Soft Updates Made Simple and Fast on Non-volatile Memory

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

https://www.usenix.org/conference/atc17/technical-sessions/presentation/dong

This paper is included in the Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC '17).

July 12–14, 2017 • Santa Clara, CA, USA

ISBN 978-1-931971-38-6

Soft Updates Made Simple and Fast on Non-volatile Memory

Mingkai Dong, Haibo Chen
Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University

Abstract

Fast, byte-addressable NVM promises near cache latency and near memory bus throughput for file system operations. However, unanticipated cache line eviction may lead to disordered metadata update and thus existing NVM file systems (NVMFS) use synchronous cache flushes to ensure consistency, which extends critical path latency.

In this paper, we revisit soft updates, an intriguing idea that eliminates most synchronous metadata updates through delayed writes and dependency tracking, in the context of NVMFS. We show that on one hand byte-addressability of NVM significantly simplifies dependency tracking and enforcement by allowing better directory organization and closely matching the per-pointer dependency tracking of soft updates. On the other hand, per-cache-line failure atomicity of NVM cannot ensure the correctness of soft updates, which relies on block write atomicity; page cache, which is necessary for *dual views* in soft updates, becomes inefficient due to double writes and duplicated metadata. To guarantee the correctness and consistency without synchronous cache flushes and page cache, we propose pointer-based *dual views*, which shares most data structures but uses different pointers in different views, to allow delayed persistency and eliminate file system checking after a crash. In this way, our system, namely SoupFS[1], significantly shortens the critical path latency by delaying almost all synchronous cache flushes. We have implemented SoupFS as a POSIX-compliant file system for Linux and evaluated it against state-of-the-art NVMFS like PMFS and NOVA. Performance results show that SoupFS can have notably lower latency and modestly higher throughput compared to existing NVMFS.

1. Introduction

Soft updates, which uses delayed writes for metadata updates, tracks per-pointer dependencies among updates in memory, and enforces such dependencies during write back to disk, is an intriguing idea that promises metadata update latency and throughput close to memory-only file systems [10, 11, 23, 35]. However, soft updates is also known for its high complexity, especially the complex dependency tracking as well as enforcement (like roll-back/forward to resolve cyclic dependencies, which also lead to double writes) [1, 3, 9, 13, 22]. A known file system developer Valerie Aurora argued that "soft updates are, simply put, too hard to understand, implement, and maintain to be part of the mainstream of file system development" [1].

In this paper, we revisit soft updates for NVM and argue that two main sources of the complexity are: 1) *the mismatch between per-pointer based dependency tracking and the block-based interface of traditional disks*; 2) *excessively delayed writes that complicate dependency tracking*. We then show that soft updates can be made simple by taking advantage of the byte-addressability and low latency offered by NVM. Byte-addressability matches the per-pointer based dependency tracking by eliminating false sharing among different data structures and avoiding cyclic dependencies and complex roll-back/forward. Byte-addressability also allows the use of more efficient data structures like hash tables in the directory organization to further simplify the dependencies of file system operations. Besides, page cache and disk scheduler can be excluded from the storage hierarchy because of the byte-addressability and low latency of NVM, so that soft updates can use in-place writes with delayed persistency to simplify the dependency tracking. The simplified storage hierarchy also eliminates the gap between the page cache and the file system, making the dependency tracking and enforcement semantic-aware and even simpler.

However, there is still a major challenge that impedes the application of soft updates to NVM. Since page cache, the software cache layer designed for slow storage media, is removed for performance concerns, file system updates are directly written to CPU cache and persisted to NVM later. Unlike page cache that can be precisely controlled by file systems, CPU cache is hardware-managed such that file systems cannot control the eviction of cache lines. State-of-the-art NVM file systems (NVMFS) [8, 45], like traditional disk-based file systems, use logging or shadow copying to ensure crash consistency. Yet, instead of buffering data for explicit and periodic flushing later, NVMFS has to eagerly flush critical metadata in case of accidental eviction of such metadata to NVM in a wrong order. This necessitates the uses of high latency operations like *clflush/clflushopt+sfence* in

[1] Short for Soft updates inspired File System

the critical path of file system related syscalls, which inevitably extends the critical path latency.

To overcome the consistency issue from unanticipated cache line eviction without page cache and cache flush operations, we review *dual views*, a latest view and a consistent view, which is used in soft updates for file system metadata. All metadata in the consistent view is always persisted and consistent, while metadata in the latest view is always up-to-date and might be volatile. Without caring about the cache line eviction, a syscall handler operates directly in the latest view and tracks the dependencies of modifications. Unanticipated cache line eviction in the latest view can never affect the persisted metadata in the consistent view by design. Background *persisters* are responsible for asynchronously persisting metadata from the latest view to the consistent view according to the tracked dependencies. They use *clflush/clflushopt+sfence* operations to enforce the update dependencies in background without affecting the syscall latency. A naive approach to providing *dual views* is duplicating all metadata in the file system. Such an approach doubles the memory usage and causes unnecessary memory copies when synchronizing metadata between the latest view and the consistent view. To implement *dual views* efficiently, we propose pointer-based *dual views*, in which most structures are shared by both views and different views are observed by following different pointers. Thanks to pointer-based *dual views*, SoupFS avoids almost all synchronous cache flushes in the critical path, and the consistent view can be immediately used without performing file system checking or recovery after crashes.

We have implemented SoupFS as a POSIX-compliant[2], NVM-based file system at the backend of the virtual file system in Linux. Evaluations using different NVM configurations show that SoupFS provides notably lower latency and modestly higher throughput compared to state-of-the-art NVM file systems such as PMFS and NOVA. Specifically, SoupFS achieves up to 80% latency reduction for file system related syscalls in the micro-benchmarks and improves the throughput by up to 89% and 50% for Filebench and Postmark.

In summary, the contributions of this paper include:

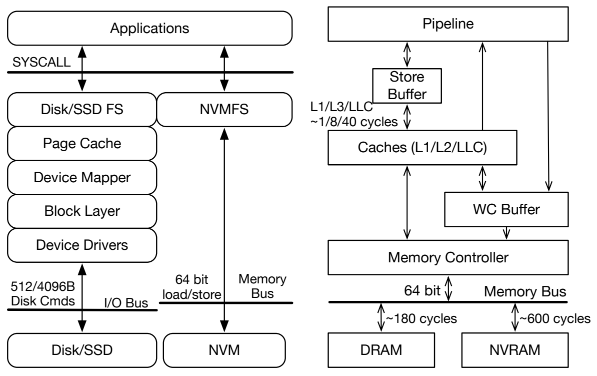
- A detailed analysis of the complexity of soft updates and the argument that soft updates can be made simple for NVM (§2).
- A review of the update dependencies of file systems on NVM, a simple semantic-aware dependency tracking and enforcement mechanism and efficient pointer-based *dual views* (§3).
- An implementation of SoupFS on Linux and an extensive evaluation (§4) that confirms the efficiency of SoupFS.

[2] SoupFS has passed the POSIX-compliant test in `http://www.tuxera.com/community/posix-test-suite/`.

2. Background and Motivation

2.1 NVM and NVMFS

Emerging NVM technologies such as PCM, STT-MRAM, Memristor, NVDIMM and Intel/Micron's 3D XPoint are revolutionizing the storage hierarchy by offering features like byte-addressability, non-volatility, and close-to-DRAM speed. STT-RAM has lower latency than DRAM but high cost, making it a promising replacement for on-chip cache instead of DRAM replacement [47]. Other emerging NVM media like PCM or 3D XPoint generally have higher latency especially higher write latency than DRAM, which indicates that synchronous write to NVM would cause higher latency than that to DRAM. NVDIMM, a commercially available NVM solution, generally has the same performance characteristics with DRAM as it is essentially battery-backed DRAM, though it is usually with 10–20X higher price than DRAM according to a recent price quotation.



