Cluster Computing (2020) 23:2501–2515
https://doi.org/10.1007/s10586-019-03023-y

HasFS: optimizing file system consistency mechanism on NVM-based hybrid storage architecture

Yubo Liu[1,2] · Hongbo Li[1] · Yutong Lu[1] · Zhiguang Chen[1] · Nong Xiao[1] · Ming Zhao[2]

Received: 2 January 2019 / Revised: 21 October 2019 / Accepted: 2 December 2019 / Published online: 11 December 2019

Abstract
In order to protect the data during system crash, traditional DRAM–DISK architecture file systems (e.g., EXT4) need to synchronize the dirty metadata and data from the memory to disk. At the same time, the disk synchronization may break the consistency of file system upon a crash, so traditional file systems use some mechanisms to guarantee the file system consistency when the dirty metadata and data is synchronized onto persistent storage devices (e.g., HDD and SSD). Journaling is a consistency mechanism widely used by file systems. We observe that the overhead of periodic disk synchronization and journaling is high. Emerging non-volatile memories (NVMs) can be potentially utilized to reduce these overheads. In this paper, we present hybrid architecture for storage file system (HasFS), a file system designed for the DRAM–NVM–DISK architecture. HasFS extends the main memory with NVM and considers NVM as a persistent page cache to eliminate the periodic disk synchronization overhead of dirty data. Then we design an efficient consistency mechanism based on the hybrid memory architecture to provide strong (both metadata and data) consistency guarantee with low overhead. The evaluation demonstrates that HasFS outperforms mainstream DRAM–DISK file systems for many workloads. For instance, HasFS has between 1.6X to 46.6X performance improvement over other tested file systems in random write workload. In particular, HasFS outperforms EXT4 without journal in some cases even though HasFS provides metadata and data consistency guarantees (similar to EXT4 with journal data mode).

Keywords Operating system · File system · Hybrid storage · Non-violate memory · NVM · Consistency · Recovery · Page cache

1 Introduction

Emerging non-volatile memories (NVMs), such as spin-torque transfer RAM (STT-RAM) [17], phase change memory (PCM) [4, 19], resistive RAM (ReRAM) [39], and 3D XPoint [15] are promising storage media for building fast, byte-addressable, and persistent storage systems. These new technologies give us opportunities to optimize the design of file systems.

Many NVM-based file systems use NVM as the persistent storage. They are excellent choices for small-scale storage systems such as those used in personal computers and small storage clusters. However, NVM is not yet eligible as persistent storage media for large-scale storage systems due to the small capacity and high cost. For example, NVM and disk still have a big gap in their capacity [16] ($\approx$ 256 GB for NVM vs $>$ 1 TB for disk). Instead, a practical way for large-scale systems is to

✉ Yubo Liu
yliu789@asu.edu

Hongbo Li
hongbo.li@nscc-gz.cn

Yutong Lu
yutong.lu@nscc-gz.cn

Zhiguang Chen
zhiguang.chen@nscc-gz.cn

Nong Xiao
xiaon6@mail.sysu.edu.cn

Ming Zhao
mingzhao@asu.edu

[1] School of Data and Computer Science, Sun Yat-sen University, Guangzhou, China

[2] School of Computing, Informatics, and Decision Systems Engineering, Arizona State University, Tempe, USA

consider NVM as a cache and/or buffer. Taking into account the characteristics of different storage media, we still use disk as a persistent storage and leverage NVM to tackle two bottlenecks of traditional DRAM–DISK file systems.

The first bottleneck of DRAM–DISK file systems is the large overhead of periodic disk synchronization. In DRAM–DISK architecture, file system has to periodically write back the dirty data from the DRAM page cache to the disk to protect against data loss. The conditions of triggering periodic synchronization include the background flushing interval and the dirty page rate exceeding their thresholds, etc. This periodic writeback operation is costly, and it cannot be eliminated in the DRAM–DISK scheme, as DRAM is a volatile media.

NVM offers new opportunities to eliminate periodic disk synchronization. HasFS extends the main memory with NVM and considers NVM as a persistent page cache. Therefore, the disk synchronization can be replaced by permanently storing the dirty data on the NVM page cache. In order to use NVM efficiently, HasFS keeps metadata on DRAM because metadata is smaller and more frequently queried than data in general, and data is stored on NVM so that NVM capacity can be fully utilized and dirty data does not need to be periodically synchronized to the disk.

File systems must guarantee the crash consistency when metadata and data is written to persistent device. Therefore, they use consistency mechanisms such as journaling and copy-on-write to maintain the system consistency during a system crash. Journaling is the most popular consistency mechanism in current file systems. However, our observations show that the journaling process significantly degrades the performance of a file system, especially in the journal data mode which provides metadata and data consistency guarantee.

The second bottleneck of DRAM–DISK file systems is the high overhead of journaling. To reduce this overhead, keeping the journal on high-speed storage media (like NVM or SSD) is an intuitive idea. FGM [5] and NJS [40] keep the journal on NVM to improve the efficiency of journaling. However, they did not address the periodic disk synchronization problem. To fully utilize the advantages of NVM, we not only store the journal on NVM but also uses NVM as a persistent page cache. We propose an efficient consistency mechanism based on the NVM page and NVM journal to provide the metadata and data consistency guarantee with low overhead.

Our evaluations show that HasFS outperforms mainstream file systems in many cases. For instance, HasFS has 1.6 times to 46.6 times performance improvment compared to other tested file systems in random write. HasFS performs even better than EXT4 without journaling for many workloads, while guaranteeing similar to EXT4 with journal data mode. In addition, we also optimize the fsync operation based on the hybrid architecture and the novel consistency mechanism in HasFS. HasFS has significant performance improvement over other test systems. This paper makes the following contributions:

- We analyze the overhead of periodic synchronization and journaling in the file system.
- We propose an efficient way to use hybrid memory in the file system. Thus, HasFS eliminates data periodic synchronization and makes full use of the capacity advantage of NVM.
- We design an efficient consistency mechanism based on hybrid memory. It can provide metadata and data consistency guarantee with low overhead.
- We implement HasFS and compare its performance with the mainstream file systems.

The rest of the paper is organized as follows. Section 2 describes the background and motivations. Sections 3, 4, 5 and 6 present the design and consistency mechanism of HasFS. Section 7 discusses the evaluation results. Section 8 examines the related works. Section 9 concludes the paper.

2 Background and motivations

We describe the hybrid storage architecture and its benefits in Sect. 2.1. Then, we analyze the overhead of periodic synchronization and journaling in Sects. 2.2 and 2.3. Finally, we introduce the motivations of HasFS in the last two subsections.

2.1 Hybrid storage architecture

We consider three different types of storage media in our system. Their characteristics are different: DRAM is fast, volatile, and byte-addressable, but the price and the capacity are its disadvantages; NVM is non-volatile and is byte-addressable, while its write latency is about an order of magnitude higher than that of DRAM; disk is slow but of a large capacity, and it is much cheaper than DRAM and NVM. There are four mainstream storage architectures used in file systems.

Figure 1 shows four different storage architectures and related file systems in these architectures. The first column is the traditional DRAM–DISK architecture. Many mature file systems [3, 13, 22, 26, 27, 31] are designed for this architecture.

The second column in Fig. 1 shows the full-NVM architecture. The key goal of these single-level file systems is to to take advantage of NVM in the file system as much as possible. For example, PMFS [12] reduces the software

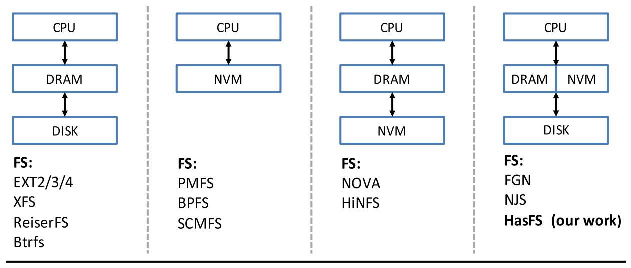


Fig. 1 Storage architecture comparison

overhead by bypassing the traditional page cache; BPFS [10] reduces the overhead of consistency guarantee on NVM by a hardware modification and a short-circuit shadow paging technique; SCMFS [36] utilizes memory management in the OS to implement a full-NVM file system.

