Spindle: A Write-Optimized NVM Cache for Journaling File System

Ge Yan, Kaixin Huang, and Linpeng Huang(✉)

Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai, China
{bueryg,kaixinhuang,lphuang}@sjtu.edu.cn

Abstract. Journaling techniques are widely employed in modern file systems to guarantee crash consistency. However, journaling usually leads to system performance decrease due to the frequent storage accesses it entails. Architects can utilize emerging non-volatile memory (NVM) as a persistent cache or journaling device to reduce the storage accesses of journaling file systems. Yet problems such as double writes, metadata write amplification and heavy transaction ordering overhead still exist in current solutions. Therefore, we propose Spindle, a write-optimized NVM cache to address these challenges. Spindle decouples data and metadata accesses by processing data in DRAM while pinning metadata in NVM. With redesigned metadata log and state switch mechanism, Spindle eliminates double writes and relieves metadata write amplification. Moreover, Spindle adopts a lightweight transaction scheme to guarantee crash consistency and reduce transaction ordering overhead. Experimental results reveal that Spindle achieves up to 47% throughput improvement compared with state-of-the-art design.

Keywords: File system · Non-volatile memory · Journaling · Data consistency

1 Introduction

Crash consistency is a significant feature that enables file systems to recover to a consistent state after unexpected system crashes or power failures. Journaling is a prevalent technique adopted by modern file systems, such as ext4 [11] and JFS [1], to maintain crash consistency. For example, redo journaling first writes the modified data to the journal area during the committing of a transaction. After the redo log is successfully committed, the modified data can be written to its desired location via checkpointing.

Although journaling can guarantee crash consistency, it significantly degrades system performance due to the heavy overheads entailed by frequent storage accesses. Precisely, journaling needs to write two blocks (i.e., the committed block and the checkpointed block) for every modified block and hence results in double disk I/Os. This problem is known as double writes of journaling.

© IFIP International Federation for Information Processing 2019
Published by Springer Nature Switzerland AG 2019
X. Tang et al. (Eds.): NPC 2019, LNCS 11783, pp. 251–263, 2019.
https://doi.org/10.1007/978-3-030-30709-7_20

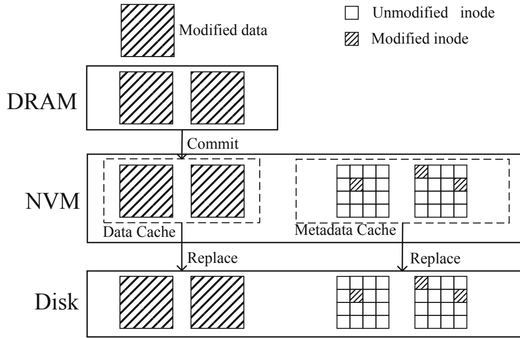


Fig. 1. Architecture of Spindle

Many in-memory file systems, such as BPFS [2], SCMFS [16], PMFS [4], NOVA [18], and HMVFS [20], have been proposed to utilize NVM as a replacement of disk. Since NVM provides persistency, byte-addressability and DRAM-like latency, these file systems avoid the bottleneck of disk I/O and provide high performance. However, two factors limit the utilization of these file systems in real environments. First, the price of NVM is predicated to be much higher than that of traditional storage (e.g., HDD and SSD). Second, current NVM technologies without broad application may be less stable than traditional storage.

To overcome the concerns discussed above, some researchers have investigated using NVM as a middle layer to accelerate disk-based file systems. For instance, UBJ [6] utilizes NVM to build a buffer cache united with the journaling layers; Tinca [15] proposes a transactional NVM cache with high performance and crash consistency. Although these systems utilize the low latency of NVM and the low cost of disk, write amplification and heavy transaction ordering overhead still exist due to their block-based updating strategy. Concretely, UBJ commits data in-place in NVM by freezing data and later checkpoints them to the disk. When updating a frozen block, UBJ can not overwrite it but must do it in a copy-on-write (COW) way, which incurs double writes on the critical path. Tinca does role switch to change the roles of modified blocks committed to the NVM cache. However, it does not differentiate metadata blocks from data blocks and metadata write amplification remains unsolved. Besides, it needs massive executions of cache line flush (e.g., clflush) and memory fence (e.g., mfence) to change the roles of modified blocks involved in the committing transaction.