(a) Storage stack (b) CPU path

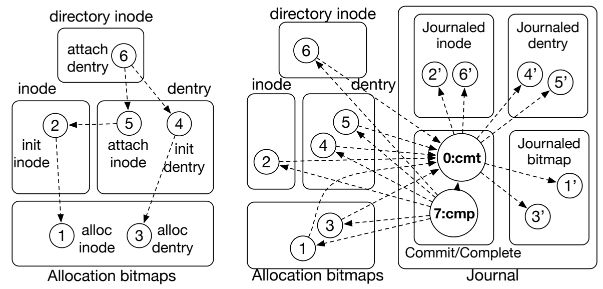
Figure 1: Storage stack and path

While new software can leverage the load/store interface to access NVM directly, quite a lot of software may continue to access persistent data in NVM through the file system interface. Hence, there have been intensive efforts in designing NVM file systems (NVMFS) [4–6, 8, 25, 44, 45]. Figure 1(a) illustrates the storage stacks from applications to persistent storage for disks (including SSD) and NVM. Compared to disk file systems, NVMFS can avoid major storage software layers like page cache, device mapper, block layer and drivers, but instead only relies on memory management for space management. However, there are also several challenges that require a redesign of file systems for NVM.

Fine-grained Failure-Atomic Updates: Although it is claimed that memory controllers supporting Intel DIMM will also support Asynchronous DRAM Refresh [32], the failure-atomic write unit is only one cache line size, still far less than 512-/4096-byte for disks. This fine-grained failure atomicity prevents the use of prior approaches (like backpointers [3]) relying on coarse-grained failure atomicity.

Hardware-controlled Ordering: NVMFS elevates the level of persistency boundary from DRAM/Disk to CPU cache/NVM. However, unlike disk-based file systems that have complete control of the order of data flushed to disk, CPU cache is hardware-managed and unanticipated cache line eviction may break the ordering enforced by sfence/mfence, which only orders on-chip visibility of data updates across CPU cores. To this end, prior NVMFS needs to eagerly use *clflush* or *clflushopt* to flush data from CPU cache to the memory controller [4, 8, 25, 44, 45]. *clflushopt* allows asynchronously flushing data compared to synchronous and serialized feature of *clflush*. But the ordering of *clflushopt* must be enforced by memory fences like *sfence*. Eagerly flushing cache lines and persisting data would cause high latency in the critical path, especially for NVM with higher write latency than DRAM.

Software Efficiency: Unlike in conventional file systems where slow storage devices dominate access latency, the cost of *clflush* and *clflushopt+sfence* is much higher compared to CPU cache accesses. It becomes the new bottleneck and must be minimized in the critical path to approach the near-cache access speed. Besides, the scarcity of CPU cache volume requires economizing cache usage to provide more space for applications.



(a) General (b) Journaling

Figure 2: Persistency dependency of creating a file

2.2 The Cost of Consistency

To address fine-grained atomicity and hardware-controlled ordering, prior file systems need to order operations carefully and use synchronous flushing to preserve crash consistency, which leads to high latency. Figure 2(a) illustrates the dependency to create a file, where the dashed arrows denote the persistency ordering. For example, the arrow from *init inode* to *alloc inode* dictates that the initialization of the new inode must not be persisted until its allocation information is made persistent. Prior file systems like PMFS [8] usually use journaling to handle this issue, which further complicates the dependencies (as shown in Figure 2(b)) and still requires eagerly flushing the logs. In this example, there are around 19 persistency dependencies to be respected, which requires around 14 *clflushes*. Packing multiple journaled metadata into a single cache line can reduce the number of *clflushes*, but cannot eliminate them.

2.3 Soft Updates

Soft updates [10, 11, 23, 35] is an intriguing metadata update technique and has been implemented in FreeBSD UFS [23]. It has the promise of mostly eliminating synchronous metadata updates and providing fast crash recovery by instantly providing a consistent file system. While soft updates is an intriguing and concise idea to achieve low latency and high throughput, the block interface exposed by the underlying disk complicates dependency tracking and enforcement in the following ways:

Block-oriented directory organization complicates dependencies: Like many other disk-based file systems, soft updates treats directories as regular files organized by direct blocks and indirect blocks. This block-oriented directory organization simplifies the implementation of file systems for block devices but complicates the dependencies due to false sharing. For example, placing multiple dentries in the same block allows cyclic dependencies, which must be resolved by complicated roll-back/forward. It also necessitates the additional tracking of whether the block to store the new dentry is newly allocated or reused, so that it can be treated differently in the enforcement.

Delayed writes complicate dependency tracking: Delaying disk writes of metadata updates is one key idea of soft updates specially designed for disk-based storage with high write latency. A sequence of dependent metadata changes, which otherwise can be written synchronously, is delayed with various dependency tracking structures attached. While asynchronous disk writes improve creation throughput by a factor of two compared with synchronous writes [23], soft updates must track the status of delayed operations to maintain ordering for integrity and security. However, the page cache usually is unaware of the content in the page, which creates a semantic gap between the page cache (where enforcement happens) and the file system (where tracking happens). The gap forces soft updates to involve complex structures for status and dependency tracking, which complicates both the critical path of synchronous system calls and the *syncer* daemon that is responsible for flushing the delayed writes.

Roll-back/forward complicates dependency enforcement: Soft updates tracks per-pointer metadata updates to eliminate false sharing. However, during enforcement, as a disk block still contains many metadata structures, there are still many cyclic dependencies at the block level during write-back. Soft updates handles this complexity by rolling back metadata changes that have pending dependencies to only write consistent metadata updates and then rolling forward the reverted metadata changes to persist the change again. This, however, would double the disk updates and diminish its gain over journaling mechanisms [36].

Soft updates is considered difficult and complicated to implement and maintain [1, 13]. By rethinking soft updates on NVM, we find that the byte-addressability of NVM can simplify the design of soft updates and delayed persistency of soft updates can further boost the performance of file systems on NVM.

3. Design and Implementation

To embrace high performance and byte-addressability of NVM, we design SoupFS, a soft updates implementation that is simple and fast on NVM. SoupFS redesigns the directory organization using hash tables to simplify the complicated dependencies caused by block-oriented directory organization. The roll-back/forward complexity is eliminated by removing page cache, thanks to byte-addressability of NVM. The removal of page cache also enables a semantic-aware dependency tracking which alleviates the complexity caused by delayed writes.

As a result, a syscall handler simply tracks the operation type along with related pointers, and with file system semantics in mind, background *persisters* can enforce the persistency and dependencies according to the type and pointers tracked during the syscall.

SoupFS is fast mainly because it applies delayed persistency which eliminates almost all synchronous cache flushes in the file system syscall critical path. Providing *dual views*, a latest view and a consistent view, of file system metadata is the key technique to allow delayed persistency and eliminate file system checking after a crash.

However, page cache, which facilitates the implementation of *dual views* in soft updates, is removed in SoupFS for performance and simplicity. To provide *dual views* without page cache, we propose efficient pointer-based *dual views* by specially designing its metadata structures so that most structures are shared by both views and different views are observed by following different pointers.