Although the performance of NVM is much better than disk, it still has a certain gap with DRAM, especially in the write performance. Some studies combined DRAM and NVM, and took advantage of the characteristics of different hardware devices. The third column in Fig. 1 shows this DRAM–NVM architecture. HiNFS [24] focuses on reducing the write performance gap between the DRAM and NVM; NOVA [37] designs a log-structure file system on NVM, which stores index structure on DRAM but stored data on NVM.

These NVM-based (full-NVM and DRAM–NVM) file systems are suitable for scenarios where performance requirement is high but capacity requirement is not high, because NVM's capacity and price have significant disadvantages compared to the disk. We believe that DRAM, NVM, and disk will coexist in many scenarios. The final column in Fig. 1 shows a DRAM–NVM–DISK architecture, which is used in this paper. FGM [5] and NJS [40] are designed based on this architecture. They use NVM as a journal device. However, they do not make full use of the advantages of NVM. We will discuss their details in Sect. 2.3. In HasFS, we utilize this hybrid architecture to reduce the periodic synchronization overhead and the consistency guarantee overhead in the file system.

2.2 Periodic synchronization overhead

In the DRAM–DISK architecture, file system (or operating system) must synchronize the dirty pages from DRAM to disk periodically to protect against data loss, because DRAM is volatile. Periodic synchronization will be triggered under some conditions, such as when the dirty page ratio exceeds a threshold or the synchronization interval reaches the set period. Disk synchronization overhead comes from two aspects: First, the dirty files are locked until the synchronization operation is completed. Second, it consumes a lot of disk bandwidth. We ran a 4 KB random write benchmark on EXT4 with a 20 GB file (DRAM size was 16 GB) to measure the periodic synchronization overhead. We disabled the periodic synchronization by resetting the parameters of background flushing in the proc file system and calculated the degradation of throughput. Table 1 shows that the synchronization leads to 0.25% to 76.6% performance loss on the DRAM–SSD architecture, and 0.13% to 78.9% on the DRAM–HDD architecture. It only leads to 0.25% and 0.13% on SSD and HDD in journal data mode because the journaling cost is much higher than the periodic synchronization cost.

Motivation 1 NVM gives us an opportunity to reduce this synchronization overhead through its non-volatile feature. HasFS considers NVM as a persistent page cache and keeps file data on NVM to reduce the synchronization cost of dirty file data. In this design, the disk synchronization of dirty data can be replaced by persistently storing the dirty data on NVM. In our evaluations, HasFS outperformed full DRAM file systems for many workloads, even though NVM does not perform as well as DRAM.

2.3 Journaling overhead

In the DRAM–DISK architecture, disk synchronization may break the file system consistency during a crash. Journaling is a popular consistency technique used in file systems. Many DRAM-based file systems (e.g., EXT4 [13], XFS [31], and ReiserFS [26]) use journaling to maintain the after-crash consistency.

However, the journaling overhead is large, especially in the journal data mode, which provides metadata and data consistency guarantee. We ran a benchmark in EXT4 in 4KB random write with a 20GB file (DRAM size is 16GB). Table 2 shows the performance on different hardware (HDD and SSD) with different consistency levels (writeback, ordered data and journal data modes). The performance loss rate is calculated relative to EXT4 with no journal. Our experiment shows that: (1) Keeping the journal on a fast storage device can reduce the cost of journaling. The performance impact of journal is more obvious

Table 1 Periodic synchronization cost

Journal mode	Periodic synchronization cost	
	SSD (%)	HDD (%)
Journal data	0.25	0.13
Ordered data	75.7	77.3
Writeback	76.6	78.9

Table 2 Journaling cost

Journal mode	Consistency guarantee cost	
	SSD (%)	HDD (%)
Journal data	61.5	96.8
Ordered data	10.3	22.3
Writeback	4.2	20.7

on HDD than on SSD. (2) Performance degrades significantly in journal data mode.

At the same time, we also evaluated a simple way to incorporate NVM into local file system: consider NVM as a journal block device (called NVMJ) and keep the journal on NVM. We used DRAM to emulate NVM in this case (no latency added). We built a ramdisk on DRAM and mounted the file system journal on the ramdisk, and then used different ramdisk sizes (128MB and 1GB) to evaluate the impact of the journal size on performance. We compared the traditional EXT4 (denoted by "ondisk" in the figure) and EXT4 with NVMJ on random write benchmark with different I/O sizes. The testbed has 16GB DRAM and the test file is 20GB. We used journal data mode in this experiment, because our design aims to provide consistency guarantee similar to the journal data mode in traditional journal file systems. We ran this experiment on HDD and SSD to evaluate the journaling overhead in different storage devices.

Figure 2 shows the results. Under the default journal size (128MB), NVMJ can bring a small amount of performance improvement in the HDD case, whereas NVMJ has no obvious advantage in the SSD case. Throughput can be increased by expanding the size of the journal, but still far less than the EXT4 without journal in most cases. This experiment shows that simply use NVM as a journal block device cannot take full advantage of NVM. The reasons are: (1) It still needs to synchronize the dirty data in the journal to the disk in checkpointing. (2) It cannot fully use the DRAM and NVM space because data pages may exist in the both page cache and NVM journal.

Some studies leveraged NVM to enhance the journaling efficiency. FGM [5] considers NVM as a journal device and proposes a fine-grained method to protect metadata consistency. Benefiting from the byte-addressable feature of NVM, FGM can reduce the write amplification in the journaling. However, FGM provides only metadata consistency. NJS [40] is a similar study to FGM, which provides both metadata and data consistency with low overhead. Both FGM and NJS only utilize NVM to reduce the overhead of journaling, so their performance is still worse than the non-journal setting in most scenarios.

Motivation 2 HasFS aims to provide metadata and data consistency guarantees with low overhead. Unlike other NVM journal file systems, HasFS not only keeps the journal on NVM but also considers NVM as a persistent page cache. We propose a novel consistency mechanism on this special architecture. As a result, HasFS outperforms mainstream journal file systems in many scenarios, even EXT4 without journal. At the same time, HasFS can provide metadata and data consistency guarantee (similar to EXT4 with journal data mode).

3 HasFS overview

In this section, we show the architecture of HasFS in Sect. 3.1. Then we introduce the hybrid memory in HasFS and discuss its benefits in Sect. 3.2.

3.1 Architecture

Figure 3 shows the architecture of HasFS. HasFS has two main modules: a hybrid memory management module and a consistency mechanism. In addition, HasFS reuses some

