A File System Bypassing Volatile Main Memory: Towards A Single-Level Persistent Store

Deng Zhou
San Diego State University
San Diego, CA
dzhou.ustc@gmail.com

Wen Pan, Tao Xie
San Diego State University
San Diego, CA
{wpan,txie}@sdsu.edu

Wei Wang
Micron Technology Inc.
Milpitas, CA
wwanga@micron.com

ABSTRACT

Existing persistent memory (PM) based file systems rely on a DRAM and PM hybrid store. Although a hybrid store does boost system performance while avoiding some current PM limitations like limited endurance, we envision that with more advances PM technologies could provide applications with a single-level persistent store in the not-so-distant future. As a first step to explore this direction, in this paper we design, implement, and evaluate a new persistent memory file system called SPFS (Single-level Persistent File System), which completely bypasses conventional DRAM-based volatile main memory. Unlike all existing PM-based file systems, SPFS never leverages DRAM to manage its metadata. Thus, redundant copies of metadata in volatile main memory (e.g., a copy of an inode in DRAM) and data movements between the two memories (e.g., copying an inode from PM to DRAM) can be totally eliminated. The goal of this paper is to explore how to manage files and their metadata with guaranteed data consistency on PM without the support of DRAM, which makes a first step towards the ultimate success of a single-level persistent store. Our experimental results demonstrate that SPFS outperforms traditional DRAM-based in-memory file systems *ramfs* and *tmpfs* in most cases. Besides, its performance is only moderately worse than that of NOVA, a state-of-the-art PM-based file system.

CCS CONCEPTS

• **Software and its engineering** File systems management;

KEYWORDS

file system, persistent memory, single-level persistent store, pool-based metadata management, metadata dual-copy

ACM Reference Format:
Deng Zhou, Wen Pan, Tao Xie, and Wei Wang. 2018. A File System Bypassing Volatile Main Memory: Towards A Single-Level Persistent Store. In *CF '18: CF '18: Computing Frontiers Conference, May 8–10, 2018, Ischia, Italy.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3203217.3203277

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CF '18, May 8–10, 2018, Ischia, Italy
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5761-6/18/05...$15.00
https://doi.org/10.1145/3203217.3203277

1 INTRODUCTION

Persistent memory (PM, also called storage class memory or SCM [22]) is defined as a byte-addressable non-volatile memory (NVM) that can be directly connected to the main memory bus [6]. Emerging PM technologies include 3D XPoint [3], phase change memory (PCM) [10], spin-transfer torque memory (STT-RAM) [11], and resistive memory (RRAM) [13]. They possess some desirable features like byte-addressability, high-capacity, low cost as well as low energy-consumption compared to DRAM, and latencies close to that of DRAM [6][5]. For example, the latencies of PCM for read and write reach 50 *ns* and 150 *ns*, respectively [16]. In January 2016, Intel revealed a 6 TB 3D XPoint DDR4 DIMM, which has 1,000 times greater endurance than NAND flash and is 10 times denser than DRAM [7]. In a nutshell, they exhibit a huge potential to become a viable DRAM alternative.

To fully exploit the low latency and byte-addressability of PM, much effort has been recently made to build a DRAM and PM hybrid store where a PM device is placed side-by-side with DRAM on the memory bus and can be accessed by a CPU through normal load/store instructions [22][6][5][16][15][19][23]. In such a hybrid store, most modules of an operating system (e.g., the virtual memory manager) are still running on a DRAM system while a PM-based file system is managing user data on a PM device [22][6][5][23]. The two memories share a single address space but their space and data are separately managed. Fig. 1a shows the architecture of a hybrid store. Prior studies [22][6][5][4][23] show that a hybrid store can greatly boost the performance of a PM-based file system as it allows applications to fully exploit the complementary features of DRAM and PM in terms of performance, durability, scalability, and energy consumption. For example, in a hybrid store all existing PM-based file systems leverage DRAM to manage their metadata so that two major limitations of current PM, a relatively low speed and limited endurance, can be avoided. In addition, there is no need to maintain metadata consistency in DRAM.

However, DRAM is starting to hit the density and power ceiling due to technology scaling challenges [19][17]. Limited capacity and energy inefficiency make it hard for DRAM to meet the fast growing needs of data-intensive applications where TB-scale data are generated and stored [21]. On the other hand, PM technologies are generally more energy-efficient and denser than DRAM. Moreover, they still have considerable room to further improve in various respects (e.g., endurance [5]). Therefore, it is reasonable to speculate that in the not-so-distant future PM could provide applications with a single-level persistent store where only a PM device is used to serve as main memory [14]. The structure of a single-level persistent store is depicted in Fig. 1b. Although a hybrid store already demonstrated its efficacy through the excellent performance of existing PM-based

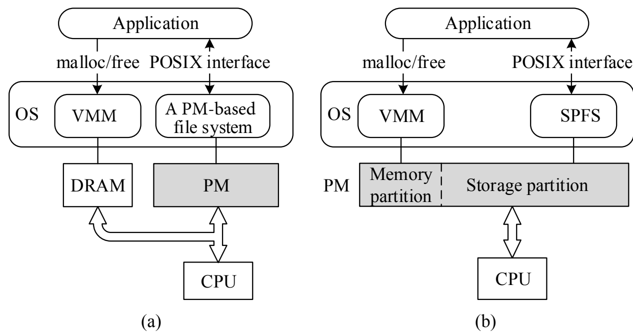


Figure 1: (a) Hybrid; (b) single-level persistent.

file systems [6][23], we argue that it is worth exploring the development of a single-level persistent store now. In fact, compared with its hybrid counterpart, a single-level persistent store has some desirable advantages. First, redundant copies of metadata in DRAM become unnecessary. Existing PM-based file systems [22][6][5][23] generate redundant copies of metadata. For example, when a file is opened a copy of its inode is created in DRAM to maintain its status [23], an old fashion designed for traditional two-level storage model (i.e., DRAM-based main memory and disk-based secondary storage). However, the inode of the file actually already exists in a PM device, which can also be directly accessed by the CPU. Eliminating metadata redundancy not only saves memory space but also simplifies their maintenance. Second, a single-level persistent store has a higher energy efficiency. This is mainly because DRAM is energy inefficient due to its inherent periodic refresh mechanism. In addition, moving metadata between DRAM and PM wastes energy. A study shows that a single-level persistent store can improve energy efficiency by eliminating the instructions and data movement [14]. Third, a single-level persistent store can quickly recover from a software failure as it can correct the software error without reinitializing the system-wide memory space. Fast software failure recovery is essential for a computing node in a cloud service or a data center because failed nodes could result in a degraded level of service. On the contrary, a hybrid store usually requires a long warming up period as it needs to reinitialize the entire DRAM space and all OS data structures.