3.1 Directory Organization

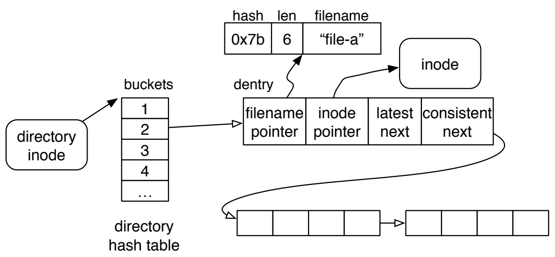


Figure 3: Directory and dentries in SoupFS

Directory operations are the core of many file system syscalls. In traditional file systems, a directory is organized akin to regular files but with different registered operations. Despite poor performance of lookups due to linear scans, reusing the regular file structures as dentry arrays is simple to implement and conforms to the usage of block devices. However, storing multiple variable-length dentries in one block causes false sharing that allows cyclic dependencies, which must be resolved by roll-back/forward and thus significantly complicates the dependency enforcement.

With the byte-addressability of NVM, we re-organize directories using hash tables as shown in Figure 3. The root of a directory points to an array of buckets, each of which points to a list of dentries. A dentry is a fixed-sized structure consisting of four pointers to the filename, the inode, the latest and the consistent next dentry. The filename is externalized to reduce fragmentation, and the hash value and length are stored ahead of the filename for fast comparison. Next pointers point to the next dentry in the hash table. Two next pointers are stored since a dentry can be in two hash tables (*dual views* of the directory) at the same time. The usage of these two next pointers is explained in §3.4.

Hash-table-based directory organization simplifies dependencies in SoupFS. Finer-grained structures used in hash tables avoid false sharing and further the roll-back/forward in the enforcement. Also, since SoupFS allocates a new dentry for each directory insertion, we don't need to track the dependency additionally. As a result, tracking the operation type and a pointer to the added/removed dentry is sufficient for persisting the metadata and enforcing the dependencies for most of the time. Update dependencies are further discussed in §3.3.

A dentry occupies 32B which is usually less than one cache line size. In the implementation, the first few bytes of a filename can be stored together with its corresponding dentry for memory efficiency.

3.2 Content Oblivious Allocator

Some file systems like EXT4 pre-allocate dedicated space for inodes. Dedicating space for inodes facilitates inode management and yields good performance in disk-based file systems. However, it fixes the number of available inodes and incapacitates the file system when inode area is full even though free data blocks are abundant. Such an issue is exacerbated significantly when more data structures are involved, such as the filename and the dentry in SoupFS.

To address this issue, we provide a content-oblivious allocator which treats the whole NVM space as a large memory pool and allocates memory without knowing what the memory is used for. The content-unawareness of the allocator breaks the boundary between various data structures, making the memory management more flexible and simpler without sacrificing performance and correctness.

We also categorize the data structures into two kinds according to the size they use for allocation (see Table 1). As a result, the content-oblivious allocator only needs to manage the memory in page size (4KB) and cache line

Table 1: Data structure sizes in SoupFS

Data Structure	Size	Allocation Size
inode	64B	64B
filename	variable	64B
dentry	32B	64B
hash table (buckets)	4KB	4KB
B-tree node	4KB	4KB
data block	4KB	4KB

size (64B). Filenames, the only variable-length structure, are split into multiple cache lines linked with pointers if a single cache line is not sufficient (see Figure 4).

Metadata of the allocator is stored in the bitmap in NVM, and in-DRAM per-CPU free-lists are used to improve the performance of frequent allocations and deallocations. The implementation of the allocator is derived from ssmalloc [21] and simplified according to two fixed allocation sizes.

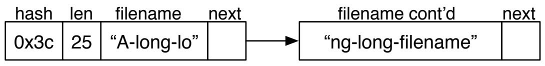


Figure 4: Long filenames in SoupFS

3.3 Update Dependencies

It is seldom a single operation to finish a syscall in file systems. Different data structures are modified in the file system, and the orders of these modifications are dedicatedly arranged for crash consistency. Soft updates summaries these ordering requirements in three rules:

- *C1:* Never point to a structure before it has been initialized, e.g., an inode must be initialized before a directory entry references it.
- *C2:* Never re-use a resource before nullifying all previous pointers to it, e.g., an inode's pointer to a data block must be nullified before that disk block may be re-allocated for a new inode.
- *C3:* Never reset the old pointer to a live resource before the new pointer has been set, e.g., when renaming a file, do not remove the old name for an inode until after the new name has been written.

These three rules are the guidelines of soft updates dependency tracking and enforcement. SoupFS follows *C2* and *C3* and generalizes *C1* which is over-restrictive in most file system operations. Taking the file creation as an example, the new dentry can hold the reference to the new inode even before the initialization of the inode is persisted, as long as the dentry has not been persistently inserted into the hash table in NVM. That is, before the dentry becomes reachable from the root, pointers in the dentry can temporarily violate *C1* without causing any consistency issue. Based on such an observation, we generalize *C1* to be "never let the structure be reachable from the root until it has been initialized," which can further simplify the dependencies in SoupFS.

We then review the update dependencies in different file system operations in SoupFS. For a file creation, a series of operations need to be done as shown in Figure 5.

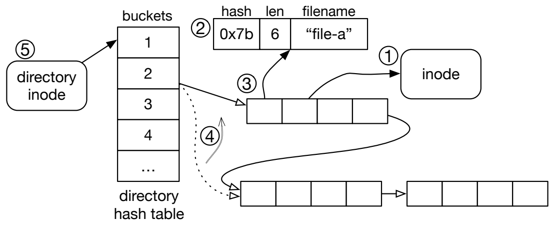


Figure 5: Dependencies of creating a file

① An inode is allocated and initialized with correct information. ② A block of memory is allocated and initialized with the filename. ③ A dentry is allocated and the pointers to the inode and filename are filled. ④ The dentry is inserted into the hash table of the directory. ⑤ The inode of the parent directory is updated. There are several pointers in the above operations. However, the only persistency dependencies we need to guarantee are:

1. *① ② ③ are persisted before the insertion of the dentry is persisted (④).*
2. *The parent directory inode information(⑤) is persisted after the persistence of dentry insertion (④).*

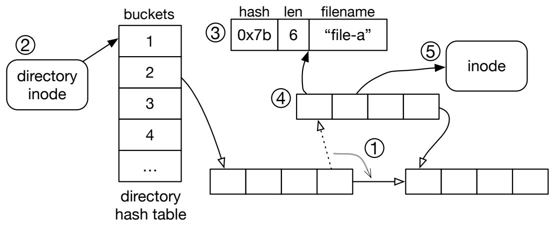


Figure 6: Dependencies of removing a file

For a file removal, the operations are reverted as shown in Figure 6[3]. ① Remove the dentry from the hash table. ② The parent directory inode is modified. ③ The filename can be freed. ④ The dentry can be freed. ⑤ The inode can be freed if its link count is zero. This time, the only ordering requirement is that ② ③ ④ ⑤ *shall be done after the dentry removal (①) is persisted.*

The creation and removal of directories are largely as described above. One difference is that additional operations for hash table construction and destruction are needed. The construction of a hash table includes allocating a page as hash table buckets and erasing it to be all zeros. The destruction is simply freeing the memory since a directory can be removed only if it is empty, i.e. there is no dentry in the hash table. Additionally, we omit the "." dentry since it points the current inode. For the ".." dentry, we simply store it in the inode to avoid other unnecessary dependencies.