With such observations, we propose Spindle, a write-optimized NVM cache for journaling file systems, as shown in Fig. 1. The key design of Spindle is decoupling data and metadata accesses. Data is processed in DRAM and committed to the NVM cache in the unit of block. Yet metadata is pinned in NVM and updated in the unit of inode. Such a design fully utilizes the byte-addressability of NVM and different updating granularities of data and metadata. Our main contributions can be summarized as follows.

- We propose a novel design to decouple data and metadata accesses by processing data in DRAM while pinning metadata in NVM at runtime. By redesigning the log and updating strategy of metadata, Spindle can efficiently reduce metadata writes to relieve metadata write amplification.
- We provide a lightweight transaction scheme to guarantee the consistency of data and metadata. Utilizing state switch and COW updating, Spindle can avoid double writes and reduce the transaction ordering overhead.
- We implement Spindle on ext4 file system and evaluate its performance with several benchmarks. Experimental results show that Spindle achieves up to 47% throughput improvement compared with state-of-the-art design.

The remainder of this paper is organized as follows. Section 2 introduces the background and motivation. Sections 3 and 4 describe the design and implementation of Spindle, respectively. Section 5 evaluates Spindle. We discuss related work in Sect. 6 and conclude this paper in Sect. 7.

2 Background and Motivation

2.1 NVM-based Systems

Emerging non-volatile memory technologies, such as STT-RAM [14], ReRAM [17], PCM [19], and 3D-XPoint [3], exhibit high storage density and low power consumption. Moreover, they can provide persistency, byte-addressability, DRAM-like latency and throughput. These properties enable architects to build fast and persistent systems with NVM.

Current NVM-based file systems can be classified into two categories. The first kind utilizes NVM as a replacement of disk, such as BPFS [2], SCMFS [16], PMFS [4], NOVA [18] and HMVFS [20]. Since the storage is substituted by NVM, these systems can exploit the properties of NVM to provide low latency and high throughput. The other one exploits NVM as a persistent cache or journaling device to improve the performance of disk-based file systems, such as UBJ [6] and Tinca [15]. In these systems, data is temporarily stored in NVM for fast accesses and eventually flushed to disk when data replacement is executed.

2.2 Crash Consistency

File systems should guarantee the consistency of data and metadata due to unpredictable power failures and system crashes. In-place updating is unfavored by file systems because a crash amid the updating process may cause inconsistency issue. Copy-on-write (COW) and journaling are two prevalent techniques for crash consistency. COW updates data out of place and substitutes the original data by changing respective metadata. Unlike COW, journaling does not change the metadata of a file but reserves backup copies of data for crash recovery.

The consistency of file system can be classified into metadata consistency, data consistency and version consistency. At the low level, metadata consistency only provides the consistency of metadata, while the consistency of data

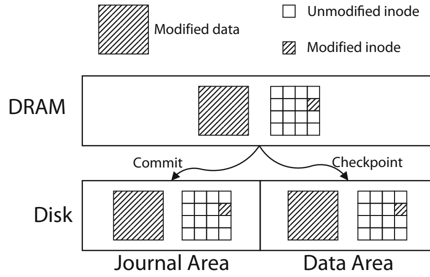


Fig. 2. Double writes and metadata write amplification

is ignored. At a higher level, data consistency guarantees that both data and metadata of a file system are consistent. Version consistency is an even higher level that ensures an old version of the file can be retrieved. This paper targets data consistency that can recover both data and metadata to a consistent state.

2.3 Motivation

Double Writes of Journaling. In journaling file systems, a committed block and corresponding checkpointed block have the same contents, but write operations are done twice. This problem is known as double writes of journaling. When NVM is provided as a persistent cache for disk-based file systems, double writes of journaling will lead to double memory copies and double executions of cache line flush and memory fence. Since data blocks in the NVM cache are flushed to the disk eventually, double writes of journaling also entail amplified disk I/O. Therefore, performance can get benefits if double writes can be eliminated.