The single-level persistent store architecture, however, changes the assumptions that motivated the design of current operating systems, and thus, requires a new design of OS [4]. Designing a completely new OS devoted to a single-level persistent store is our long term goal, which is out of the scope of this research. In this paper, we design, implement, and evaluate a new file system called SPFS (Single-level Persistent File System), which is dedicated to a single-level persistent store. Since a file system is an integral part of an OS, SPFS makes a first step towards our long term goal. Unlike all existing PM-based file systems [22][6][5][23], SPFS completely bypasses the traditional DRAM-based volatile main memory. It never leverages DRAM to manage its metadata. Rather, all operations on its metadata are carried out only on a PM device. For compatibility reasons, SPFS also adopts the POSIX file system interfaces so that all existing applications can work on it. However, unlike all existing PM-based file systems [22][6][5][23], SPFS assumes that there is only one underlying PM device on which both the other parts of an OS and itself are running. The single PM device is logically split into two partitions during a system boot. The memory partition is used as a main memory and the storage partition is considered as a storage device for a file system (see Fig. 1b). The capacities of the two partitions can be dynamically tuned based on applications' requirements. To fully exploit the random access capability and byte-addressability of PM, SPFS employs a pool-based metadata management strategy (see Section 4.2). To ensure metadata consistency and reliability, SPFS maintains two copies of each file system metadata (see Section 4.3). For data consistency, SPFS leverages the copy-on-write (CoW) scheme (see Section 4.3). SPFS targets single-node systems. We implement SPFS in Linux 3.14 kernel and evaluate it under two benchmarks.

The rest of this paper is organized as follows. Section 2 briefly introduces the related work and the motivation. The design and implementation details are presented in Section 3 and Section 4, respectively. Section 5 evaluates the performance of SPFS. Section 6 concludes this research.

2 BACKGROUND AND MOTIVATION

Conventional in-memory file systems can be generally categorized into two camps: DRAM-based [1][2][20] and PM-based [22][6][5][23]. *ramfs* is a simple file system that exports Linux's disk caching mechanisms (e.g., page cache) as a dynamically resizable RAM-based file system [2]. *ramdisk*, on the other hand, emulates a disk drive by using the normal DRAM in main memory [1]. Rather than using dedicated physical memory such as a "RAM Disk", *tmpfs* uses the operating system page cache for file data [20]. It is a file system based on SunOS virtual memory resources.

With the advances of PM, four PM-based file systems have been proposed in chronological order: BPFS [5], SCMFS [22], PMFS [6], and NOVA [23]. Condit *et al.* proposed BPFS, which uses a technique called short-circuit shadow paging to provide atomic, fine-grained updates to persistent storage [5]. There is no kernel level implementation of BPFS. Wu and Reddy developed SCMFS, which is directly built on virtual memory space and lays out files as large contiguous virtual address ranges [22]. However, it lacks of a mechanism to ensure data/metadata consistency [22][23]. PMFS developed by Dulloor *et al.* manages a PM device completely independent of the operating system's virtual memory management [6]. It employs the conventional block file system data layout and management scheme for a PM device [6]. NOVA, a state-of-the-art PM-based file system, extends traditional log-structured file system techniques to exploit the characteristics of a hybrid memory system [23]. Xu and Swanson demonstrated that NOVA outperforms PMFS in almost all cases [23]. All of them except BPFS [5] maintain the traditional POSIX file system interface. They all exhibited significant performance improvements compared with either traditional DRAM-based in-memory file systems [22] (e.g., *ramfs* [2] and *tmpfs* [20]) or block file systems like Windows NTFS on a RAM disk [5] and Linux Ext4 on an Intel PMDB (persistent memory block driver) platform [6][23]. NOVA [23] is the only one that has ever been compared with one of its peers. Only PMFS [6] and NOVA [23] are open source.

All existing PM-based file systems [22][6][5][23] target a hybrid store so that the complementary benefits of DRAM and PM can be

reaped. They all shed new light on how to exploit the characteristics of PM to boost file system performance. In this research, however, we explore a different route. Instead of developing one more hybrid store supported PM-based file system whose performance can beat all existing ones, the enormous potential of a single-level persistent store and its desirable advantages motivate us to implement a file system dedicated to a single-level persistent store, which is a first step towards our long term goal. To the best of our knowledge, SPFS is the first in its kind.

3 SPFS DESIGN OVERVIEW

In a traditional computer system (e.g., a hybrid store system employed in [22][6][5][23]), all kernel data structures (e.g., process related data structures, file system related data structures, etc.) stay in the DRAM. In particular, Linux employs a virtual file system (VFS) to manage all file system related data structures including page caching, file system metadata (e.g., superblock, inodes, bitmap), and dentry, which is created on-the-fly to facilitate a file search. In an existing PM-based file system like PMFS [6] or NOVA [23], page caching has been eliminated as the CPU can directly access files. However, its file system metadata still have a copy in DRAM. Whenever there is an update on a metadata copy in DRAM, the system has to synchronize the update to the metadata on PM. In addition, dentry has to be created again in DRAM after each reboot. Thus, the system has a long warming up period before it can provide service.