B-tree nodes are metadata structures to organize data blocks which contain file data. When a file is enlarged, *the newly written data and metadata structure modifications need to be persisted before the new file size is persisted.* These metadata structure modifications include B-tree height increases, B-tree node allocations and mod-

[3] It is not shown in the figure that if the dentry to remove is the head of the list, the pointer in the corresponding bucket is modified.

ifications, and data block allocations. Adding new data blocks and new inner B-tree nodes to the B-tree cannot be done atomically without copy-on-write. But even if this is not done atomically, it will not cause any problems since all the changes are not visible to users before the update of the file size. The B-tree root and height are stored in the inode and can be persisted atomically.

When a file is truncated, *the reduced file size shall be firstly persisted before reduced file data space is reclaimed.* For efficiency, the reclamations of these space are usually delayed for later file size increases unless the free space in the file systems is nearly exhausted.

3.4 Pointer-based Dual Views

One key property of soft updates is that it ensures that metadata present in memory always reflects the latest state of the file systems (i.e., the latest view) and metadata persisted in disks is consistent (i.e., the consistent view). The *dual views* technique allows delayed persistency and eliminates file system checking after a crash. Soft updates implements *dual views* based on page cache. However, as memory becomes storage, the page cache is removed for performance concerns in most NVMFS.

Thus, to provide *dual views* in NVMFS, a naive approach is to bring back the “cache” for metadata by maintaining another copy of the file system metadata. This approach, however, doubles the memory usage and causes unnecessary memory copies when synchronizing metadata between the latest view and the consistent view. For example, the whole persisted metadata structure has to be copied to its cache before the modification.

To implement efficient *dual views*, we propose pointer-based *dual views* in which most structures are shared by both views and different views are observed by following different pointers. We will then describe how different metadata structures (shown in Table 1) are designed to provide pointer-based *dual views* in SoupFS.

Inodes are already duplicated since VFS has its own data structure (VFS inode) to present an inode. So SoupFS uses its own inodes to store the consistent view and the corresponding VFS inode for the latest metadata.

Filenames are immutable in SoupFS. A filename always co-exists with its dentry and this binding relation never changes before the removal of the dentry. Thus the filename of a dentry can be directly shared by both views.

Dentries are almost read-only. During insertion, a dentry is inserted to the head of the linked list in the corresponding bucket. This procedure modifies no existing dentries. When removing a dentry, its predecessor is modified to point to the next dentry. SoupFS should be aware of such a modification so that it can traverse the list without the removed dentry in the latest view. At the same time, the removed dentry should be still observable in the consistent view if the whole removal has not been persisted. Otherwise, a crash might leave the file system inconsistent when there are multiple not-yet-persisted insertions and removals.

To share dentries in both views, SoupFS stores a pair of next pointers, *latest next* and *consistent next*, in a dentry. With these two next pointers, a traversal in the latest view is done by following the *latest next* pointer if it is non-null. Otherwise, the *consistent next* pointer is used. This guarantees that the latest information is retrieved by following the latest-next-first rule and the consistent view is observed by following only the *consistent next* pointers. Since the *latest next* is also stored in NVM, to differentiate the real *latest next* and the leftover *latest next* of a crash, SoupFS embeds an epoch number in the *latest next*. The epoch number is increased after a crash and *latest next* pointers with old epoch numbers are treated as null. This on-demand checking prevents after-crash stop-the-world checking in which all leftover *latest next* pointers are nullified.

Directory hash table buckets are changed upon an insertion to the dentry list or the removal of the last dentry. To provide two views, we maintain a *latest bucket* for each of the buckets and if not null, it always points to the latest first dentry in the dentry list. A *latest bucket* and its corresponding real bucket together act similarly to the two next pointers in dentries. For convenient memory management, all *latest buckets* for a hash table are gathered together in a volatile page and allocated on demand.

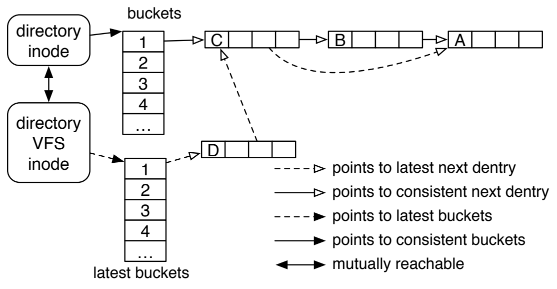


Figure 7: *Dual views* in a directory

An example is shown in Figure 7, in which the latest view can be observed by following dashed arrows and the consistent view is organized by solid arrows. We can also know from Figure 7 that dentry D is recently inserted and B is removed from the directory and both the insertion of D and removal of B have not been persisted yet[4].

B-tree nodes and data blocks: SoupFS focuses on metadata and does not protect consistency of written data, which is the same as soft updates[5]. SoupFS does not provide two views for B-tree nodes and data blocks. Nevertheless, there are still two views of file data in SoupFS.

[4] Since modifications to D and C are directly written in NVM, these changes might have been persisted. But these changes will be ignored after a crash since they are not observable in the consistent view.

[5] Even though soft updates can leverage page cache to provide two copies of data, it cannot guarantee a write spanning two blocks can be persisted atomically.

Table 2: Operations tracked by SoupFS

OP Type	Recorded Data Structures
diradd	added dentry, source directory*, overwritten inode*
dirrem	removed dentry, destination directory*
sizechg	the old and new file size
attrchg	-

One is the latest file data that are available by inspecting the B-tree root and the file size stored in VFS inode. The other is the persisted file data which can be obtained by following the B-tree root and the file size in SoupFS's inode. These two B-tree roots and file sizes form two B-trees that are built on the same set of B-tree nodes and data blocks. However, neither data in two B-trees are guaranteed to be consistent after a crash. To provide data consistency in SoupFS, techniques like copy-on-write can be adopted.

Allocator: The allocation information in NVM bitmap presents the consistent view and in-DRAM free-lists provide the latest view.

3.5 Dependency Tracking

Dependency tracking is one of the key parts of soft updates and is much simplified in SoupFS. Thanks to the byte-addressability of NVM, there are no more cyclic dependencies in the system. We thus can use a DAG to present dependencies among different parts of the file system, according to the paper of soft updates [10]. However, since SoupFS abandons the page cache and the block layer, the gap between the page cache and the file system disappear. In other words, a *persister* can know which operation on which structure needs to be persisted. By endowing with file system semantics, dependency tracking and enforcement are further simplified.

Dependency Tracking Structures: Although soft updates tracks dependencies in byte-level granularity, it is still a block-oriented dependency tracking that uses additional structures to track the dependencies among different blocks. Different from the original soft updates, SoupFS uses an inode-centric semantic-aware way to organize all dependencies.

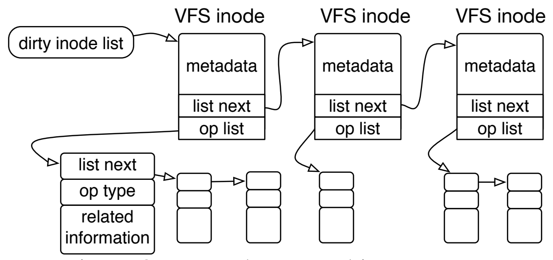


Figure 8: Dependency tracking structures

Figure 8 shows dependency tracking structures in SoupFS. In each VFS inode, an operation list (op list) tracks all recent operations on this inode that are not yet persisted. Each of the operations consists of an operation type (op type) and related information such as pointers to data structures involved during the operation. Table 2 shows the operations SoupFS tracks in detail, in which *diradd* and *dirrem* are used to track in-directory operations, *sizechg* is for regular file structure changes and *attrchg* is for attribute-only updates of an inode. An operation is created and inserted to the operation list of the VFS inode during the syscalls (see §3.6). Once the operation list is not empty, the VFS inode is added to the dirty inode list, waiting to be handled by the persisters.