Metadata Write Amplification. Metadata is organized in the unit of block according to the design of traditional journaling file systems. Partial-updated metadata results in whole-block write that seriously amplifies data I/O. For example, Tinca [15] writes data and metadata into NVM in the unit of block without differentiation. Suppose that an inode is 256B and a block is 4 KB, Tinca needs to write 4 KB data into NVM when only one inode is modified in the metadata block. Such a strategy leads to 16× execution time compared to solely updating an inode of 256B. Moreover, it squanders the byte-addressability of NVM. Therefore, it is desirable to reduce metadata write amplification to improve the overall performance. Figure 2 illustrates the issues of double writes and metadata write amplification in journaling file systems.

Transaction Ordering Overhead. Current CPU may reorder writes to the memory to optimize system performance [12]. A crash that happens amid reordered writes may cause inconsistency issue. For example, modification of file metadata should be performed after the updating of file data. If metadata is modified before the updating of file data and a crash happens, then the file will

be inconsistent. To maintain the desired writing order, one widely-used method is to use regular store instructions (e.g., mov) followed by cache line flush and memory fence. Tinca adopts this method and entails massive executions of cache line flush and memory fence to change the roles of modified blocks.

3 Design

3.1 Cache Layout

As shown in Fig. 1, Spindle[1] consists of a data cache and a metadata cache.

Data Cache. Details of the data cache are depicted in Fig. 3(a). The data cache is made up of three components: the ring buffer, the data cache entries and cached data blocks. The ring buffer is used to coordinate a transaction and can be viewed as an array of 16-byte elements. Two pointers, `Head` and `Tail`, are provided to use the ring buffer in a round-robin way. Data cache entries with 16-byte elements are used for address mapping and crash recovery. The last part contains cached data blocks delivered by the file system.

Metadata Cache. Figure 3(b) shows the layout of metadata cache. The first part is a metadata log area with the granularity of inode size. After the metadata log area, there are multiple metadata cache entries that have the same structure with entries in the data cache. The ring buffer entries, data and metadata cache entries consist of an 8-byte disk block number and an 8-byte NVM block number. The major difference between these entries is that metadata cache entries are designed for metadata blocks while the others are used by data blocks. The last part is an area with cached metadata blocks.

A metadata block can be divided into many sub-blocks of inode size. Each running transaction in Spindle has a modified data block list and an original metadata sub-block list. When a metadata sub-block is going to be updated, we copy it to the log area and link it to the metadata sub-block list. Then we can perform in-place updates on the sub-block. When a transaction is successfully committed, related metadata sub-blocks in the log area can be removed.

3.2 Lightweight Transaction

Spindle provides a lightweight transaction scheme to ensure the consistency of file data. In particular, it exploits state switch and COW updating to avoid double writes on the critical path.

State Switch. In journaling file systems, a disk block has the state of being *committed* or *checkpointed*. When it is written to the journal, its state is *committed*. After it is flushed to its original location, its state becomes *checkpointed*. We utilize state switch to change the state of a cached data block. In our system, a data block with a ring buffer entry has the state of being *committed*,

[1] The memory hierarchy looks like a spindle with the data cache and the metadata cache interposed between DRAM and the disk.

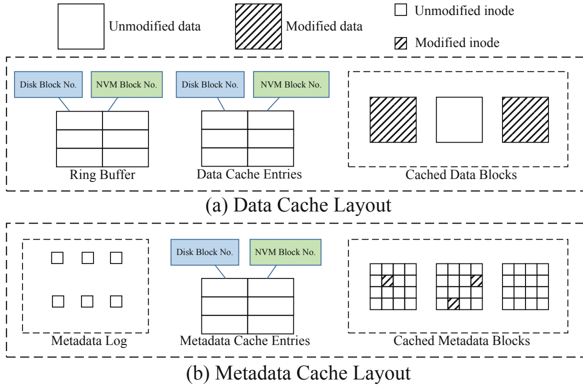


Fig. 3. Data and metadata cache layout of Spindle

which indicates it is involved in a committing transaction. After the transaction is successfully committed, corresponding ring buffer entry can be removed and the data block is switched to *checkpointed* state.

COW Updating. We adopt COW to write data into the data cache. Before writing a data block, we first search the data cache entries. If it has been cached, we copy its data cache entry to the ring buffer. Then we write the data to a new NVM block and update the data cache entry to point to the newly written block. If the block has not been cached, we create a new ring buffer entry with its disk block number and a special `NON` tag. Subsequently, we write the data to a new NVM block and create a new data cache entry. During the committing of a transaction, a cached data block may have two versions at the same time. Once the committing is successfully completed, the old one can be removed and the newly written one remains as persistent cached data.