Our first design choice is to transfer file system metadata management from DRAM to PM. Thus, all file system related data structures, which are originally maintained in DRAM, are now managed in PM. We believe that decoupling a file system from DRAM-based main memory is an initial step towards a new OS dedicated for a single-level persistent store where DRAM does not exist. Since all metadata are managed in-place within the file system, a DRAM copy is no longer needed. The I/O path is also simplified as the DRAM layer is not needed to manage file system metadata. Besides, dentry does not have to be created on-the-fly after each reboot as it now stays in PM. This design choice allows the system to have a shorter warming up period as tasks such as loading file system metadata to DRAM and creating dentry on-the-fly can be skipped.

Our second design choice is to use a partition of a PM device (i.e., *memory partition* shown in Fig. 1b) to hold all non-VFS modules of an OS (e.g., process management, main memory management, etc.) and their data structures as DRAM no longer exists in a single-level persistent store. The main consideration is to reuse all existing non-VFS modules of an OS so that the modifications of an existing OS can be minimized. Note that all these non-VFS modules treat the memory partition as DRAM. Similar to an existing computer system, all these modules (i.e., their executable files and configure files) are initially stored in the *storage partition* of the same PM device. During a system reboot, theses modules are loaded into the memory partition and then build their data structures on-the-fly there. If any of them fails or runs into an error, a system reboot takes place to re-initialize and re-build all data structures of these modules, just like a current computer system does. Thus, there is no need to maintain a consistent status for their related data structures. When there are free spaces in the storage partition, the capacity of the memory partition can be increased in real-time based on an application's needs, which offers users a great flexibility in system configuration.

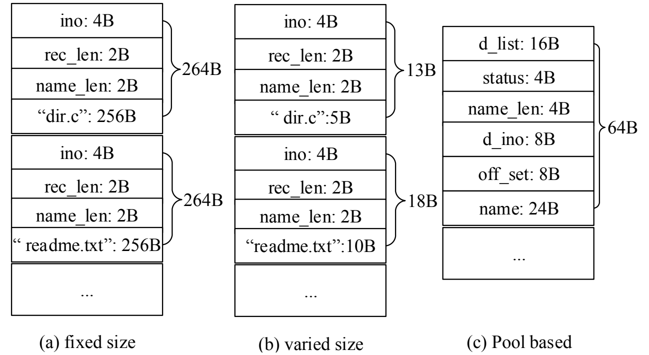


Figure 2: The structure of dentry.

Our third design choice is to abandon traditional block-oriented metadata management techniques. We find that there are two problems of a block-oriented on-disk layout. First, conventional directory management techniques are inefficient. For a traditional Unix/Linux file system, there are two design options for dentry management. The first one uses a fixed size entry that can accommodate the longest file name (e.g., Ext4 allows up to a 255-byte long file name) for all files (see Fig. 2a). Clearly, this option could waste memory space when most files have a short name. The second option employs a flexible entry size scheme where each entry has a variable (i.e., *rec_len* in Ext2) to record its size (see Fig. 2b). Although it can save space, it demands a full scan of all data blocks of a directory, which is time-consuming. Second, space reservation techniques used in current metadata management fail to dynamically respond to the changes of workloads. For instance, it is difficult to re-configure a formatted volume/partition. In order to fully exploit the random access capability and byte-addressability of PM, SPFS employs various pools to manage file system metadata. The sizes of the pools can be dynamically adjusted whenever needed. As a result, SPFS dentry data structure design (see Fig. 2c) does not have the problems associated with Ext2 and Ext4 dentry management (see Section 4.2). We argue that it is more appropriate to use a memory management method rather than a block-oriented technique to manage file system metadata in PM. Although pool-based memory management [12] is not new, we are the first to use it to manage filesystem metadata.

4 SPFS IMPLEMENTATION

We implement SPFS in Linux kernel 3.14.15 of x86_64 architecture. It has about 8,000 lines of C code. In this section, we first explain how SPFS manages space. Next, we describe its metadata management strategy followed by data and metadata consistency mechanisms.

4.1 Space Management

SPFS employs zones to manage its space. However, unlike a traditional Linux zone technique, the number of zones in SPFS can be configured initially and each zone is acting as a partition for data isolation. Since the buddy allocator used in a Linux kernel has proved to be an efficient space allocator, SPFS uses it to manage its free space. When a write request arrives, if its size is smaller than 2

MB SPFS always allocates a data block that contains n 4-KB pages, where n is a power of 2. Note that the size of the newly allocated data block should be the smallest value that is no less than the request's size. For example, assume that the size of the request is 12 KB. SPFS tries to allocate a 16 KB data block (i.e., 4 4-KB pages) for it. If a write request's size is larger than 2 MB, SPFS will allocate multiple 2-MB data blocks to accommodate it. A volume is a file system instance that has its own metadata and namespace. In our design, while one SPFS volume has to be located within a particular zone, one zone can have multiple SPFS volumes. Each volume is essentially a virtualized file system.

4.2 Metadata Management

Each file system instance/volume maintains a series of pools including an inode pool, multiple name pools, and a number of extent node pools. SPFS utilizes a uniform pool structure for all metadata and each pool can be configured during the initialization process. The structure of a pool is shown in Fig. 3. All free metadata entries are indexed by free lists. Thus, allocating or freeing a metadata entry (hereafter, meta entry) can be finished in *O(1)* time. Each pool contains a spectrum of variables to record its type ("Entry type", e.g., inode), its size ("Entry size", the size of meta entries), the total number of meta entries ("Total count"), the number of free meta entries ("Free count"), an exclusive lock ("Pool lock") for serializing update operations on these pool-level variables (e.g., "Total count"), a pointer to the currently active block ("Active block index"), and a pointer to the head of the block list ("Block list head"). Since files that are created and have been accessed together will have a high possibility to be accessed together again in the future, SPFS tries to allocate the metadata of these files to the same block, which improves the TLB (translation lookaside buffer) hit rate. Therefore, in each pool structure SPFS maintains a pointer called "Active block index", which always points to a block with free meta entries.