Tracking these operations is sufficient for dependency enforcement. Supposing a VFS directory inode contains a *diradd*, by checking *added dentry*, SoupFS knows almost everything about the operation, e.g., the filename and the new inode. SoupFS can then persist these operations in the correct order that enforces update dependencies.

3.6 POSIX Syscalls

SoupFS classifies POSIX syscalls into the following categories and handles them accordingly.

Attribute-only Modification: These syscalls, *chmod* and *chown* for instance, only change the attributes of an inode. SoupFS handles these syscalls by directly updating the attributes in the corresponding VFS inode and insert an *attrchg* into the inode's operation list.

Directory Modification: Syscalls like *create*, *mkdir*, *unlink* and *rmdir* modify the content of a parent directory. SoupFS handles these syscalls according to steps described in §3.3. Then it inserts a *diradd* or a *dirrem* to the directory inode's operation list. The affected dentry is recorded as related data in the operation.

File Data Modification: These syscalls might modify the B-tree structures of a regular file. Examples include *write* and *truncate*. The deallocations of nodes are delayed, and the allocations and attachments of B-tree nodes and data blocks are directly done in the B-tree. The new file size and new file root (if changed) are updated only in the latest view (VFS inode). Finally, a *sizechg* is inserted into the inode's operation list.

Rename: *Rename* is special since it can involve more than one directory. Same as soft updates, SoupFS treats *rename* as a combination of a creation in the destination directory and a removal in the source directory. As a result, two operations, *diradd* and *dirrem* are inserted into the operation lists of the destination directory inode and the source directory inode respectively. Soft updates's ordering requirement for *rename* is also adopted by SoupFS, i.e., the persistence of the creation should be completed before the persistence of the removal. To track this dependency, *source directory* and *destination directory* are respectively recorded in *diradd* and *dirrem* as shown in Table 2.

If an existing dentry is overwritten in *rename*, it is directly modified by replacing its inode pointer. To reclaim

* These structures are only for rename.

the original inode in the overwritten dentry, SoupFS records it in *diradd* as the *overwritten inode*.

3.7 Dependency Enforcement

Dependency enforcement is done by daemons called *persisters*. *Persisters* periodically wake up, retrieve all dirty inodes from the dirty inode list, and persist each operation in the operation list according to the ordering requirements. The wake-up frequency can be specified at mount time as the bound of persistence.

It is simple to persist an operation. For *diradd*, a *persister* first ensures the persistence of the allocations and new structures. Then, it reflects the operation in the consistent view by updating the corresponding *consistent pointer* with the *latest pointer*.

For *dirrem*, a *persister* first makes the target data structure persistently removed from the consistent view, then reclaims memory used by the removed data structures.

For *sizechg*, a *persister* can get the newly allocated B-tree nodes and data blocks by inspecting the old and the new file size. Allocations are firstly persisted and then data blocks and B-tree nodes are persisted in a correct order. The file size, the B-tree root and height in the consistent view are updated only after all modifications within the B-tree are persisted. If it is a truncation, the truncated B-tree nodes and data blocks can be reclaimed after the persistence of the new file size, B-tree root and height. As an optimization, the reclamation is delayed for later file size increases.

For *attrchg*, attributes in the VFS inode are persistently written to the consistent inode. The atomicity of these operations is discussed in §3.8.

Finally, the persisted operation is removed from the operation list and the VFS inode is removed from the dirty inode list when its operation list is empty.

In the implementation, we deploy one *persister* for each NUMA node to prevent expensive cross-NUMA memory accesses.

3.8 Atomicity

SoupFS assumes that the failure-atomic write unit of NVM is a cache line, which is no less than 64 bytes. Based on this assumption, there are two kinds of atomic writes used in SoupFS.

The most commonly used one is an atomic update of a pointer, which is done using atomic operations provided by CPU. The other write is persisting an inode. Since the inode size in SoupFS is 64 bytes which is the cache line size, SoupFS needs to prevent the cache line from being evicted before updates to the inode are finished. This is guaranteed by using Intel RTM technology which will hold the cache line in cache until the transaction ends. Per-CPU cache-line-sized journals can be used as fallbacks of RTM to guarantee progress.

Both kinds of atomic writes only guarantee that an update is not partially persisted. It is the *clflush/clflushopt+sfence* that can guarantee the persistence of the update.

3.9 File System Checking

SoupFS is the same as soft updates in file system checking and recovery. Thanks to the consistent view, SoupFS can be instantly used after a crash without having to wait for file system checking or recovery. But in order to collect the memory leaked by the crash, a specialized *fsck* needs to be invoked manually. The *fsck* traverses the whole file system from the root and reconstructs the allocation bitmaps in which all allocated memory is useful.

3.10 Endurance

Although write endurance is not the design goal of SoupFS, we expect better write endurance than other NVMFS, since SoupFS eliminates journaling mechanisms which frequently need to write temporary backup data in NVM. In SoupFS, almost all writes to NVM are to modify the persistent state of the file system. The only exception is updates to the *latest next* pointer in dentries. While storing *latest next* in DRAM can further benefit the endurance, it involves additional data structures to track the relation between dentries and *latest next* pointers, which is complex. Moreover, the negative effect of updating *latest next* pointers is limited since in the implementation the update only happens in removal operations and it is likely to be kept in the cache before the operation is persisted by the persisters.

4. Evaluation

4.1 Experimental Setup

To evaluate the performance of SoupFS, we run micro-benchmarks and macro-benchmarks with a dual-socket Intel Xeon E5 server equipped with NVDIMM. Each 8-core processor runs at 2.3 GHz with four DDR4 channels. There are 48 GB DRAM and 64 GB NVDIMM equipped on the server, and we use one pmem device whose 32GB NVDIMM locate on one NUMA node in the evaluation.

We compare SoupFS against four Linux file systems: EXT4, EXT4 with DAX (EXT4-DAX), PMFS and NOVA. EXT4 can be directly used in Linux 4.9.6, but PMFS and NOVA need simple modifications to run on Linux 4.9.6.

PMFS, NOVA, and SoupFS obtain a range of NVM from kernel driver and manage NVM independently. EXT4-DAX bypasses the page cache using the DAX interface exposed by the persistent memory driver. We evaluate EXT4 only for reference since it cannot guarantee crash consistency in NVM. We provide no comparison with the original soft updates as there is no available soft updates implementation in Linux and simply running

FreeBSD UFS with soft updates on NVM cannot guarantee consistency due to the lack of block write atomicity.

Table 3: Micro-benchmark characteristics

Name	Workload
filetest	(I) create ($10^4 \times 100$) (II) unlink ($10^4 \times 100$)
dirtest	(I) mkdir ($10^4 \times 100$) (II) rmdir ($10^4 \times 100$)

4.2 Micro-benchmarks

We use two single-threaded micro-benchmarks to evaluate the throughput and latency of SoupFS, as shown in Table 3. The benchmarks run 100 iterations and in each iteration, the *filetest* creates 10^4 files in one directory and then deletes all of them. The *dirtest* is similar to the *filetest* but it creates directories instead of files.