Transaction Committing. We follow the routine of traditional journaling file systems to coalesce multiple blocks in a transaction. Since the NVM cache provides persistency that a file system desires, the committing of a transaction does not entail disk I/O when free data and metadata blocks can be found in the NVM cache. Before the committing of a transaction, `Head` and `Tail` point to the same entry in the ring buffer. Figure 4 illustrates the committing of a transaction, which can be summarized as follows.

(1) **Write ring buffer entry.** We search the block with the modified data block's disk block number. If it has been cached (i.e., cache hit), we copy its data cache entry to the ring buffer. Otherwise (i.e., cache miss), we create a new ring buffer entry with the disk block number and a special `NON` tag.
(2) **Write data block and data cache entry.** We write the data to a new NVM block and update corresponding data cache entry (cache hit) or create a new one (cache miss).
(3) **Update `Tail` pointer.** We move tail pointer forward by one.

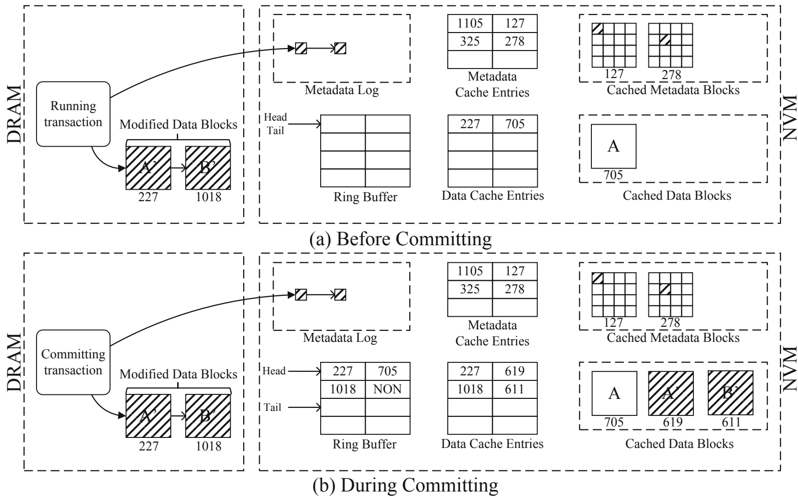


Fig. 4. Transaction committing of Spindle

(4) **Update `Head` pointer.** We repeat (1)–(3) until no block is left in the committing transaction. At last, we set `Head` to be identical to `Tail`.

When `Head` is equal to `Tail`, the committing is completed and the old data blocks can be recycled. Since metadata sub-blocks are first copied to the metadata log area and then in-place updated, we just need to remove the metadata sub-blocks in the log area.

3.3 Data Replacement

The NVM cache can not hold all the data of a file system due to its limited capacity. Therefore, we choose to write cold blocks back to disk. Data replacement is triggered when there is no free data or metadata block in the NVM cache. The replacement strategies for data and metadata are different.

(1) For data, the access unit and replacement unit is the same (i.e., block). We utilize the least-recently-used (LRU) algorithm to select the victim block. But there is an extra limit: data blocks involved in the running or committing transaction are not allowed to be swapped out.
(2) For metadata, the access unit is inode while the replacement unit is block. To select the victim, we maintain a counter for every cached metadata block. When any inode of a metadata block is accessed, the counter of that block is increased by one. If free metadata blocks can not be found in the cache, the block with the least counter value is chosen as the victim.

3.4 Crash Recovery

During crash recovery, we first scan the data and metadata cache entries to build an in-DRAM hash table. Then we scan the ring buffer and process cache entries

involved in the unfinished committing transaction. Since crash could happen before the moving of `Tail`, we need to scan an extra entry behind `Tail`. For every ring buffer entry during the scan, we search the data cache entries with its disk block number. If the ring buffer entry's NVM block number is valid (i.e., not a `NON` tag), the block with this number has been cached and corresponding data cache entry exists. Otherwise, the block has not been cached. For the two cases, we search in the data cache entries. If the data cache entry exists, we revoke it (valid) or delete it (invalid) according to its ring buffer entry. If it does not exist, nothing needs to be done. After all related ring buffer entries have been scanned and processed, we set `Head` to be identical to `Tail`. Eventually, we scan the metadata log area to revoke the modified metadata sub-blocks.

