among multiple logs and update records and adopts the group commit mechanism to alleviate the expensive cache flush overhead on NVMs. Our work also adopts distributed journaling mechanism to avoid global contention point, thus improving concurrency, However, in contrast to tracking transaction dependencies, our work adopts scalable, persistent data structures to eliminate dependencies between updates.

Transactional file systems. Atomic-msync [12] proposes to provide atomic-msync primitive to shoulder the burden of the upper-layer application atomic protocol design and implementation, thus improving the application performance. Atomic-msync [12] leverages the file system data journaling mechanism to provide all or nothing atomicity at the data level.

Scalable I/O stacks. Clements et al. [4] propose the commutativity principle to guide how to design and implement a scalable software. In order to demonstrate their principle, their work implements a scalable, in-memory file system that adopts scalable data structures whenever possible to build their file system.

6 Conclusions and Future Work

oooooooooo

7 Acknowledgments

This work is supported in part by 973 program (No. 2014CB340300) and NSFC (No. 61322207&61421003).

References

[1] ARULRAJ, J., PAVLO, A., AND DULLOOR, S. Let's talk about storage & recovery methods for non-volatile memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (2015), pp. 707–722.

[2] CHATZISTERGIOU, A., CINTRA, M., AND VIGLAS, S. D. REWIND: recovery write-ahead system for in-memory non-volatile data-structures. *PVLDB 8*, 5 (2015), 497–508.

[3] CHEN, S., AND JIN, Q. Persistent b+-trees in non-volatile main memory. In *PVLDB* (2015), pp. 768–797.

[4] CLEMENTS, A. T., KAASHOEK, M. F., ZELDOVICH, N., MORRIS, R. T., AND KOHLER, E. The scalable commutativity rule: Designing scalable software for multicore processors. *ACM Trans. Comput. Syst. 32*, 4 (2015), 10:1–10:47.

[5] CONDIT, J., NIGHTINGALE, E. B., FROST, C., IPEK, E., LEE, B. C., BURGER, D., AND COETZEE, D. Better I/O through byte-addressable, persistent memory. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009* (2009).

[6] DONG, M., AND CHEN, H. Soft updates made simple and fast on non-volatile memory. In *2017 USENIX Annual Technical Conference, USENIX ATC '17* (2017).

[7] HUANG, J., SCHWAN, K., AND QURESHI, M. K. Nvram-aware logging in transaction systems. *PVLDB 8*, 4 (2014), 389–400.

[8] KANG, J., ZHANG, B., WO, T., HU, C., AND HUAI, J. Multilanes: A providing virtualized storage for os-level virtualization on many cores. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies, FAST 2014* (2014), pp. 317–329.

[9] KANG, J., ZHANG, B., WO, T., YU, W., DU, L., MA, S., AND HUAI, J. Spanfs: A scalable file system on fast storage devices. In *2015 USENIX Annual Technical Conference, USENIX ATC '15* (2015), pp. 249–261.

[10] KIM, W., KIM, J., BAEK, W., NAM, B., AND WON, Y. 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* (2016).

[11] OU, J., SHU, J., AND LU, Y. A high performance file system for non-volatile main memory. In *Proceedings of the Eleventh European Conference on Computer Systems, EuroSys 2016* (2016).

[12] PARK, S., KELLY, T., AND SHEN, K. Failure-atomic msync(): a simple and efficient mechanism for preserving the integrity of durable data. In *Eighth Eurosys Conference 2013, EuroSys '13* (2013).

[13] RAO, D. S., KUMAR, S., KESHAVAMURTHY, A., LANTZ, P., REDDY, D., SANKARAN, R., AND JACKSON, J. System software for persistent memory. In *Ninth Eurosys Conference 2014, EuroSys 2014* (2014).

[14] VENKATARAMAN, S., TOLIA, N., RANGANATHAN, P., AND COMPBELL, R. H. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies, FAST 2011* (2011), pp. 167–181.

[15] VOLOS, H., NALLI, S., PANNEERSELVAM, S., VARADARAJAN, V., SAXENA, P., AND SWIFT, M. M. Aerie: flexible file-system interfaces to storage-class memory. In *Ninth Eurosys Conference 2014, EuroSys 2014* (2014).

[16] WANG, T., AND JOHNSON, R. Scalable logging through emerging non-volatile memory. *PVLDB 7*, 10 (2014), 865–876.

[17] XU, J., AND SWANSON, S. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *14th USENIX Conference on File and Storage Technologies, FAST 2016* (2016), pp. 323–338.

[18] YANG, J., WEI, Q., CHEN, C., WANG, C., YONG, K. L., AND HE, B. Nv-tree: Reducing consistency cost for nvm-based single level systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies, FAST 2015* (2015), pp. 167–181.