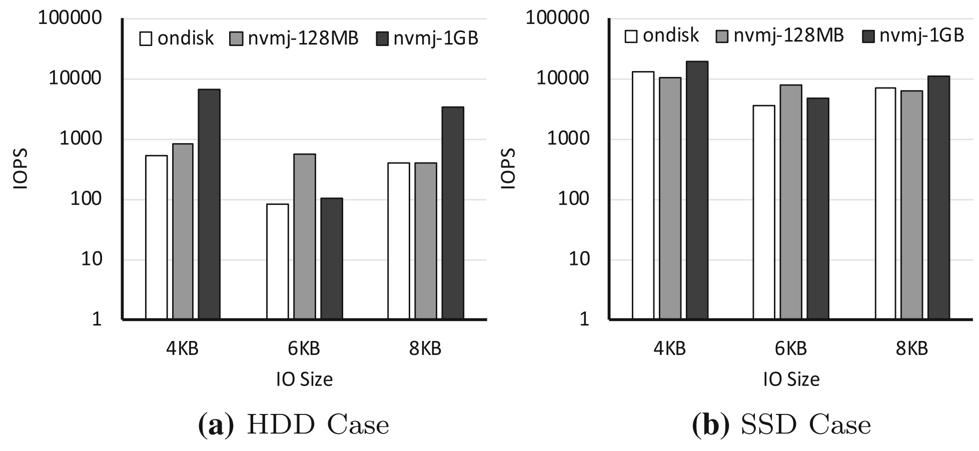


Fig. 2 Traditional EXT4 vs. EXT4 with NVM journal

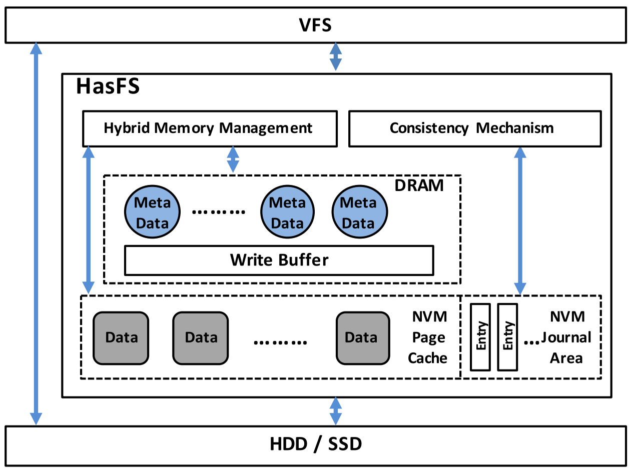


Fig. 3 HasFS architecture

components in EXT4 and VFS, such as namespace management and disk management. The hybrid memory management module is responsible for managing the metadata on the DRAM and the data on NVM. The consistency mechanism is used to provide metadata and data consistency after the system crash, including non-ordering journaling (for metadata) and lazy copy-on-write (for data).

3.2 Hybrid memory

Hybrid memory is designed to reduce the synchronization overhead that we mentioned in the Sect. 2. Hybrid memory consists of DRAM space and NVM space. HasFS keeps metadata on DRAM but data on NVM. There are two reasons for this design. First, the structure of metadata is more complex than data. Metadata includes some tree index structures, which will be frequently queried and modified. Prior studies have pointed out that it is costly to keep the consistency of tree indexes on NVM [25, 32, 38]. As such, HasFS stores metadata on DRAM. Second, file data size is usually much larger than metadata in file systems, and the file data has no complex structure. So storing the file data on NVM is good for taking advantage of the hardware features of the NVM.

The NVM space is divided into two areas: an NVM page cache and an NVM journal. As shown in Fig. 3, the file data pages are stored on the NVM page cache. In the DRAM-based scheme, periodic synchronize needs to be triggered under some conditions, such as when the dirty page ratio exceeds a threshold or the synchronization interval reaches the set period. In HasFS, the file's data pages can be used with cacheline flushing (clflush) and memory barrier (mfence) for persistent storage on the NVM, which means that the system does not need to periodically synchronize dirty data to the disk. Of course, dirty metadata still needs to be synchronized to the persistent storage. Under some conditions, HasFS's consistency mechanism will write the dirty metadata to the NVM journal to ensure after-crash consistency of the metadata. We will describe the consistency mechanism in Sect. 4. In this hybrid memory design, periodic synchronization of dirty data is eliminated, and dirty metadata only needs to be logged onto a fast NVM rather than a slow disk.

In the DRAM–DISK scheme, the cost of disk I/O in synchronization may mask the speed advantage of DRAM. Our evaluations show that HasFS outperforms DRAM–NVM file systems in many cases even though DRAM is faster than NVM. The reason is that the dirty files will be locked during the disk synchronization and the next operations need to wait for the slow disk IOs to complete in the DRAM–DISK file systems.

4 Consistency mechanism in HasFS

As discussed in the background section, simply using NVM as a journal block device can not fully utilize the advantages of NVM. We find that the efficiency of consistency mechanism can be improved by the hybrid memory architecture. For the metadata, HasFS logs dirty metadata in the NVM journal before it is persistently stored on disk to protect metadata consistency. For the data, thanks to the NVM page cache, HasFS can simply use a method which is similar to copy-on-write (lazy copy-on-write) to protect data consistency. HasFS provides consistency guarantee similar to EXT4 with journal data mode with low overhead.

4.1 Metadata consistency

To guarantee metadata consistency, HasFS uses the principle of traditional journaling with ordered mode, which is to write metadata to the journal after the data is persistently stored. Figure 4 shows the structure of the NVM journal. It is similar to the traditional journal structure. Specially, the field of dirty NVM page listed in the entry is used to record the out-of-date pages that are related to the entry. The metadata journaling will be triggered by some system events (e.g. periodic journaling) and some user behaviors

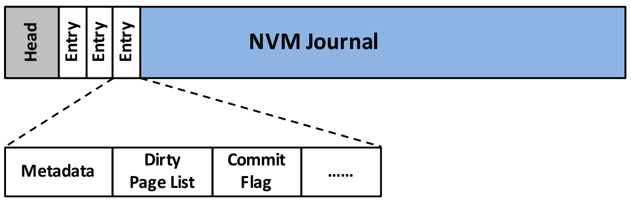


Fig. 4 Struture of NVM journal

(e.g. fsync). When the journaling process is triggered, a new entry will be created in the NVM journal and the dirty data pages will be copied to a new space (lazy copy-on-write, we discuss the data consistency in Sect. 4.2), and then the locations of these dirty pages will be recorded into the NVM journal. The metadata on NVM journal will be committed to disk when checkpointing, similar to the traditional journal.

4.2 Data consistency

HasFS uses lazy copy-on-write (LCOW) for the dirty data pages when the dirty metadata is logged. LCOW is similar to COW. The difference is that LCOW does not copy the page in each update. Instead, it copies the page when a new journal entry is generated by the file system. Such mechanism leads to multiple versions of a page on the NVM. HasFS divides the pages on the NVM page cache into three types: consistent page, out-of-date page, and active page.

Consistent pages represent pages that can be written back to disk without affecting the file system consistency. In other words, consistent pages are not associated with any dirty metadata in the NVM journal. HasFS manages consistent pages in the same way as clean pages in traditional DRAM page cache. They can be evicted to disk at any time. Consistent pages are generated from the last checkpoint or be loaded from the disk.

Active pages are new pages generated by LCOW, which are the latest version in the copy-on-write process. Such pages cannot be moved to another memory location or disk. All read and update requests will occur on the active pages.

Out-of-date pages are old pages generated by LCOW, they are the old (but not the latest) version pages in the copy-on-write process. They are locked in NVM page cache and cannot be accessed (including read and write) and moved to anywhere (other memory location or disk).

Figure 5 gives an example. P0 is a consistent data page in the NVM page cache and entry 1 is created before P0 is updated. Update 1 causes P0 to be copied to P1, and subsequent updates (update 2 and 3) are updated in place (on P1). Then, entry 1 is written to the journal and entry 2 is generated. P1 is set to out-of-date and subsequent updates (update 4 and 5) are updated to a new active page P2. When the memory space is insufficient, the operating system needs to replace some pages out of memory. However, a page may have multiple versions in HasFS. We will commit the journal if the memory is full, so each page has only one consistent version before the page replacement happens.

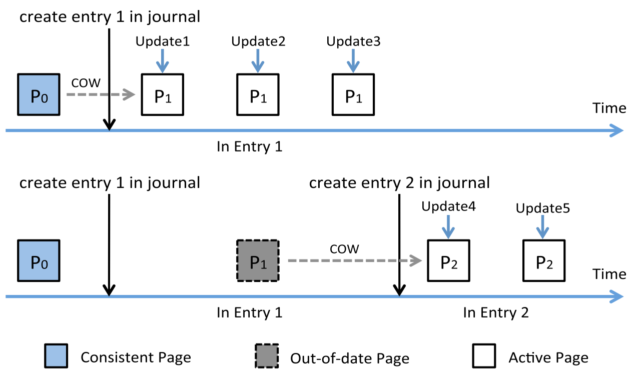


Fig. 5 Lazy copy-on-write

4.3 Write ordering guarantee

In order to protect both metadata and data consistency after the system crash, HasFS must guarantee the dirty metadata (entry) is written into the NVM journal after the dirty data is persisted. However, file system does not know when the dirty data will be flushed from a CPU cacheline to memory (NVM). We can not control the persistent order because the instructions reordering in the CPU without using the some hardware or software methods [10, 12, 24, 36].