Note that `Head` is not modified until the last step of the recovery. Therefore, any failure during the recovery does not affect the consistency of file data, because the recovery can be redone as long as the ring buffer entries can be retrieved. As for metadata, its consistency is maintained when it is successfully copied to the log area hence a crash during the recovery does not affect the consistency of metadata.

4 Implementation

We implement Spindle on `ext4` file system. The implementation mainly includes in-memory structures, data cache and metadata cache.

In-Memory Structures. A cuckoo hash table is used to accelerate the search with a disk block number for a data or metadata cache entry. Moreover, two LRU lists are managed in DRAM to select the victim to evict. Besides, bitmaps of data and metadata cache are also stored in DRAM. Since these structures can be reconstructed during system reboot, we do not keep them in NVM.

Data Cache. The data cache is implemented based on JBD2 [11]. We alter JBD2's interfaces to realize Spindle's characteristics. In particular, JBD2's descriptor block, revoke block and commit block are substituted by the ring buffer, data cache entries, `Head` and `Tail` pointers. Since the consistency of cached data and metadata is guaranteed when the transaction is successfully committed, the checkpointing function is removed from JBD2.

Metadata Cache. Implementation of the metadata cache primarily consists of three parts. First, metadata fetching and flushing is redesigned. Since metadata is not flushed to the disk unless under space pressure, we modify the implementation of `ext4-handle-dirty-metadata` to prevent metadata from being written back to disk. Second, the allocation and reclamation of metadata is renovated. Utilizing NVM as the metadata cache, we add a set of functions to allocate/free memory from/to the metadata cache. Third, the updating of metadata is substituted by our inode-based updating strategy.

5 Evaluation

5.1 Setup

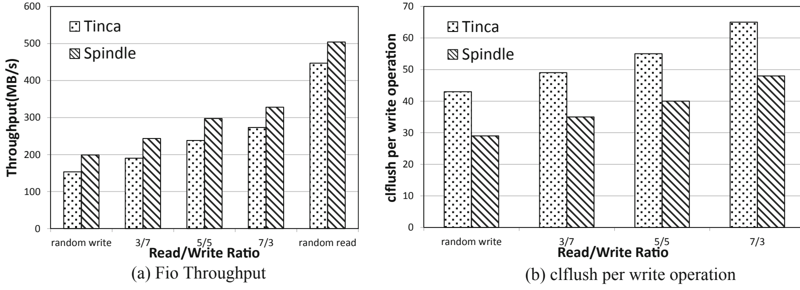
We implement Spindle on Linux 4.18.1 and use `ext4` file system as our code base. All experiments are performed on an Intel Xeon E5 server with 98 GB DRAM and 1 TB HDD. The competitor we use to compare against Spindle is Tinca, which also utilizes NVM as a persistent cache to improve the performance of journaling file systems. Since Tinca is not open-source, we develop a prototype for it according to its implementation [15]. We use `ext4` file system to evaluate Tinca's performance and set the default cache mode as *writeback*. Since real NVM devices are not available to us yet, we add read/write latencies (50ns/180ns) to DRAM to simulate the NVM device. In the evaluation, 8 GB DRAM is used to simulate the NVM cache of Tinca and Spindle. For Spindle, we use 1 GB NVM as the metadata cache and the left 7 GB NVM as the data cache. Characteristics of the benchmarks in our evaluation are summarized in Table 1. All the results are averaged over five runs.

Table 1. Benchmark characteristics

Benchmark	Read/write ratio	Request size	Dataset size	Running time
Fio	0/10,3/7,5/5,7/3,10/0	4 KB	16 GB	20 min
Fileserver	1/2	16 KB	20 GB	30 min
Webproxy	5/1	16 KB	20 GB	30 min
Varmail	1/1	16 KB	20 GB	30 min