After a block is allocated, it is inserted into the block list. Each block has a block head, which maintains multiple block-level variables. While "Total entry account" logs the total number of meta entries within a block, "Free entry count" records the number of free meta entries within a block. "Block entry type" is inherited from the

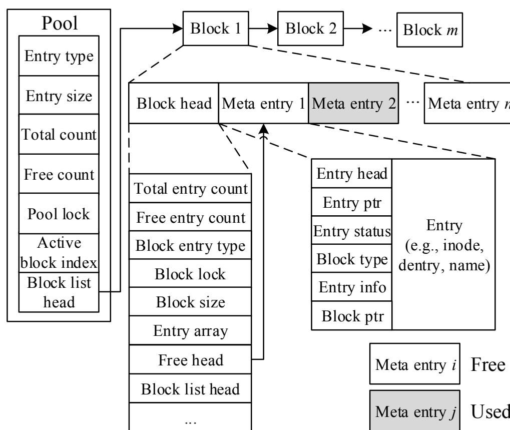


Figure 3: The structure of a pool.

pool-level variable "Entry type" (see Fig. 3). "Block lock" makes sure that in a particular block only one meta entry can be updated by a thread at a time. However, multiple threads can update meta entries in distinct blocks simultaneously. Note that when a thread updates a meta entry in a block it does not need to hold the "Pool lock". However, after a thread grabs a free meta entry from a block, it needs to acquire the "Pool lock" in order to update the value of "Free count". The size of a block (i.e., "Block size") is configurable and is set to 2 MB in current SPFS implementation. The number of blocks in a pool can be dynamically changed based on the requirements of applications. In order to improve the allocation efficiency and reduce TLB misses, SPFS ensures that the size of each block is relatively large. A large block size enables SPFS to map a metadata block using the huge-page mapping technique, which alleviates the TLB miss problem.

Each meta entry has two parts: OOB (out-of-band) area and metadata area (see Fig. 3). Two important items in the OOB area are "Entry head" and "Entry status". The "Entry head" field keeps a pointer to the meta entry block that an meta entry belongs to so that it can be returned to the block when it is freed. The "Entry status" field is used to trace the status of an meta entry. For example, when a meta entry is just allocated and has not been linked to a file system yet, this field is marked as *allocated*. When it is freed, this field is then changed to *unused*. When it is under an editing condition (e.g., an inode is being updated in Fig. 4b), this field is logged as *editing*. After the editing has been done (e.g., the updates of an inode have been flushed from cache lines to PM), the field then restores to *consistent*. This field can enhance SPFS metadata consistency.

Dentry: In order to fully exploit the random access capability of PM, we re-design dentry management. The length of each dentry is fixed to 64 bytes. In addition to the inode id and file name, an entry also contains multiple other variables.

File Name: In order to manage and search file names more efficiently, SPFS uses an array of pools to manage long file names. To avoid wasting memory space, several pools are provided to serve file names with various lengths (i.e., less than 64 bytes, between 64 and 127 bytes, between 128 and 191 bytes, between 192 and 255 bytes, and above 256 bytes). SPFS optionally provides a hash table for name indexing. This option enables the file system to avoid storing the same file name multiple times. In addition, with this option in hand, the administrator can force the file system to store all the file names in the name pools regardless of the length. As a result, each entry can use the space that is originally assigned for a file name to store a list node and a directory inode id. Thus, a file system is able to search a file name from the hash table instead of going through data block of the parent directory.

Super Block, Inode, and Directory: Since synchronization operations increase the risk of inconsistency, each SPFS volume stores a super block within its data structure. When a file system is mounted, it directly uses its super block instead of applying for a super block from the kernel. Unlike a traditional file system, SPFS uses only one inode for a file for all of its operations. Since the inode of the in-memory copy has extra information that can be obtained on-the-fly, SPFS divides the inode information into two parts: persistent part maintained by a file system and memory part used by memory management. The two parts are stored separately. In particular, SPFS uses different pools to manage each part of the inode information.

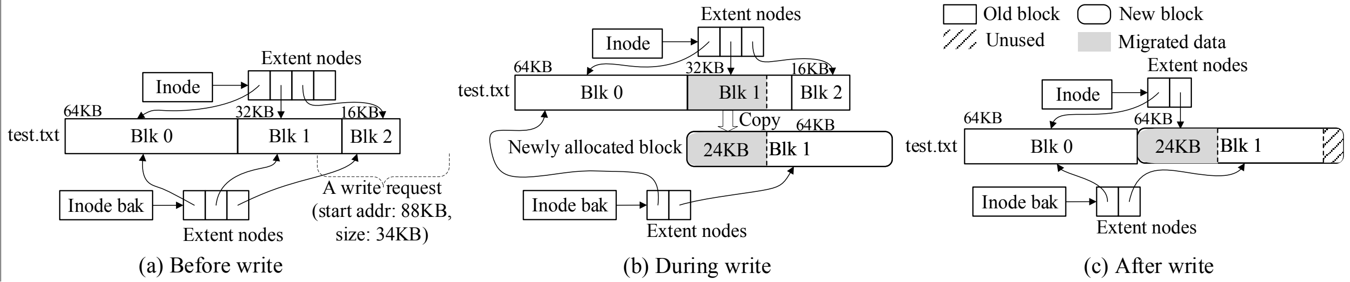


Figure 4: An example of data consistency maintenance.

For a large-size directory (i.e., a directory with more than 64 items in it), similar to a traditional dentry allocation scheme, SPFS allocates one or more 4 KB data blocks for it. For a small-size directory (i.e., a directory with no more than 64 items in it), SPFS employs a pool-based dentry allocation strategy. SPFS divides small-size directories into three categories: 16 items or fewer, between 17 items and 32 items, more than 32 items but no more than 64 items. The pool-based dentry allocation strategy has the following benefits. First, it allows all directories that are created or modified during the same period of time to stay in one 2 MB block. Consequently, TLB misses caused by directory data accesses can be reduced. Second, in the traditional dentry allocation scheme, since each 4 KB page requires a page table entry, there are 2,048 entries in the page table when 2,048 such small-size directories are managed. However, there is only one entry in the page table when a pool-based dentry allocation strategy is employed. Obviously, the pool-based dentry allocation strategy can largely decrease the number of entries in the page table, which in turn improves the TLB hit rate. Third, it saves memory space. For example, the traditional dentry allocation scheme needs a 8 MB memory space to store 2,048 such small-size directories. However, the pool-based dentry allocation strategy only requires a 2 MB memory space. Thus, it saves 75% memory space. Finally, it reduces the use of 4 KB data blocks, and thus, alleviates the memory fragmentation problem.