HasFS uses a software method to make the dirty data persistently stored on NVM in order. We use cacheline flushing instruction (clflush) to flush all dirty data to memory and used memory barrier (mfence) to guarantee the storing order. Some works [25, 32, 38] pointed out that these instructions will reduce the CPU efficiency because they break the data locality in the CPU cacheline. In practice, this problem is not serious in HasFS because the NVM persistence operation is incurred only in entry creation.

4.4 Main processes in consistency mechanism

We introduce several important processes of the HasFS consistency mechanism in this subsection.

4.4.1 Entry creation

Entries will be created when some events happen, such as periodic journaling start and fsync. There are three steps to create a new entry in an NVM journal: (1) Find out all dirty active pages in NVM and change their type to out-of-date. Then we use clflush and mfence to guarantee all out-of-date pages are persistently written on the NVM before their metadata is recorded into the NVM journal. Later updates on these pages will trigger LCOW. (2) Write the dirty metadata and the locations of these out-of-date pages into the entry. (3) Use clflush and mfence for the new entry to make sure it is persistently written to NVM journal.

Figure 6a gives a simple example of creating entry 3. We assume that there are only three pages in the NVM

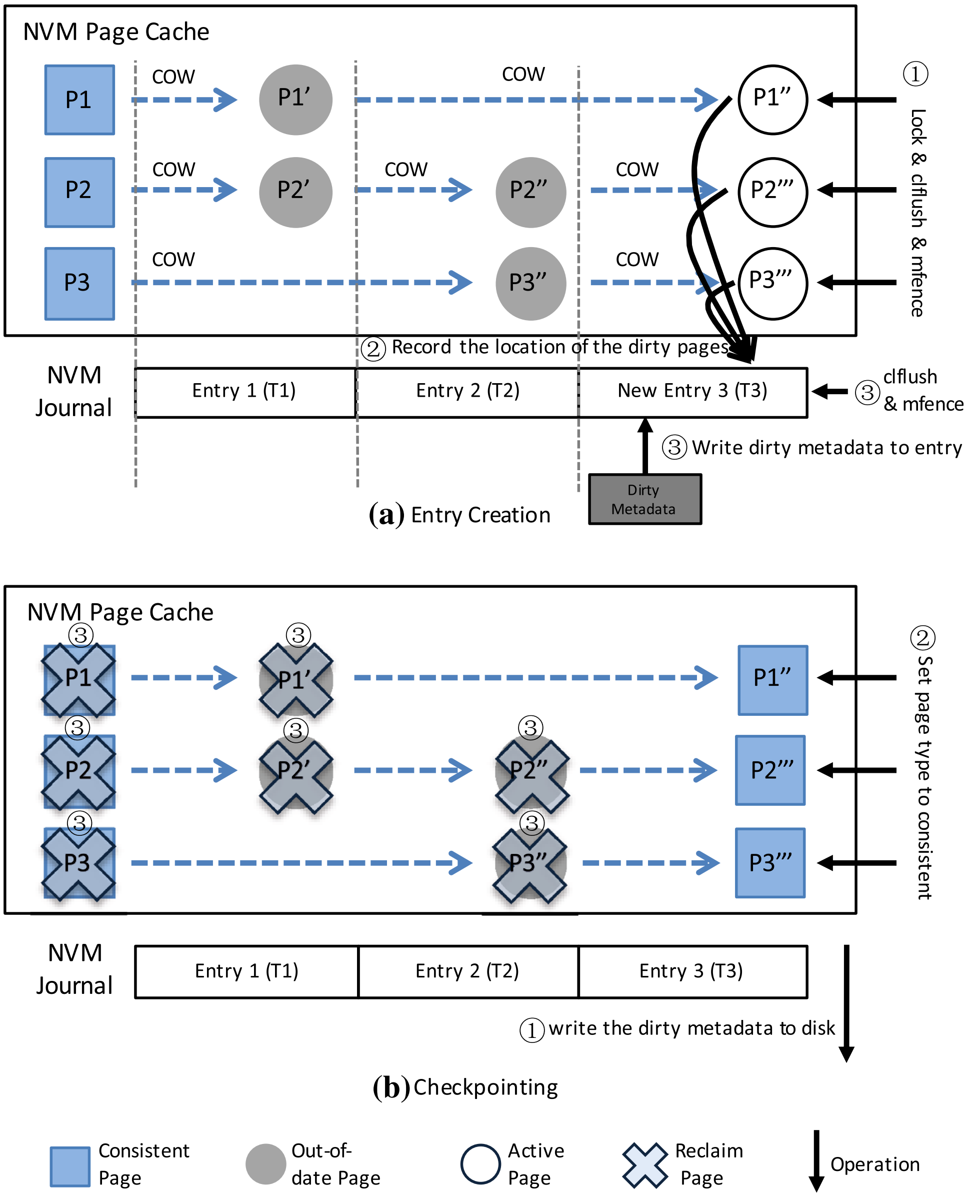


Fig. 6 Main processes in HasFS

page cache. Entry 1 and 2 are created during T1 and T2. P1' P2' and P1'' P2'' belong to entry 1 and 2 respectively. P1'', P2''' and P3''' are dirty active pages in T3, and they are set to "out-of-date" after the entry is created.

4.4.2 Checkpointing

HasFS commits the NVM journal to the disk when the journal size reaches the threshold. Furthermore, some events such as memory space insufficiency and system call can also lead to checkpointing. There are three steps to commit a journal: (1) Commit the metadata in the entries to the disk. (2) Trace all latest out-of-date pages and set their page type to "consistent". (3) Reclaim all out-of-date pages, old consistent pages, and the entires in the NVM journal.

Figure 6b gives a simple example of checkpointing. The initial status of this example is based on Fig. 6a. P1'', P2''', and P3''' are the latest out-of-date pages when the journal is committed at the end of T3. HasFS writes only the dirty metadata back to the disk in checkpointing, because the dirty pages have persisted on the NVM page cache when the entries are created in the journal.

4.4.3 Recovery

The recovery process in HasFS is similar to that of the traditional journaling file systems. Due to the lazy COW, there are multiple versions of pages in the NVM page cache. HasFS needs to write back only the pages belong to the latest consistent version. For example, in the previous example (Fig. 6a), the latest out-of-date pages are P1',P2", and P3" (if the system crash before the entry 3 is created). Algorithm 1 shows the recovery process in details. The first step of recovery is to find out the latest out-of-date pages in the NVM page cache, and flush the latest out-of-date pages and all consistent NVM pages to the disk. In our prototype, we assume that the NVM allocator can track those pages that have already been allocated after the system crash. The second step is to redo the metadata in the NVM journal. Finally, HasFS cleans the NVM journal and the NVM page cache. In addition, if the system crashes occurs during the recovering, HasFS can easily recover through the same process, because the NVM journal and the pages in the NVM page cache will not be reclaimed until recovery is successful.

Algorithm 1 Recovery Process

```
INPUT 1: nvm_pagecache
INPUT 2: nvm_journal
 1: for entry in nvm_journal do
 2:     for outofdate_page in entry.odpages do
 3:         if outofdate_page is latest then
 4:             sync_to_disk (outofdate_page)
 5:         end if
 6:     end for
 7: end for
 8: flush_consistent_pages (nvm_pagecache)
 9: for entry in nvm_journal do
10:     redo_metadata (entry)
11: end for
12: clean_nvm_journal (nvm_journal)
13: clean_nvm_pagecache (nvm_pagecache)
14: end recovery
```

4.4.4 Consistency level

HasFS provides consistency similar to EXT4 with journal data mode. In the journal data mode, EXT4 will write the dirty metadata and data to the journal before they are written to their original locations, so it can recover both metadata and data in the journal after system crash. In HasFS, dirty metadata and data will be periodically flushed to NVM, and the NVM journaling and LCOW processes ensure that metadata and data can be recovered to a consistent state after system crash.

5 File operations

The I/O paths in HasFS are different from that of the traditional DRAM-based file system. We show the I/O paths of the main file operations in this section.

5.1 Write

In the data write I/O, the dirty data will be buffered in the DRAM write buffer and wait to be flushed to the NVM. The conditions for buffer committing include any one of the following: (1) write buffer is full and (2) a new entry is created in the NVM journal. The metadata write IOs are always applied on DRAM. When the journal or checkpoint process is triggered, dirty metadata will be flushed to the NVM journal or disk.