5.2 Microbenchmarks

Figure 5 shows the performance of Tinca and Spindle on `Fio` benchmark. As the write percentage declines from 100% (random write) to 0% (random read), the throughputs of Tinca and Spindle increase simultaneously. This is because writes will trigger data replacement to write cold data to disk. When the ratio of writes decreases, less disk writes are performed and the throughputs get promotion. Spindle outperforms Tinca with 1.13×–1.30× throughput under different read/write ratios. Concretely, Spindle achieves 30.0% throughput improvement for random writes and this is mainly due to the reduced cache line flush and memory fence. As for random reads, massive small metadata updates are executed and Spindle can effectively reduce metadata writes through its redesigned metadata updating strategy. The number of clflush per write operation is presented in Fig. 5(b). It can be observed that Spindle reduces up to 32.6% cache line flushes over Tinca. Although read operations only update the file metadata such as *access time*, Tinca needs to update extra cache entry and execute several clflush to change the role of a modified data block, which makes its transaction committing takes longer time than Spindle.



(a) Fio Throughput (b) clflush per write operation

Fig. 5. Fio performance

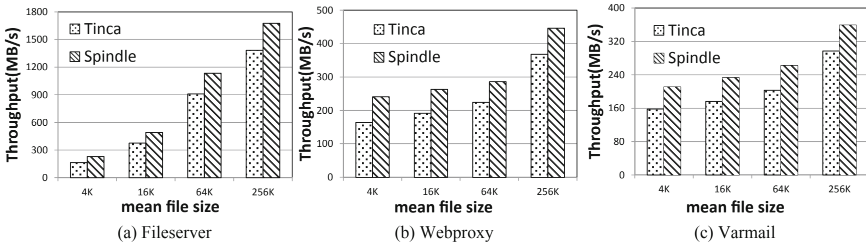
The performance gaps are caused by two major differences between Tinca and Spindle: (1) Tinca writes metadata into NVM in the unit of block while Spindle updates metadata in the unit of inode. (2) Tinca needs to change the role of every modified block and entails extra cache line flush and memory fence.

5.3 Macrobenchmarks

We select fileserver, webproxy, and varmail as the macrobenchmarks to evaluate Spindle's performance. Figure 6 presents the throughputs of Tinca and Spindle under different file sizes.

Fileserver is a write-intensive workload that simulates a file server with operations like file creates, appends, writes, reads and deletes. We observe that for small files, the throughput improvement (40.7%) of Tinca is more striking. This is because metadata write amplification is more obvious for small writes. As the file size grows, the reduction of cache line flush, memory fence and metadata writes is counteracted by the writes of data.

Webproxy is a read-intensive workload that frequently reads the entire file. In such workload, Spindle can get few benefits from the clflush reduction as writes takes up only 16.7% in all read and write operations. However, webproxy entails massive small metadata updates and Spindle could benefit from its redesigned metadata updating strategy thus achieving up to 47% throughput improvement.



(a) Fileserver (b) Webproxy (c) Varmail

Fig. 6. Filebench performance

Varmail simulates the activity of an email server and performs a `fsync` after every file appending write. Both Spindle and Tinca do not need to write data to disk when there are free data and metadata blocks in the NVM cache. However, Tinca needs to write whole metadata blocks into NVM so its throughputs are 20.9%–33.5% lower than that of Spindle.

In conclusion, for read-intensive workloads like webproxy, Spindle benefits from its inode-based metadata updating strategy as it effectively reduces metadata write amplification; for write-intensive workloads like fileserver and varmail, the lightweight transaction scheme of Spindle plays an important role. Since writes to the NVM demands strict ordering, Spindle can effectively reduce the executions of clflush and mfence to promote system performance.

6 Related Work

Using flash-based SSD to develop devices with transactional supports or accelerate HDD-based systems has been explored. Nitro [9] utilizes data compression and deduplication into SSD cache to accelerate primary storage. LightTx [10] tries to reduce the transactional cost while providing better performance. TxFlash [13] utilizes the COW characteristic of NAND flash to provide transaction interface. However, these works focus on the flash-based SSD and do not take the emerging NVM into consideration.

Besides, many works utilize NVM to accelerate disk-based file systems. UBJ [6] is a buffer cache that commits in-place in NVM by freezing data and later checkpoints them to disk. Lee et al. [7] proposes to store modified data in the NVM cache and improve performance with space-efficient management techniques. Tinca [15] is a transactional NVM cache that utilizes the role switch mechanism to reduce write amplification of journaling file systems.