4.3 Data and Metadata Consistency

In this sub-section, we first explain how SPFS enforces write ordering and atomicity. Next, we use an example to illustrate how SPFS ensures consistency for data and metadata.

Enforcing ordering and atomicity: Both caches and memory controllers can re-order writes before they arrive in memory in order to enhance performance [5]. However, write re-ordering could lead a file system to an inconsistent state if a crash happens before all data have been written back to PM [5]. Similar to SCMFS [22], SPFS also adopts the combination of MFENCE and CLFLUSH. The main reason is that it does not require any specific hardware. Also, the two instructions are currently available. One drawback of it is that its performance penalty could be high [23]. Many file system operations require atomicity in the sense that either all steps of a file operation are successfully carried out or none of them has ever been executed [23]. Without an atomicity guarantee, a power outage can leave a data in an inconsistent state. Like BPFS [5] and NOVA [23], SPFS uses 8-byte atomic update instructions to ensure the atomicity of small writes like updating a pointer.

Data consistency: File systems normally use one of the following two methods to support consistency [5]: journaling [23] and copy-on-write (CoW) [5]. Apparently, journaling becomes less attractive due to its double copy overhead and the associated write amplification problem. Thus, similar to BPFS [5], SPFS also adopts CoW to support data consistency. Besides, it always maintains two copies of all file system metadata (e.g., inode, dentry, and indexing block) in order to ensure meatadata consistency and reliability. For example, each file has two copies of its inode: a primary inode and a backup inode. We use an example in Fig. 4 to illustrate how SPFS maintains data consistency.

Assume there is a file named test.txt, which consists of three data blocks: 64 KB, 32 KB, and 16 KB (see Fig. 4a). It has two copies of inode: a primary inode (i.e., *Inode*) and a backup inode (i.e., *Inode bak*). The contents of the extent nodes of these two inodes are identical as they are pointers to the same three data blocks (see Fig. 4a). Also, the two inodes both have *consistent* in their "Entry status" fields (see Fig. 3). Now, assume that a write request arrives. Its size is 34 KB and its start address is 88 KB (i.e., an offset within the file) (see Fig. 4a). In order to execute the write request, SPFS first uses *SPFS_recursive_alloc_blocks()* to allocate a new 64 KB data block for it (see Fig. 4b). Note that a data migration process will be triggered if partial content of an old data block will be overwritten by a write request. This is because SPFS needs to keep the valid data (i.e., untouched data) of the old data block by copying it to a newly allocated data block. In this example, 24 KB data (area in grey) of the old data block Blk 1 needs to be migrated into the newly allocated data block Blk 1 (see Fig. 4b). And then SPFS writes the new data of 34 KB into the newly allocated data block Blk 1. Next, a number of 8-byte atomic write operations on *Inode bak* separated by MFENCE instructions are carried out in the following order: mfence() → changing the "Entry status" field of *Inode bak* from *consistent* to *editing* indicating that an editing process on *Inode bak* is about to begin → mfence() → logging the size of the new data block → mefence() → updating the file size → mefence() → updating the modification timestamp → mfence() → updating the pointer of the second extent node to the new data block → mfence() → changing its "Entry status" field back to *consistent* → mfence(). Only after all these 8-byte atomic metadata updating operations are successfully committed on *inode bak* can SPFS start to modify the contents of the primary inode accordingly (see Fig. 4c). Similarly,

before SPFS begins to update *Inode*, it first changes its "Entry status" field from *consistent* to *editing*. After all updates have been done, its "Entry status" then returns to *consistent*. Next, SPFS sends a positive response to the caller to inform him that the write request has been successfully executed. While the outlined diamond area in the new data block Blk1 shown in Fig. 4c represents the 34 KB new data, the diagonal stripe area stands for unused space. Note that for each file an unused space can only exist in its last data block.

If a failure like a process crash or a power outage happens during the process of writing the new data of 34 KB into the newly allocated data block Blk 1 (see Fig. 4b), SPFS has no chance to send a positive response to the caller. Thus, the caller knows that the write request fails. Data consistency of the file is still guaranteed because the old data of the file are untouched and the "Entry status" fields of the two inodes are all in a *consistent* status (i.e., the pointers in both *Inode* and *Inode bak* all point to the old data blocks). In next paragraph, we explain how metadata consistency can also be guaranteed.

Metadata consistency and reliability: For a small size metadata update such as updating a 64-bit pointer in the backup inode (i.e., *Inode bak*, see Fig. 4b), SPFS uses an atomic 8-byte write supported by the processor to ensure its consistency. For each piece of file system metadata, SPFS always maintains two copies of it to ensure its consistency and reliability. We still use the example shown in Fig. 4 to illustrate how a metadata dual-copy technique can ensure metadata consistency. Since *Inode* and *Inode bak* are always updated sequentially, at any time it is impossible that both of them are in an *editing* status. If a power outage happens when SPFS is updating one of them, after a system reboot only the following three cases could exist: (1) *Inode* is in a *consistent* status while *Inode bak* is in an *editing* status; in this case SPFS simply recovers *Inode bak* by copying *Inode* to it; (2) *Inode* is in an *editing* status while *Inode bak* is in a *consistent* status; in this scenario SPFS uses *Inode bak* to re-construct *Inode*; (3) both *Inode* and *Inode bak* are in a *consistent* status but *Inode bak* pointing to the new data block while *Inode* still pointing to the old data; this rare situation takes place only when a power outage happens **right after** SPFS finishes updating *Inode bak* and **just before** it starts to change *Inode* accordingly; in this extreme scenario SPFS uses *Inode* to overwrite *Inode bak* so that the two copies are synchronized again. Note that only after both copies of the inode have been successfully updated can SPFS inform the caller that the write request is done. Otherwise, the caller can never receive a "job done" response, in which case he knows that the request fails.