Figure 9(a) and 9(b) show the throughput and latency of *create*, *unlink*, *mkdir* and *rmdir* tested using the *filetest* and *dirtest*. Generally, SoupFS performs best in all these tests. It outperforms NOVA in throughput by 43% to 405%, and reduces latency by 30% to 80%. We attribute the performance improvement to the reduction of flushes in the system call path. NOVA also performs well in the tests since it leverages in-DRAM radix trees to manage its directories. However, it still needs logs and cache flush operations to guarantee crash consistency, which causes relatively worse performance than SoupFS. Besides journaling and excessive flush operations, PMFS has high latency and low throughput also because its cost of directory lookup grows linearly with the increasing number of directory entries. This is notable for *create* and *mkdir* since one insertion to the directory needs to scan all existing dentries to find an available slot. For *unlink* and *rmdir*, the latency is very low since our benchmarks delete the files/directories in the same order they are created. If the dentry preceding the to-be-removed dentry is not in use, PMFS will merge those two dentries during the removal. Thus in *unlink* and *rmdir*, PMFS needs to search at most two dentries to find the dentry to remove, yielding low latencies as shown in the figure. EXT4 and EXT4-DAX leverage hashed B-trees to speed up directory accesses, thus they achieve better performance than PMFS.

Figure 9(c) shows the latency distribution for *create* in the *filetest*. Latencies longer than 30us are not shown in the figure for clarity. The result proves the average latencies shown in Figure 9(b). Most of the latencies for SoupFS locate at around 3us and latencies for NOVA at around 4us. Due to the inefficient directory organization, latencies for PMFS evenly distribute and steadily rise as the number of files in a directory increases.

Table 4: Filebench workload characteristics

Workload	Average file size	# of files	I/O size	r:w ratio
Fileserver	128KB	10000	1M	1:2
Fileserver-1K	1KB	10000	1M	1:2
Webproxy	16KB	10000	16K	5:1
Varmail	16KB	5000	1M	1:1

4.3 Macro-benchmarks

We evaluate the performance of SoupFS for real world applications by running a set of macro-benchmark workloads, including Filebench and Postmark.

Filebench: Filebench is a file system benchmark that simulates a large variety of workloads by specifying different models. We integrate the recent fixes to Filebench by Dong et al. [7] to make our evaluation more accurate. Table 4 shows the characteristics of Filebench workloads. We run these benchmarks from 1 to 20 threads multiple times and report the average to show the throughput and scalability. The coefficient of variation is 1.8% in average.

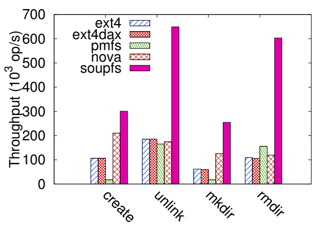
As shown in Figure 11(a) to 11(d) SoupFS performs best in general. The performance drop after eight threads is caused by the NUMA architecture. When the number of threads exceeds eight, either the threads contend on eight cores of a NUMA node, or there are a lot of cross-NUMA memory accesses. We thus evaluate with Filebench bound to one NUMA node and report the result in Figure 12(a) to 12(d), in which the throughput still cannot scale well, but the performance drop disappears.

The throughput of fileserver is lower than those of other Filebench workloads. This is because the default average file size is 128KB, causing each operation to write more data and the data write speed dominates. SoupFS performs slightly better in this workload since it provides two views of the file size so that it does not need to persist the B-tree structures immediately. As a drawback, the file data are not guaranteed to be persisted after the *write* syscall, which is different from other NVMFS. We also evaluate fileserver with 1K file size to highlight the metadata operations (Figure 11(b)). The throughput of all file systems increases and SoupFS outperforms NOVA by up to 89% and PMFS by up to 47%.

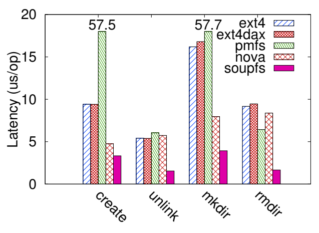
The webproxy involves recreating and reading several files in a directory with many files. PMFS performs worst due to its inefficient directory access, while other file systems, by using hash tables (SoupFS), radix trees (NOVA) and hashed B-trees (EXT4 and EXT4-DAX), perform much better. SoupFS performs slightly better when there are fewer threads because of metadata operations like file removals and creations.

The varmail acts as a mail server on which users read, remove, reply, and write mails. In this workload, *fsync* operations eliminate the benefit of page cache in EXT4 and the performance of PMFS is limited by its slow directory design. SoupFS outperforms NOVA by up to 75% due to fast metadata operations.

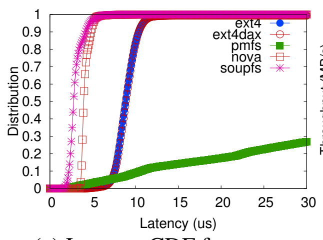
Postmark: Postmark is a benchmark to simulate mail servers. We enlarge the number of transactions to 10^6 in the default single-threaded Postmark configuration to test the performance. Figure 10 shows that SoupFS outperforms other file systems by about 50%.



(a) Throughput



(b) Latency



(c) Latency CDF for *create*

Figure 9: Throughput and latency of file system operations

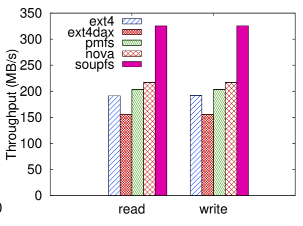
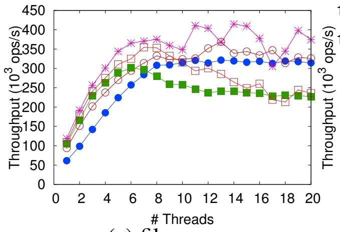
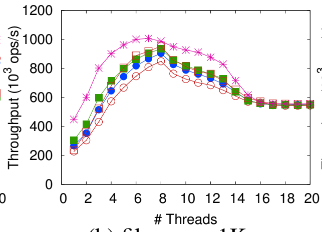


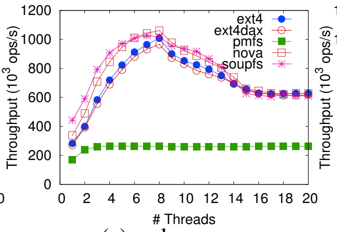
Figure 10: Postmark



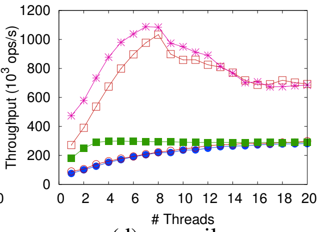
(a) fileserver



(b) fileserver-1K

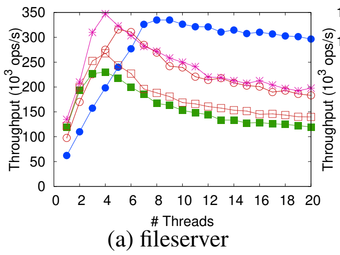


(c) webproxy

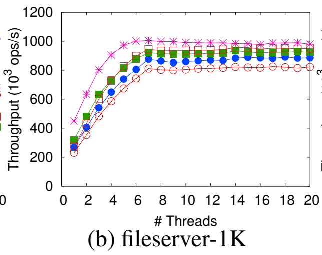


(d) varmail

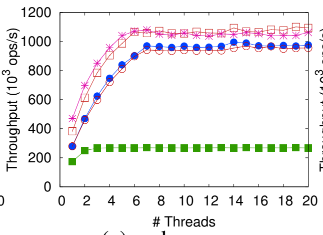
Figure 11: Throughput of Filebench (EXT4 does not guarantee correctness)