5.2 Read

Data read I/Os have two paths: one from DRAM (if the target data page is in the write buffer), and the other from NVM (if the page is cached in the NVM page cache). We recognize that the read performance of DRAM and NVM is at the same level, so HasFS does not cache pages on DRAM in read operations. At the same time, this design can take full advantage of the capacity advantages of NVM to expand the main memory capacity. The metadata read I/O is similar to the traditional DRAM-based file systems because metadata is stored in the DRAM.

5.3 Fsync

Fsync is a costly operations in traditional file system. One of the reasons is that fsync needs to flush dirty metadata and data to the disk synchronously. Benefiting from the NVM page cache design, HasFS needs to persist only the dirty pages on NVM and then create a new entry in the NVM journal when fsync is called. Fsync does not cause disk synchronization in HasFS so it is much more efficient than traditional file systems.

6 Implementation

We implemented an emulation file system based on the EXT4 in Linux kernel 4.11.1. We use the same method as [24] to simulate NVM. We use the x86 RDTSCP instruction to read the processor timestamp counter to emulate the NVM delay. In our prototype, we do not separate DRAM and NVM space at physical addresses. Instead, we use an NVM page structure to indicate these two different spaces. Similar to some studies [24, 37], we add 200 ns latency to emulate the write latency of NVM in some NVM write operations, such as DRAM buffer flushing and journaling. In order to evaluate the performance when maximizing the use of NVM space, we do not accurately control the size of the NVM in our prototype. We believe the capacity of NVM will be much larger than DRAM in the future. In fact, the NVM size is approximately equal to the total size of the DRAM minus the DRAM write buffer size and other memory overhead of the kernel.

We implemented the write buffer, NVM page management module and consistency mechanism in EXT4. The NVM page management module is used to manage different types of pages on NVM. EXT4 uses JBD2 to guarantee the crash consistency, but JBD2 is designed for disks and it is inconvenient to implement our design directly by modifying JBD2. To implement the consistency mechanism, we discarded JBD2 and implemented our mechanism in the memory. Similar to EXT4, the journaling period of HasFS was set to 5 seconds and the size of DRAM write buffer was set to 20 MB.

In addition, we reset the parameters related to the background flushing in the proc file system to disable the periodic synchronization. For the disk management, traditional file systems have sophisticated strategies. We reuse the disk management of EXT4 to manage the files' data in our the prototype. Our work focuses more on the designs of the hybrid memory.

7 Evaluation

In this section, we show the evaluation results of HasFS and answer the following questions:

- How does HasFS perform against existing file systems?
- How is the performance of HasFS for different workloads?
- How is the performance of HasFS for different hardware?
- How expensive is the recovery of HasFS?

We ran microbenchmarks and marcobenchmarks on HasFS and compared it with two mainstream file systems to answer the first two questions. To answer the third question, all experiments were run in both HDD and SSD scenarios. Finally, we evaluated the recovery efficiency on three different workloads to answer the latest question.

7.1 Experimental setup

All experimental results were collected on a Dell Optiplex 7040 with a 4-core 4.40 GHz Intel Core i7 CPU and 16 GB DRAM. We used Filebench [14] in our evaluation. We used DRAM to simulate NVM because real NVM devices are not yet available to us. We ran DRAM-based file systems in full DRAM architecture (16 GB DRAM), and ran HasFS in DRAM–NVM architecture (total capacity is 16 GB). We compared HasFS with EXT4 and XFS. EXT4 was evaluated in three different journal modes: journal data ('ext4-data'), ordered data ('ext4-ordered') and writeback ('ext4-wb'). We also evaluated EXT4 without journal ('ext4-non').

Some studies leveraged NVM to optimize the journaling [5, 40]. We did not add them to our comparison, because the experiments in their papers show that their performance can not exceed EXT4 with no journal setting for many workloads. Unlike these systems, HasFS not only optimizes the journaling but also reduces the overhead of periodic synchronization. Our evaluations show that HasFS performs better than EXT4 without journal in many cases. At the same time, thanks to the NVM page cache and the novel consistency mechanism, HasFS has a more efficient fsync mechanism than them. So HasFS has advantages over these NVM journal file systems.

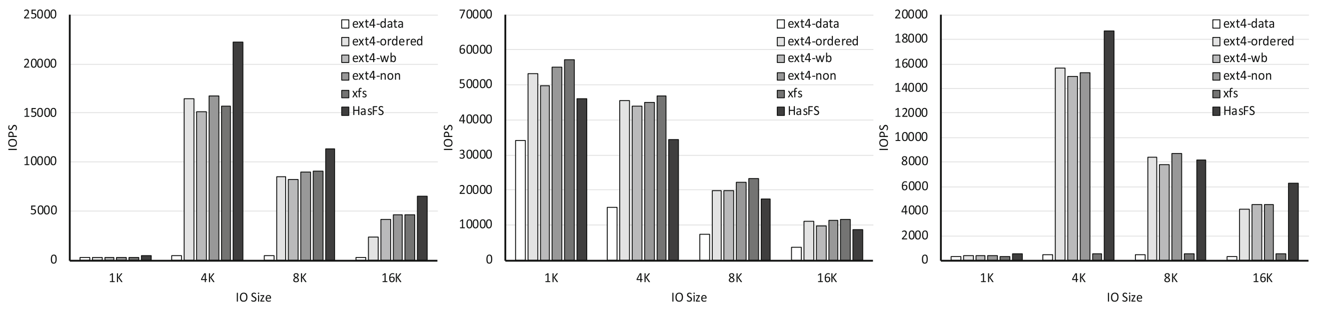
Some NVM file systems use NVM as long-term persistent storage to replace disk (e.g. NOVA [37]). They can get high throughput because they keep everything on NVM. However, these NVM file systems face the cost and capacity limitations of NVM as we described in Sect. 2. HasFS has different design goals than these NVM file systems, it still uses disk as long-term persistent storage and leverages NVM to tackle the bottlenecks of traditional file systems. Unlike these NVM file systems, which primarily pursue performance, HasFS aims to make a tradeoff between performance, cost, and capacity.

7.2 Mircobenchmarks

We ran three single-thread microbenchmarks (sequential write, random write, and random read write) for 30 seconds on a preallocated file, and then compared the performance of EXT4, XFS, and HasFS in different request sizes. In order to evaluate the performance on different file sizes, we evaluated the performance on 20 GB and 100 MB files (DRAM size or hybrid memory size is 16 GB).

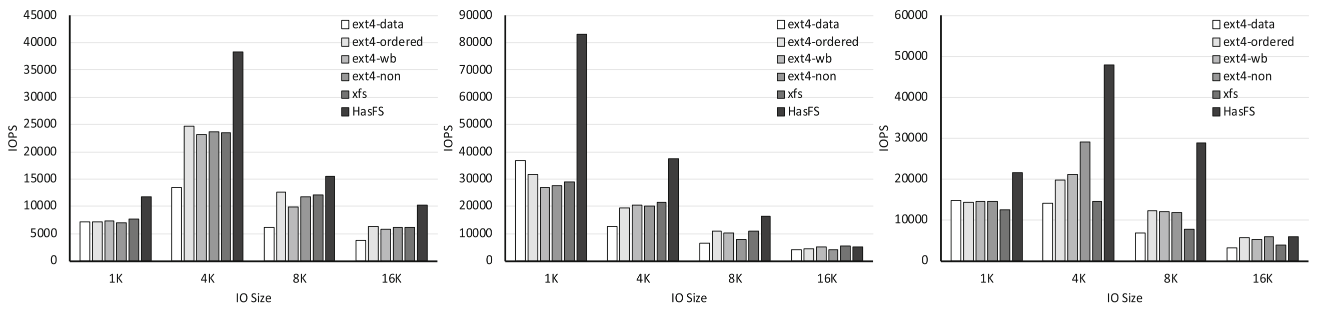
Figures 7 and 8 show the results of a large file (20 GB) mircobenchmark (the test file can not fit the memory). With the large file random write workload, HasFS outperformed other tested file systems by 1.6X to 46.6X on HDD, and 1.7X to 2.9X on SDD. The reason of the performance increase is that the random write workload generates many dirty pages that are scattered across various locations on the disk. Therefore, the periodic synchronization overhead is large with large file random writes, thus, HasFS works well for this workload. HasFS is better than EXT4 without journal, even though HasFS provides the consistency guarantee similar to EXT4 with journal data mode. HasFS greatly improves over EXT4 with journal data mode, especially in the HDD case.