Our proposed metadata dual-copy technique not only ensures metadata consistency but also enhances their reliability. Our design decision is made based on following considerations. First, the CPU overhead caused by journaling or CoW is comparable to that of maintaining dual copies of metadata. For example, an update needs to be written twice in journaling, which is similar to dual-copy metadata updating. Second, the size of metadata is normally very small compared with the data size in a file system. For instance, the metadata overhead of most file systems including NTFS and Ext4 is less than 3.5% [8]. In our SPFS experiments, we find that in the worst case (i.e., Postmark) the extra copies of all metadata only take 2.9% space of the entire file system. Besides, a PM device normally has a larger capacity than a DRAM system. Thus, the space cost of maintaining a dual copy for each file system metadata is acceptable. Third, metadata normally are frequently updated. As a result, metadata are more likely to be corrupted than normal data. Obviously, having a second copy of a piece of metadata can enhance its availability and reliability. In case a PM hardware failure corrupts one copy of an inode, SPFS can simply use the other copy to re-construct it in a new location. Note that in an existing PM-based file system there are also two copies of metadata (i.e., one inode in DRAM and one inode in PM). However, they are just redundant copies, which can neither ensure metadata consistency nor enhance metadata reliability.

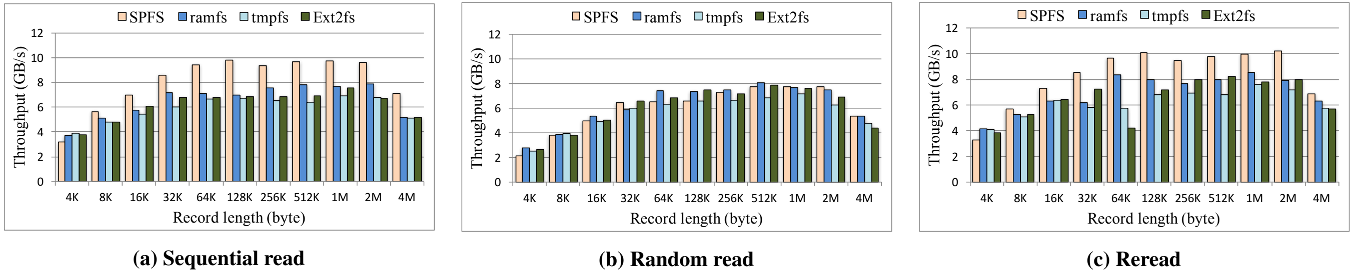
5 EVALUATION

In this section, we evaluate the performance of SPFS. Like SCMFS [22], we compare SPFS with two existing memory based file systems ramfs [2] and tmpfs [20] as well as one block file system Ext2fs running on a ramdisk [1]. Since SCMFS was also compared with the three file systems under the same two benchmarks [22], we indirectly compare SPFS with SCMFS. The comparisons between SCMFS and SPFS are not accurate as the two file systems plus the three baselines are tested under two different experimental setups. Still, they provide us with some insights. Also, we conduct a group of experiments to compare SPFS with NOVA [23]. The commit number of NOVA is 356a492 and we use its default configurations except *memmap* = *8G*!*8G*. In fact, it is evident that SPFS is at a disadvantage relative to its hybrid counterparts [22][6][5][23] in terms of performance. After all, they do not need to guarantee metadata consistency in volatile DRAM. On the other hand, maintaining metadata consistency in PM incurs a high overhead in SPFS.

5.1 Experimental Setup

Due to the lack of real PM hardware, similar to existing PM-based file systems [22][6][23], SPFS also uses volatile DRAM to emulate a PM device. In other words, all experiments in this paper were conducted on DRAM only. Since the Intel PMDB platform is not publicly available and open-sourced NOVA does not provide an interface to inject PM latencies, one has to use DRAM to emulate a PM device in order to run NOVA. Thus, like SCMFS [22] as well as open-sourced PMFS [6] and NOVA [23], all experimental results presented in this section are based on the latency and bandwidth of DRAM. All experiments are conducted on an iMac Mid 2011 computer, which has a quad core i5-2400 CPU (3.1 GHz) equipped with 32 GB DRAM (four 8GB DDR3 PC3-10600 1333MHz 204Pin), a 256 GB SSD, and a 2 TB hard disk. The maximal bandwidth of DRAM is 21.2 GB/s. An Ubuntu 14.04 is running on the computer with a customized Linux kernel. Since DRAM is volatile, we store the entire PM partition as an image to the hard disk before a power off and copy it back after a booting.

Since operating system needs enough memory space to ensure a stable performance, we configure half of the total memory (i.e., 16 GB) as the storage partition while the other half is used as main memory partition. We use both IOzone [18] and Postmark [9] as benchmarks in our experiments. In order to mimic an aged file system, we run both benchmarks twice before we begin to collect the results. Based on our observations, results of IOzone vary in different running instances, so we run each configuration 10 times and calculate an average value to represent a stable performance. Postmark, however, shows us stable results on a configuration. Based

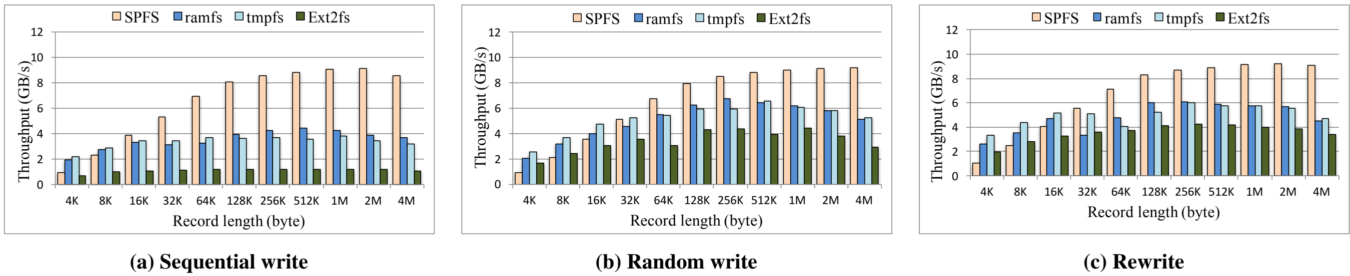


(a) Sequential read

(b) Random read

(c) Reread

Figure 5: IOzone read performance.