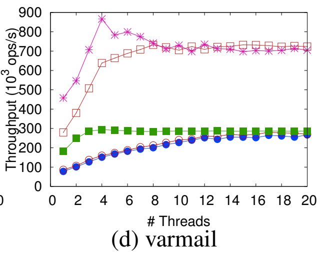
(a) fileserver



(b) fileserver-1K



(c) webproxy



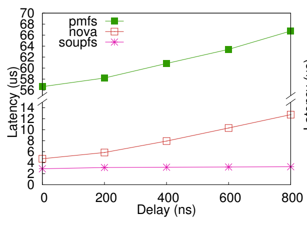
(d) varmail

Figure 12: Throughput of Filebench with NUMA binding (EXT4 does not guarantee correctness)

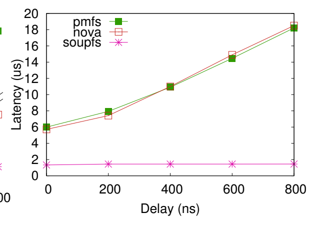
4.4 Sensitivity to NVM Characteristics

Different NVM technologies have different write latencies. The NVDIMM we use has the same performance as DRAM; however, NVM built by PCM and 3D XPoint is expected to have higher latency especially higher write latency than DRAM. We roughly evaluate the sensitivity to NVM write latency of different file systems by inserting delays after each *clflush* instruction.

Figure 13 shows the latency of *create* and *unlink* in the *filetest* micro-benchmark with different delays inserted after *clflush*. In both cases, the latency of SoupFS remains unchanged with increasing delays due to the elimination of cache flushes in the critical path. The latency of PMFS and NOVA increases because they need cache flushes for crash consistency during the syscall. Increasing the delay from 0 to 800ns, the latency of NOVA increases 8us which is nearly 200% of its original value for *create* (Figure 13(a)). The increased value matches our estimation in §2.2. Although the increased latency for PMFS is similar, the *create* latency of PMFS is still dominated by the slow directory lookup performance, so the relative influence is not significant. For *unlink* in Figure 13(b), both NOVA and PMFS are affected by the clflush delays, with latency increased from 6us to 18us.



(a) create



(b) unlink

Figure 13: Latency of *filetest* with different clflush delays

clflushopt, the asynchronous and parallel version of *clflush*, is not available on our evaluation platform. We thus give no evaluation of *clflushopt*. However, since most of the cache flushes in the file systems are for persistency ordering guarantee, file systems usually use a combination of *clflushopt*+*sfence* which neutralizes the merit of asynchronism and parallelism. Other approaches, like running instructions between *clflushopt* and *sfence* or using existing atomic operations to replace fences, are feasible but they can only be used under certain conditions where appropriate instructions and existing atomic operations are available. *clwb*, which is not available on our platform either, is similar to *clflushopt* but keeps the data in the cache after persisting it. However, the merit of *clwb* can barely bring performance improvement since existing NVMFS are designed to avoid re-accessing flushed cache lines in one syscall.

We would also like to compare the impact of different NVM bandwidth on the performance of SoupFS. Unfortunately, we fail to change the BIOS configuration of the PCIE extended area as suggested by Sengupta et al. [37], as it is inaccessible to a normal user. We thus leave such a comparison as future work. Yet, since the delayed persistency of SoupFS has fewer requirements for immediately persisting the data which urgently need bandwidth, we envision that SoupFS would gain more benefits compared to other alternatives like PMFS and NOVA that require synchronous flushes.

5. Related Work

Metadata update approaches: Other than soft updates, there have been various approaches to preserving metadata consistency, including shadow paging [14, 18, 30], log-structuring [17, 19, 24, 31, 33, 34], journaling [2, 12, 39] and WriteAtomic [29]. There have been various other ways to represent write ordering, using backpointers [3], transactional checksums [28], patches [9]. For example, NoFS [3] proposes backpointers to reduce ordering requirement when persisting data. It, however, requires adding a backpointer to not only metadata but also data, which increases storage overhead. Moreover, a key assumption is that a backpointer and its data or metadata are persisted atomically, a property not available in NVM.

NVM-aware file systems: Some designs have considered using NVM to accelerate metadata update performance. For example, Network appliance's WAFL [14] leverages NVRAM to keep logs to improve synchronous log update performance. The Rio cache [27] enabled by uninterruptible power supply can also be used to log synchronous metadata updates with high performance. However, even with NVM as cache, they may still suffer from consistency issues from unanticipated cache line eviction for metadata updates.

The promising feature of NVM has stimulated the design and implementation of several recent NVM file systems such as BPFS [4], SCMFS [44], Aerie [41], EXT4-DAX [5, 6], NOVA [45] and HiNFS [25]. Generally, they allow "execute in place" (XIP) to bypass the block layer and page cache to reduce management overhead, or provide a buffer cache with in-place commit feature [20].

Wu and Zwaenepoel describe eNVy [43], a storage system that directly presents flash memory as a linear address space into memory bus using paging translation. To overcome slow write performance of flash memory, eVNy uses a small battery-backed SRAM as a buffer to create a copy of the updated page to give the illusion of in-place update. As the NVM nowadays could achieve within one order of magnitude speed of DRAM, SCMFS [44] and SIMFS [38] further directly map a file data space into the virtual address space of a process. These techniques are orthogonal to the design of SoupFS.

Data structures for NVM: Venkataraman et al. [40] describe a persistent B+ tree implementation for NVM, yet requires synchronous flushes at each update path. NV-Tree [46] instead uses DRAM as indexes to reduce synchronous flushes cost, but requires scanning all NVM in order to reconstruct the indexes upon a crash. Mnemosyne [42] provides a transactional interface for consistent updates of application data structures. SoupFS eliminates cache flushes in the critical path of file system operations and need no journaling for crash consistency.

Crash consistency and memory persistency models: Chidambaram et al. [2] propose separating ordering from durability and introduce optimistic crash consistency by leveraging a hypothetical hardware mechanism called asynchronous durability notification (ADN). SoupFS can be made simple and efficient with ADN by avoiding flushing already persistent cache lines.

Foedus [15] leverages the duality of volatile pages and stratified snapshot pages to provide snapshots and crash consistency in an NVM-based in-memory database. Most of the pointers in Foedus are dual-page-pointers stored together. SoupFS uses a similar technique like "dual pointers" to present *dual views* of the file system metadata in some structures like dentries. However, the latest pointers for the latest view may be created on demand and stored separately from the consistent pointer in SoupFS.

Pelley et al. [26] introduce the concept of memory persistency as an analogy of memory consistency, summarize a set of persistency models such as strict and epoch persistency and additionally introduce strand persistency. Kolli et al. [16] further describe a set of techniques like deferring commit until lock release to different persistency models to relax write orderings for transactions whose read/write sets are known in advance. Unlike prior work, SoupFS extends soft updates instead of logging for ensuring the persistency models.

6. Conclusions

This paper describes SoupFS, a soft updates implementation for NVM. SoupFS is made simple by leveraging byte-addressability to simplify dependency tracking and enforcement. SoupFS is made fast through delaying most synchronous flushes from the critical path thanks to the efficient pointer-based *dual views*. Evaluations show that SoupFS outperforms state-of-the-art NVMFS.

Acknowledgment

We thank our shepherd Liuba Shrira and the anonymous reviewers for their constructive comments. This work is supported in part by National Key Research & Development Program of China (No. 2016YFB1000104), China National Natural Science Foundation (No. 61572314, 61525204 and 61672345), Zhangjiang Hi-Tech program (No. 201501-YP-B108-012) and CREATE E2S2.