For large file sequential writes workload, HasFS outperforms other tested file systems by 1.3X to 3.1X on the SSD case, but it has no advantage in HDD case. A reason of this result is that the operating system has done a lot of



(a) Random Write **(b)** Sequential Write **(c)** Random Read Write

Fig. 7 Large file (20 GB) microbenchmarks on HDD



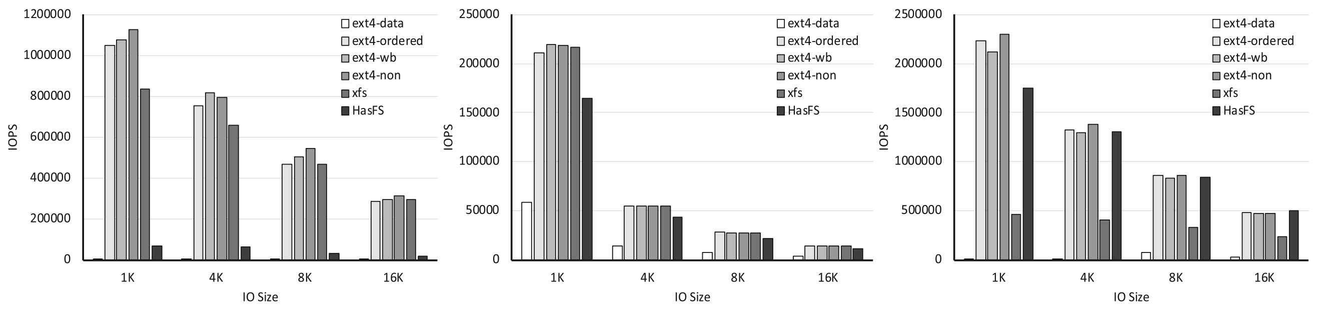
(a) Random Write **(b)** Sequential Write **(c)** Random Read Write

Fig. 8 Large file (20 GB) microbenchmarks on SSD

optimizations for sequential writes (such as I/O aggregation) and the sequential write performance of HDD is good. So the disk I/O overhead is not as high as the random write case. Unlike HDD, there is no advantage in sequential writing on SSD because the hardware features and the FTL (Flash Translation Layer), so HasFS has performance advantage in the SSD case.

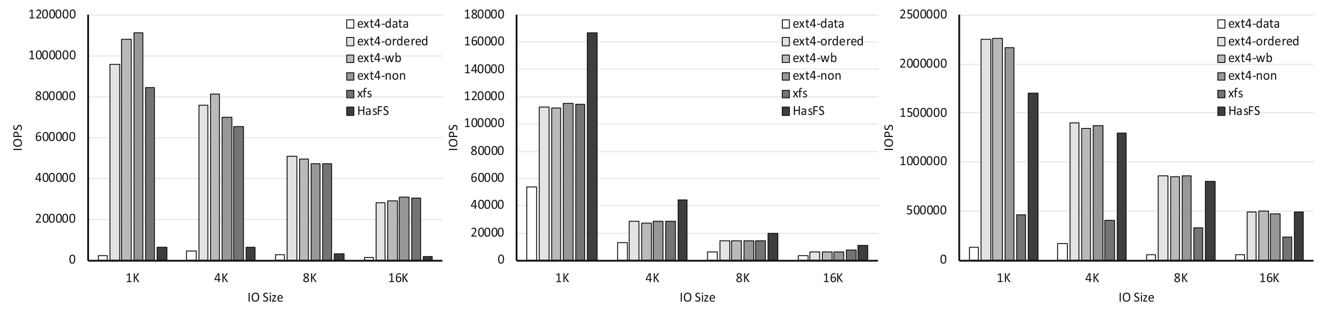
For large file random read write workload, HasFS also has performance advantage compared to other tested systems. The performance improvement mainly comes from random writes. For the read operations, since the read latency of NVM close to DRAM so we did not emulate it in our prototype.

Figures 9 and 10 show the results of the small file (100 MB) mircobenchmark. In the small file case, HasFS is worse than other tested file systems in many workloads, except EXT4 with journal data mode. The reasons are as follows. (1) The number of dirty pages is not as large as the large file scenario, so the costs of journaling and periodic synchronization are minimal, indicating that the advantage of hybrid memory is not obvious in small file workloads. (2) The file fits within the memory, so the impacts on the NVM latency and NVM persistent operation (clflush & mfence) are large. However, HasFS still outperforms EXT4 with journal data mode by more than 2.8X on HDD, and more than 1.3X on SSD. In journal data mode, EXT4 will



(a) Random Write **(b)** Sequential Write **(c)** Random Read Write

Fig. 9 Small file (100 MB) microbenchmarks on HDD



(a) Random Write **(b)** Sequential Write **(c)** Random Read Write

Fig. 10 Small file (100 MB) microbenchmarks on SSD

log all dirty data and commits the journal frequently, so the disk I/O overhead is still large in the EXT4 with journal data mode, even though the tested file fits in the memory.

7.3 Marcobenchmarks

We chose four macrobenchmarks in Filebench to evaluate HasFS's performance in different workloads. Table 3 summarizes the characteristics of these workloads. Figures 11 and 12 show the performance comparison. In particular, we added EXT4 with NVM journal ('ext4-nvmj') into this experiment. We set the NVM journal size to 1GB and used journal data mode in this case. It should be noted that the we simply used DRAM to emulate NVM without adding latency in the NVM journal, whereas HasFS we added the latency of write operations.

For the fileserver workload, HasFS outperforms other tested file systems by up to 2.9X on the HDD case. On the SSD case, HasFS's performance is close to some file systems but it still has about 1.7X performance improvement over the EXT4 with journal data mode. HasFS is optimized for write operations but the fileserver workload includes some other file system operations (like file creating and file deleting), which is why the performance improvement is not as obvious as the mircbenchmark experiments.

For the webserver workload, HasFS has a performance advantage of more than 1.7X compared to other systems in the HDD case, but their performance is close in the SSD case. The result for the webproxy workload is similar to webserver, HasFS has 2.1X to 2.3X performance improvement in the HDD case, and 1.2X to 1.5X improvement in the SSD case.

For the varmail workload, HasFS has more than 4.4X performance improvement over traditional file systems (EXT4 with traditional journal and XFS). The traditional file systems need to synchronize the dirty metadata and data to disk when fsync is called. Benefiting from the persistent page cache design of HasFS, we can replace synchronizing the dirty data to disk by the persisting them on NVM and write the dirty metadata to the NVM journal. EXT4 with NVM journal performs better than HasFS in the HDD case. One of the possible reasons is that checkpointing rarely occurs when using large journal size, so checkpointing is more frequent in HasFS. Also, we did not add NVM latency in EXT4 with NVMJ.