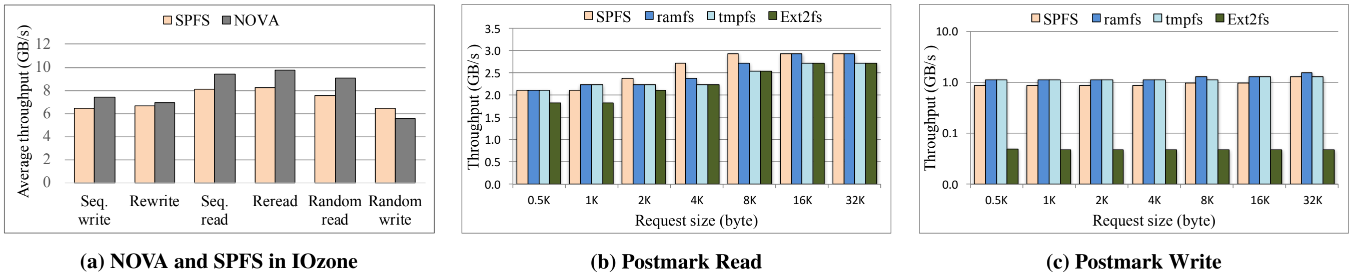


(a) Sequential write

(b) Random write

(c) Rewrite

Figure 6: IOzone write performance.



(a) NOVA and SPFS in IOzone

(b) Postmark Read

(c) Postmark Write

Figure 7: Comparisons with NOVA and Postmark performance.

on experimental results, we find that ramfs performs best in general among all three baseline file systems.

5.2 IOzone Results

The IOzone benchmark creates a file with a configured size at the beginning of each execution and then it runs an array of performance tests on the file [18]. In the experiments, we measure performance of sequential write and read, random read and write, as well as reread and rewrite in terms of throughput. In our configurations, the file size is set to 16 MB while the length of a record varies from 4 KB (i.e., the smallest size IOzone allowed) to 4 MB. In fact, we find that SPFS still outperforms the three baseline file systems when the file size is set to 512 MB or even larger.

The results from IOzone are shown in Fig. 5, Fig. 6, and Fig. 7a. In general, the performance of all four file systems improves as the record length increases from 4 KB to 512 KB. All of them show their peak throughput when the record length changes between 512 KB to 2 MB. The reason is that when record length is small CPU time on pre-processing and post-processing takes a large percentage of the total data accessing time. In addition, the two figures also show that different file systems have a similar read throughput. This is because a read process is similar among the four file systems. We notice that when the record length is 4 KB, SPFS performs even worse than Ext2fs in both read and write. In terms of read, this is because Ext2fs can fully exploit page caching when the record length is 4 KB. In terms of write, the reason is that when the record length is very small the CPU overhead caused by SPFS metadata dual-copy technique becomes dominant.

Fig. 5a shows that SPFS performs better than other file systems in most cases on sequential read except 4 KB. On average, SPFS improves the sequential read performance by 22.1%, 33.8%, 28.3% over ramfs, tmpfs, and Ext2fs, respectively. The improvement comes from SPFS XIP read optimization, which enables copying multiple contiguous pages of data at a time. From Fig. 5b, compared with ramfs, SPFS on average decreases performance by 4.4% under random read workloads. From results in [22], we find that on average SCMFS outperforms ramfs in random read by around 10%. The implication is that SCMFS is moderately better than SPFS in random

read. In reread shown in Fig. 5c, SPFS outperforms ramfs, tmpfs, and Ext2fs by 17%, 31%, and 28%, respectively.

The four file systems exhibit very different write performance. In sequential write (see Fig. 6a), SPFS achieves the highest performance improvement compared with ramfs. For example, in 2 MB case its throughput improves 164% over ramfs while SCMFS only improves around 15% [22]. On average, SPFS has a 75%, 87%, and 498% performance improvement compared to ramfs, tmpfs, and Ext2fs, respectively (see Fig. 6a). In terms of random write, SPFS performs worse than tmpfs when record size is no larger than 32 KB (see Fig. 6b). However, it performs best in all other scenarios. On average, it improves performance by 81% compared to ramfs. However, SCMFS was outperformed by both ramfs and tmpfs in most cases in random write [22]. Even Ext2fs performs better than SCMFS when the record length was between 64 KB and 512 KB [22]. In rewrite, SPFS is able to raise performance by 33%, 30%, 79% on average compared to ramfs, tmpfs, Ext2fs, respectively (see Fig. 6c). The reason is that SPFS supports contiguous blocks copying.

We also compared the performance of NOVA and SPFS under the IOzone benchmark. Experimental results of the two file systems under IOzone are shown in Fig. 7a. The results demonstrate that when the record length varies from 4 KB to 4 MB, on average NOVA outperforms SPFS by 14.4%, 16.9%, 12.4%, 15.8%, 3.7% in sequential read, random read, sequential write, reread, and rewrite, respectively. In random write, on average SPFS outperforms NOVA by 16.2%. This is because SPFS employs a pool-based metadata management that can improve the locality of metadata.