References

[1] V. Aurora. Soft update, hard problems. https://lwn.net/Articles/339337/, 2009.

[2] V. Chidambaram, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Optimistic crash consistency. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 228–243. ACM, 2013.

[3] V. Chidambaram, T. Sharma, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Consistency without ordering. In *FAST*, page 9, 2012.

[4] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. Better i/o through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 133–146. ACM, 2009.

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

[6] Support ext4 on nv-dimms. http://lwn.net/Articles/588218/, 2014.

[7] M. Dong, Q. Yu, X. Zhou, Y. Hong, H. Chen, and B. Zang. Rethinking benchmarking for nvm-based file systems. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems*, pages 20:1–20:7, 2016.

[8] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems*, 2014.

[9] C. Frost, M. Mammarella, E. Kohler, A. de los Reyes, S. Hovsepian, A. Matsuoka, and L. Zhang. Generalized file system dependencies. In *SOSP*, pages 307–320. ACM, 2007.

[10] G. R. Ganger, M. K. McKusick, C. A. Soules, and Y. N. Patt. Soft updates: a solution to the metadata update problem in file systems. *ACM Transactions on Computer Systems (TOCS)*, 18(2):127–153, 2000.

[11] G. R. Ganger and Y. N. Patt. Metadata update performance in file systems. In *Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*. USENIX Association, 1994.

[12] R. Hagmann. Reimplementing the cedar file system using logging and group commit. In *SOSP*. ACM, 1987.

[13] V. Henson. Khb: A filesystems reading list. https://lwn.net/Articles/196292/, 2006.

[14] D. Hitz, J. Lau, and M. A. Malcolm. File system design for an nfs file server appliance. In *USENIX winter*, volume 94, 1994.

[15] H. Kimura. Foedus: Oltp engine for a thousand cores and nvram. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 691–706. ACM, 2015.

[16] A. Kolli, S. Pelley, A. Saidi, P. M. Chen, and T. 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*, pages 399–411. ACM, 2016.

[17] R. Konishi, Y. Amagai, K. Sato, H. Hifumi, S. Kihara, and S. Moriai. The linux implementation of a log-structured file system. *ACM SIGOPS Operating Systems Review*, 40(3):102–107, 2006.

[18] E. Kustarz. Zfs-the last word in file systems. *http://www.opensolaris.org/os/community/zfs/*, 2008.

[19] C. Lee, D. Sim, J. Hwang, and S. Cho. F2fs: A new file system for flash storage. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 273–286, 2015.

[20] E. Lee, H. Bahn, and S. H. Noh. Unioning of the buffer cache and journaling layers with non-volatile memory. In *USENIX Conference on File and Storage Technologies*, pages 73–80, 2013.

[21] R. Liu and H. Chen. Ssmalloc: a low-latency, locality-conscious memory allocator with stable performance scalability. In *Proceedings of the Asia-Pacific Workshop on Systems*, page 15. ACM, 2012.

[22] K. McKusick. Journaling soft updates. In *BSDCan*, 2010.

[23] M. K. McKusick, G. R. Ganger, et al. Soft updates: A technique for eliminating most synchronous writes in the fast filesystem. In *USENIX Annual Technical Conference, FREENIX Track*, pages 1–17, 1999.

[24] C. Min, K. Kim, H. Cho, S.-W. Lee, and Y. I. Eom. Sfs: random write considered harmful in solid state drives. In *FAST*, page 12, 2012.

[25] J. Ou, J. Shu, and Y. Lu. A high performance file system for non-volatile main memory. In *European Conference on Computer Systems*, 2016.

[26] S. Pelley, P. M. Chen, and T. F. Wenisch. Memory persistency. In *ISCA*, pages 265–276. ACM, 2014.

[27] C. PM, N. WT, S. Chandra, C. Aycock, G. Rajamani, and D. Lowell. The rio file cache: Surviving operating system crashes. In *ASPLOS*, 1996.

[28] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Iron file systems. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, pages 206–220. ACM, 2005.

[29] V. Prabhakaran, T. L. Rodeheffer, and L. Zhou. Transactional flash. In *OSDI*, pages 147–160, 2008.

[30] O. Rodeh, J. Bacik, and C. Mason. Btrfs: The linux b-tree filesystem. *ACM Transactions on Storage (TOS)*, 9(3):9, 2013.

[31] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52, 1992.

[32] A. M. Rudoff. Deprecating the pcommit instruction. https://software.intel.com/en-us/blogs/2016/09/12/deprecate-pcommit-instruction, 2016.

[33] S. M. Rumble, A. Kejriwal, and J. Ousterhout. Log-structured memory for dram-based storage. In *Proceed-*

ings of the 12th USENIX Conference on File and Storage Technologies (FAST 14), pages 1–16, 2014.

[34] M. Seltzer, K. Bostic, M. K. McKusick, and C. Staelin. An implementation of a log-structured file system for unix. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, pages 3–3. USENIX Association, 1993.

[35] M. I. Seltzer, G. R. Ganger, M. K. McKusick, K. A. Smith, C. A. Soules, and C. A. Stein. Journaling versus soft updates: Asynchronous meta-data protection in file systems. In *USENIX Annual Technical Conference, General Track*, pages 71–84, 2000.

[36] M. I. Seltzer, G. R. Ganger, M. K. McKusick, K. A. Smith, C. A. N. Soules, and C. A. Stein. Journaling versus soft updates: Asynchronous meta-data protection in file systems. In *Usenix ATC*, 2000.

[37] D. Sengupta, Q. Wang, H. Volos, L. Cherkasova, J. Li, G. Magalhaes, and K. Schwan. A framework for emulating non-volatile memory systemswith different performance characteristics. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, pages 317–320. ACM, 2015.

[38] E. Sha, X. Chen, Q. Zhuge, L. Shi, and W. Jiang. Designing an efficient persistent in-memory file system. In *IEEE Non-Volatile Memory System and Applications Symposium*, pages 1–6, 2015.

[39] Silicon Graphics International Corp. Xfs: A high-performance journaling file system. http://oss.sgi.com/projects/xfs, 2012.

[40] S. Venkataraman, N. Tolia, P. Ranganathan, R. H. Campbell, et al. Consistent and durable data structures for non-volatile byte-addressable memory. In *FAST*, volume 11, pages 61–75, 2011.

[41] H. Volos, S. Nalli, S. Panneerselvam, V. Varadarajan, P. Saxena, , and M. M. Swift. Aerie: Flexible file-system interfaces to storage-class memory. In *Proceedings of the Ninth Euro- pean Conference on Computer Systems (EuroSys 14)*, page 14:114:14, 2014.

[42] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. *ACM SIGPLAN Notices*, 46(3):91–104, 2011.

[43] M. Wu and W. Zwaenepoel. envy: A non-volatile, main memory storage system. In *ASPLOS*, 1994.

[44] X. Wu and A. Reddy. Scmfs: a file system for storage class memory. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 39. ACM, 2011.

[45] J. Xu and S. Swanson. Nova: A log-structured file system for hybrid volatile/non-volatile main memories. In *USENIX Conference on File and Storage Technologies*, pages 323–338, 2016.

[46] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He. Nv-tree: Reducing consistency cost for nvm-based single level systems. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 167–181, 2015.

[47] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi. Kiln: Closing the performance gap between systems with and without persistence support. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 421–432. ACM, 2013.