7.4 Recovery

We emulated HasFS recovery overhead for three workloads: fileserver, webserver, and webproxy. We emulated the system crash when the journal is half full. Table 4 illustrates the results. We used a single recovery thread in this experiment. The recovery efficiency is determined by many factors, such as the workload and hardware. The main cost of recovery is from synchronization of "latest out-of-date" pages to disk. In addition to "latest out-of-date" pages, it may be necessary to flush dirty consistent pages to disk when recovering, but we did not emulate this

Table 3 Workloads characteristics

Workloads	Average file size (KB)	IO size (MB)	Total size (GB)	Main operations
Fileserver	64	1	30	Create write, append, read, state, delete
Webserver	64	1	30	Read, append
Webproxy	32	1	30	Create, append, delete
Varmail	32	1	30	Create, append, read, fsync

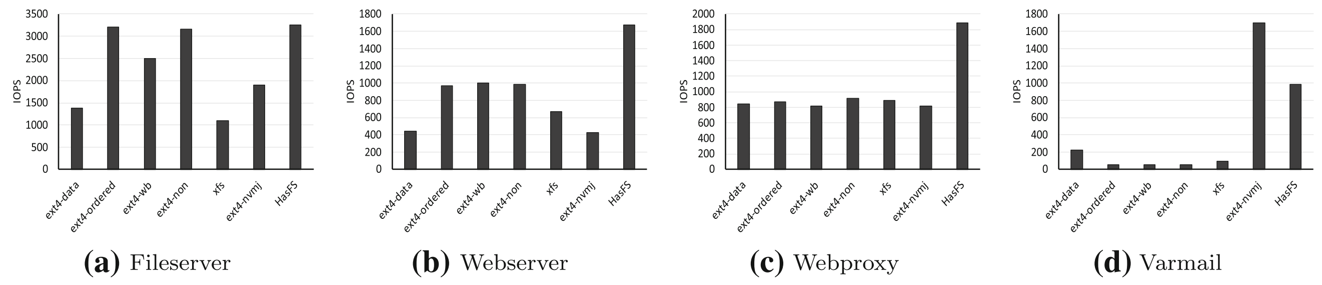


Fig. 11 Macrobenchmarks on HDD

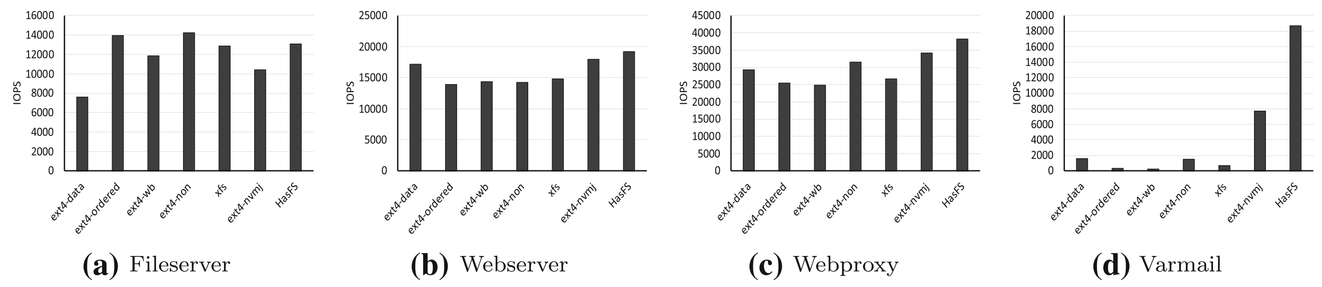


Fig. 12 Macrobenchmarks on SSD

Table 4 Recovery overhead

Workloads	HDD (s)	SSD (ms)
Fileserver	1.67	302.3
Webserver	3.13	267.7
Webproxy	1.88	232.1

situation because their recovery depends on the design of the NVM allocator. Webserver and webproxy workloads contain a large number of small files and the request size is small, so the recovery efficiency for these workloads is lower than for fileserver.

8 Related work

8.1 File system consistency

There are three mainstream consistency mechanisms: copy-on-write, log-structure, and journaling. In disk-based file system, ZFS [2] uses copy-on-write; [28] uses log-structure; EXT4 [13], ReiserFS [26] and XFS [31] use journaling. In NVM-based file systems, HiNFS [24] uses journaling as its consistency mechanism; BPFS [10] uses copy-on-write; NOVA [37] uses log-structure. There are many studies dedicated to reducing the overhead of consistency guarantee in file system. NoFS [8] uses a consistency mechanism without ordering guarantee, but it assumes hardware support for 4 KB atomic writes. Another study [7] proposed an optimistic crash consistency method in a disk-based file system. Shortcut-jfs [21] is designed for reducing the journaling overhead in the main memory file systems.

8.2 NVM-based file systems

Some works use NVM as persistent storage. BPFS [10] uses NVM to replace both DRAM and disk and employed some hardware modification to provide 64-bytes atomic write on NVM. Using the 64-bytes atomic write, BPFS designs an novel shadow paging to protect data consistency. SCMFS [36] utilizes memory management to implement a lightweight NVM file system. PMFS [12] implements a direct write file system on NVM. PMFS also combines journaling and copy-on-write. HiNFS [24] is designed based on PMFS. It buffers the write data on the DRAM to mitigate the write performance gap between DRAM and NVM. NOVA [37] implements a log-structure file system on NVM, and it keeps the indexes in DRAM but the file data in NVM. EXT4-DAX [11, 34] supports deploying EXT4 on NVM. [30] provides an empirical analysis of the mainstream NVM-based file systems. UBJ [20] improves the efficiency of consistency mechanism through an in-place checkpoint technique. Instead of replacing disk with NVM like in these related works, we believe that DRAM, NVM, and disk are more suitable for coexistence in some scenarios. Therefore, the design goal is different between the HasFS and the state-of-the-art NVM-based file systems.

8.3 Hybrid memory management

Recent studies[6, 18, 23, 35] proposed several methods to manage DRAM–NVM hybrid memory. Mnemosyne [33] and NV-heap [9] provide programming interfaces for NVM. Mojim [41] is designed for building a distributed non-volatile memory system. Kiln [42] is designed for implementing the memory transaction on NVM based on a hardware method. These works provide some efficient and safe ways to use NVM. These studies can be used in the implementation of HasFS.

8.4 Others

Local file system is a foundational component of many systems, such as distributed file systems and databases. For example, many parallel distributed file systems (e.g. Lustre [29], BeeGFS [1]) use local file system to store metadata and data on each node and many database systems (e.g. LevelDB) store the tables on local file system. Therefore, mounting the local file system on the high-speed storage device (e.g. NVM) is an effective way to improve performance, but this approach is limited by capacity and price. HasFS provides a new way to efficiently utilize NVM.

9 Conclusion

In this paper, we presented HasFS, a file system designed for DRAM–NVM–DISK hybrid storage architecture. We designed an efficient consistency mechanism based on the hybrid architecture. HasFS uses NVM as a data page cache to eliminate periodic synchronization of dirty data and take advantage of the capacity of NVM. To reduce consistency overhead, HasFS logs metadata on NVM journal and uses lazy copy-on-write for data. Our measurements show that HasFS outperforms the mainstream DRAM-based file systems in many operations, while providing metadata and data consistency (version consistency).

Funding This work was supported by 2016YFB1000302 (National Key R&D Program of China), U1611261 (National Natural Science Foundation of China), 61433019 (National Natural Science Foundation of China), 61872392 (National Natural Science Foundation of China) and 2016ZT06D211 (Program for Guangdong Introducing Innovative and Entrepreneurial Teams).