5.3 Postmark Results

Postmark was designed to create a large pool of continually changing files and to measure the transaction rates for a workload approximating a large Internet electronic mail server [9]. It is an I/O intensive benchmark [22]. In our experiments, the number of files is set to 20,000 for read tests (see Fig. 7b) and 200,000 for write tests (see Fig. 7c) within one directory. The size of files is in the range of 8 KB to 40 KB. The number of transactions is set to 1,000,000 for both read and write tests. To measure read throughput, we disabled the create/delete option. The general trend for all four file systems is that read performance improves as the request size grows (see Fig. 7b). From 2 KB onward, SPFS becomes the best player. On average, SPFS outperforms ramfs, tmpfs, and Ext2fs by 3.2%, 7.6%, and 14%, respectively. Since the write throughput of Ext2fs is much smaller than all others, we use a log scale in the Y axis in Fig. 7c. SPFS outperforms Ext2fs by 190%. However, compared with the two memory file systems ramfs and tmpfs, the write throughput of SPFS decreases by 22.2% and 18.1%, respectively. This is because the two memory file systems do not need to enforce data/metadata consistency, which incurs a noticeable performance penalty. We find that the performance degradation of SPFS is mainly caused by the metadata dual-copy technique, which requires SPFS to update two copies of a metadata serially.

6 CONCLUSIONS

SPFS is the first single-level persistent file system that completely bypasses conventional volatile main memory. It outperforms two traditional DRAM-based in-memory file systems ramfs and tmpfs in most cases. Although its performance in most cases is lower than that of NOVA, SPFS provides two unique benefits. First, it is more energy-efficient as power-hungry DRAM is longer needed. Second, it leads to a shorter system warming up period. Our next step is to transfer other OS modules from DRAM to PM.

7 ACKNOWLEDGMENTS

The authors thank their shepherd Kristian Rietveld and anonymous reviewers for their constructive comments. This research is sponsored by US National Science Foundation under grant CNS-1320738.

REFERENCES

[1] 2004. Using the RAM disk block device with Linux. https://kernel.org/doc/Documentation/blockdev/ramdisk.txt. (2004).

[2] 2013. ramfs. https://wiki.debian.org/ramfs. (2013).

[3] 2017. Intel To Launch 3D XPoint DIMMs in 2H 2018. https://www.anandtech.com/show/12041/intel-to-launch-3d-xpoint-dimms-in-2h-2018. (2017).

[4] Katelin Bailey, Luis Ceze, Steven D Gribble, and Henry M Levy. 2011. Operating System Implications of Fast, Cheap, NVM.. In *HotOS*, Vol. 13. 2–2.

[5] Jeremy Condit, Edmund B Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 133–146.

[6] Subramanya R Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System software for persistent memory. In *Proceedings of the Ninth EuroSys*. ACM, 15.

[7] Coulter Garreffa. 2016. Intel's 3D X-Point technology enables up to 6TB of system memory. http://www.tweaktown.com/news/49408/. (2016).

[8] Jones. 2009. Filesystem Metadata Overhead. https://rwmj.wordpress.com/2009/11/08/filesystem-metadata-overhead/. (2009). [Online; accessed April-2016].

[9] Jeffrey Katcher. 1997. Postmark: A new file system benchmark. www.netapp.com/tech_library/3022.html. (1997). [Online; accessed 15-April-2015].

[10] Hyojun Kim, Sangeetha Seshadri, Clement L Dickey, and Lawrence Chiu. 2014. Evaluating phase change memory for enterprise storage systems: A study of caching and tiering approaches. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*. 33–45.

[11] Emre Kultursay, Mahmut Kandemir, Anand Sivasubramaniam, and Onur Mutlu. 2013. Evaluating STT-RAM as an energy-efficient main memory alternative. In *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*. IEEE, 256–267.

[12] Chris Lattner and Vikram Adve. 2005. Automatic pool allocation: improving performance by controlling data structure layout in the heap. In *Proceedings of ACM SIGPLAN conference on PLDI*. ACM, 129–142.

[13] L. Mearian. 2014. A terabyte on a postage stamp: RRAM heads into commercialization. www.computerworld. (2014). [Online; accessed 15-April-2015].

[14] Justin Meza, Yixin Luo, Samira Khan, Jishen Zhao, Yuan Xie, and Onur Mutlu. 2013. A case for efficient hardware/software cooperative management of storage and memory. *Proceedings of the Workshop on Energy-Efficient Design* (2013).

[15] Jeffrey C Mogul, Eduardo Argollo, Mehul A Shah, and Paolo Faraboschi. 2009. Operating System Support for NVM+ DRAM Hybrid Main Memory.. In *HotOS*.

[16] Iulian Moraru, David G Andersen, Michael Kaminsky, Niraj Tolia, Parthasarathy Ranganathan, and Nathan Binkert. 2013. Consistent, durable, and safe memory management for byte-addressable non volatile main memory. In *Proceedings of ACM SIGOPS Conference on Timely Results in Operating Systems*. ACM, 1.

[17] Onur Mutlu. 2013. Memory scaling: A systems architecture perspective. In *Memory Workshop (IMW), 2013 5th IEEE International*. IEEE, 21–25.

[18] William D Norcott and Don Capps. 2003. Iozone filesystem benchmark. *URL: www. iozone. org* 55 (2003).

[19] Moinuddin K Qureshi, Vijayalakshmi Srinivasan, and Jude A Rivers. 2009. Scalable high performance main memory system using phase-change memory technology. *ACM SIGARCH Computer Architecture News* 37, 3 (2009), 24–33.

[20] Peter Snyder. 1990. tmpfs: A virtual memory file system. In *Proceedings of the Autumn 1990 EUUG Conference*. 241–248.

[21] Alexander S Szalay, Gordon C Bell, H Howie Huang, Andreas Terzis, and Alainna White. 2010. Low-power amdahl-balanced blades for data intensive computing. *ACM SIGOPS Operating Systems Review* 44, 1 (2010), 71–75.

[22] Xiaojian Wu and AL Reddy. 2011. SCMFS: a file system for storage class memory. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 39.

[23] Jian Xu and Steven Swanson. 2016. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*. USENIX Association, 323–338.