References

1. beegfs.io: Beegfs. https://www.beegfs.io (2019)
2. Bonwick, J., Moore, B.: Zfs: The last word in file systems (2007)
3. btrfs.org: Btrfs. https://btrfs.wiki.kernel.org (2018)
4. Burr, G.W., Breitwisch, M.J., Franceschini, M., Garetto, D., Gopalakrishnan, K., Jackson, B., Kurdi, B., Lam, C., Lastras, L.A., Padilla, A., et al.: Phase change memory technology. J. Vac. Sci. Technol. B **28**(2), 223–262 (2010)
5. Chen, C., Yang, J., Wei, Q., Wang, C., Xue, M.: Fine-grained metadata journaling on NVM. In: Proceedings of IEEE Conference on MASS Storage Systems and Technologies (MSST), pp. 1–13 (2016)
6. Chen, F., Mesnier, M.P., Hahn, S.: A protected block device for persistent memory. In: Proceedings of IEEE Conference on MASS Storage Systems and Technologies (MSST), pp. 1–12 (2014)
7. Chidambaram, V., Pillai, T.S., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H.: Optimistic crash consistency. In: Proceedings of the 24 ACM Symposium on Operating Systems Principles (SOSP), pp. 228–243 (2013)
8. Chidambaram, V., Sharma, T., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H.: Consistency without ordering. In: Proceedings of USENIX Conference on File and Storage Technologies (FAST), p. 9 (2012)
9. Coburn, J., Caulfield, A.M., Akel, A., Grupp, L.M., Gupta, R.K., Jhala, R., Swanson, S.: Nv-heaps: making persistent objects fast and safe with next-generation, non-volatile memories. ACM Sigplan Not. **46**(3), 105–118 (2011)
10. Condit, J., Nightingale, E.B., Frost, C., Ipek, E., Lee, B., Burger, D., Coetzee, D.: Better i/o through byte-addressable, persistent memory. In: Proceedings of the 22nd symposium on Operating systems principles (SOSP), pp. 133–146 (2009)
11. Corbet, J.: Supporting filesystems in persistent memory. https://lwn.net/Articles/610174/ (2014)
12. Dulloor, S.R., Kumar, S., Keshavamurthy, A., Lantz, P., Reddy, D., Sankaran, R., Jackson, J.: System software for persistent memory. In: Proceedings of European Conference on Computer Systems (EuroSys), p. 15 (2014)
13. ext4.org: Ext4. https://ext4.wiki.kernel.org (2016)
14. Filebench: Filebench. https://github.com/filebench/filebench (2018)
15. Intel: Intel. http://newsroom.intel.com/community (2015)
16. Izraelevitz, J., Yang, J., Zhang, L., Kim, J., Liu, X., Memaripour, A., Soh, Y.J., Wang, Z., Xu, Y., Dulloor, S.R., et al.: Basic performance measurements of the intel optane dc persistent memory module. arXiv preprint arXiv:1903.05714 (2019)
17. Kawahara, T.: Scalable spin-transfer torque ram technology for normally-off computing. IEEE Des. Test Comput. **1**, 52–63 (2010)
18. Lantz, P., Rao, D.S., Kumar, S., Sankaran, R., Jackson, J.: Yat: A validation framework for persistent memory software. In: Proceedings of USENIX Technical Conference (ATC), pp. 433–438 (2014)
19. Lee, B.C., Zhou, P., Yang, J., Zhang, Y., Zhao, B., Ipek, E., Mutlu, O., Burger, D.: Phase-change technology and the future of main memory. IEEE Micro **30**(1), 143 (2010)
20. Lee, E., Bahn, H., Noh, S.H.: Unioning of the buffer cache and journaling layers with non-volatile memory. In: Proceedings of USENIX Conference on File and Storage Technologies (FAST), pp. 73–80 (2013)
21. Lee, E., Yoo, S., Jang, J.E., Bahn, H.: Shortcut-jfs: a write efficient journaling file system for phase change memory. In: Proceedings of IEEE Conference on MASS Storage Systems and Technologies (MSST), pp. 1–6. IEEE (2012)
22. McKusick, M.K., Joy, W.N., Leffler, S.J., Fabry, R.S.: A fast file system for unix. ACM Trans. Comput. Syst. (TOCS) **2**(3), 181–197 (1984)
23. Moraru, I., Andersen, D.G., Kaminsky, M., Tolia, N., Ranganathan, P., Binkert, N.: Consistent, durable, and safe memory management for byte-addressable non volatile main memory. In:

Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems, p. 1 (2013)
24. Ou, J., Shu, J., Lu, Y.: A high performance file system for non-volatile main memory. In: Proceedings of European Conference on Computer Systems (EuroSys), p. 12 (2016)
25. Oukid, I., Lasperas, J., Nica, A., Willhalm, T., Lehner, W.: Fptree: A hybrid SCM-DRAM persistent and concurrent b-tree for storage class memory. In: Proceedings of International Conference on Management of Data (SIGMOD), pp. 371–386 (2016)
26. reiser4.org: Reiserfs. https://reiser4.wiki.kernel.org (2018)
27. Rodeh, O., Bacik, J., Mason, C.: Btrfs: the linux b-tree filesystem. ACM Trans. Storage (TOS) **9**(3), 9 (2013)
28. Rosenblum, M., Ousterhout, J.K.: The design and implementation of a log-structured file system. ACM Trans. Comput. Syst. (TOCS) **10**(1), 26–52 (1992)
29. Schwan, P., et al.: Lustre: Building a file system for 1000-node clusters. In: Proceedings of the 2003 Linux symposium, pp. 380–386 (2003)
30. Sehgal, P., Basu, S., Srinivasan, K., Voruganti, K.: An empirical study of file systems on nvm. In: Proceedings of IEEE Conference on MASS Storage Systems and Technologies (MSST), pp. 1–14 (2015)
31. Sweeney, A., Doucette, D., Hu, W., Anderson, C., Nishimoto, M., Peck, G.: Scalability in the XFS file system. In: Proceedings of USENIX Annual Technical Conference (ATC), vol. 15 (1996)
32. Venkataraman, S., Tolia, N., Ranganathan, P., Campbell, R.H., et al.: Consistent and durable data structures for non-volatile byte-addressable memory. In: Proceedings of USENIX Conference on File and Storage Technologies (FAST), pp. 61–75 (2011)
33. Volos, H., Tack, A.J., Swift, M.M.: Mnemosyne: Lightweight persistent memory. In: Proceedings of Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pp. 91–104 (2011)
34. Wilcox, M.: Support ext4 on nv-dimm. https://lwn.net/Articles/588218/ (2014)
35. Wu, M., Zwaenepoel, W.: envy: a non-volatile, main memory storage system. In: ACM SIGOPS Operating Systems Review, pp. 86–97 (1994)
36. Wu, X., Reddy, A.: SCMFS: a file system for storage class memory. In: Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC), p. 39 (2011)
37. Xu, J., Swanson, S.: Nova: a log-structured file system for hybrid volatile/non-volatile main memories. In: Proceedings of USENIX Conference on File and Storage Technologies (FAST), pp. 323–338 (2016)
38. Yang, J., Wei, Q., Chen, C., Wang, C., Yong, K.L., He, B.: NC-tree: Reducing consistency cost for NVM-based single level systems. In: Proceedings of USENIX Conference on File and Storage Technologies (FAST), pp. 167–181 (2015)
39. Yang, J.J., Williams, R.S.: Memristive devices in computing system: promises and challenges. ACM J. Emerg. Technol. Computi. Syst. (JETC) **9**(2), 11 (2013)
40. Zhang, X., Feng, D., Hua, Y., Chen, J.: Optimizing file systems with a write-efficient journaling scheme on non-volatile memory. IEEE Trans. Comput. **68**(3), 402–413 (2018)
41. Zhang, Y., Yang, J., Memaripour, A., Swanson, S.: Mojim: a reliable and highly-available non-volatile memory system. In: Proceedings of Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pp. 3–18 (2015)
42. Zhao, J., Li, S., Yoon, D.H., Xie, Y., Jouppi, N.P.: Kiln: Closing the performance gap between systems with and without persistence support. In: Proceedings of 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pp. 421–432 (2013)

Yubo Liu is currently a Ph.D. student at Sun Yat-sen University and Arizona State University. His research interests include local file system, distributed and parallel file system, non-volatile memory and operating system.

Hongbo Li is currently a graduate student of School of Data and Computer Science, in Sun Yat-sen University. His interests include file systems, storage systems.

Yutong Lu received the B.S., M.S., and Ph.D. degrees from National University of Defense Technology. She is a professor at Sun Yat-sen University and the director of National Supercomputer Center in Guangzhou. Her current research interest includes large-scale storage system, high-performance computing, and computer architecture.

Zhiguang Chen received the B.S. degree from Harbin Institute of Technology, and the M.S. and Ph.D. degree from National University of Defense Technology. He is an associate professor at Sun Yat-sen University. His current research interest includes distributed file system, network storage, and solid-state storage system.

Nong Xiao received the B.S., M.S., and Ph.D. degrees of computer science from National University of Defense Technology. Now he is a professor at Sun Yat-sen University. His current research interest includes large-scale storage system, network computing, and computer architecture.

Ming Zhao received the B.S. and M.S. degrees in Automation/ Pattern Recognition and Intelligent Systems from Tsinghua University, China in 1999 and 2001 respectively, and the Ph.D. degree in Electrical and Computer Engineering from the University of Florida, USA in 2008. He is now an associate professor in the School of Computing, Informatics, and Decision Systems Engineering at Arizona State University, USA. His research interests include distributed/cloud computing, high-performance computing, virtualization, storage systems, and operating systems.