Managing NVM with in-memory file systems has been investigated as well. SCMFS [16] utilizes existing memory management to do the block management and keep space contiguous for each file. PMFS [4] avoids the overheads of block-based storage and enables direct persistent memory access with memory mapped I/O. Lee et al. [8] and Hwang et al. [5] propose tree structures that are optimized for non-volatile memory systems. These works indicate the promising potential of NVM in the future.

7 Conclusion

In this paper, we propose Spindle, a write-optimized NVM cache for journaling file systems. It decouples data and metadata accesses to update them in different granularities. Spindle adopts redesigned metadata log and metadata updating strategy to relieve metadata write amplification. Moreover, it utilizes state switch and COW updating to write data only once in the critical path. Besides, a novel committing protocol is proposed to reduce the transaction ordering overhead. Experiment results confirm that Spindle achieves up to 47% throughput improvement compared with state-of-the-art design.

Acknowledgement. This work is supported by National Key Research & Development Program of China (Grant No. 2018YFB10033002), the National Nature Science Foundation of China (Grant No. 61472241).

References

1. Best, S.: JFS overview (2000)
2. Condit, J., et al.: Better I/O through byte-addressable, persistent memory. In: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, pp. 133–146. ACM (2009)
3. Dadmal, U.D., Vinkare, R.S., Kaushik, P., Mishra, S.: 3D X point technology. Int. J. Electron. Commun. Soft Comput. Sci. Eng. (IJECSCSE) 13–17 (2017)
4. Dulloor, S.R., et al.: System software for persistent memory. In: Proceedings of the Ninth European Conference on Computer Systems, p. 15. ACM (2014)
5. Hwang, D., Kim, W.H., Won, Y., Nam, B.: Endurable transient inconsistency in byte-addressable persistent B+-TREE. In: FAST, pp. 187–200 (2018)
6. Lee, E., Bahn, H., Noh, S.H.: Unioning of the buffer cache and journaling layers with non-volatile memory. In: FAST, pp. 73–80 (2013)
7. Lee, E., Kang, H., Bahn, H., Shin, K.G.: Eliminating periodic flush overhead of file I/O with non-volatile buffer cache. IEEE Trans. Comput. **65**(4), 1145–1157 (2016)
8. Lee, S.K., Lim, K.H., Song, H., Nam, B., Noh, S.H.: WORT: write optimal radix tree for persistent memory storage systems. In: FAST, pp. 257–270 (2017)
9. Li, C., Shilane, P., Douglis, F., Shim, H., Smaldone, S., Wallace, G.: Nitro: a capacity-optimized SSD cache for primary storage. In: ATC, pp. 501–512 (2014)
10. Lu, Y., Shu, J., Guo, J., Li, S., Mutlu, O.: LightTX: a lightweight transactional design in flash-based SSDs to support flexible transactions. In: 2013 IEEE 31st International Conference on Computer Design (ICCD), pp. 115–122. IEEE (2013)
11. Mathur, A., Cao, M., Bhattacharya, S., Dilger, A., Tomas, A., Vivier, L.: The new ext4 filesystem: current status and future plans. In: Proceedings of the Linux symposium, vol. 2, pp. 21–33 (2007)
12. Pelley, S., Chen, P.M., Wenisch, T.F.: Memory persistency. In: ACM SIGARCH Computer Architecture News, vol. 42, pp. 265–276. IEEE Press (2014)
13. Prabhakaran, V., Rodeheffer, T.L., Zhou, L.: Transactional flash. In: OSDI, vol. 8 (2008)
14. Sun, Z., et al.: Multi retention level STT-RAM cache designs with a dynamic refresh scheme. In: Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 329–338. ACM (2011)
15. Wei, Q., Wang, C., Chen, C., Yang, Y., Yang, J., Xue, M.: Transactional NVM cache with high performance and crash consistency. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, p. 56. ACM (2017)
16. Wu, X., Reddy, A.: Scmfs: a file system for storage class memory. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, p. 39. ACM (2011)
17. Xu, C., Niu, D., Muralimanohar, N., Jouppi, N.P., Xie, Y.: Understanding the trade-offs in multi-level cell reram memory design. In: 2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC), pp. 1–6. IEEE (2013)
18. Xu, J., Swanson, S.: NOVA: a log-structured file system for hybrid volatile/non-volatile main memories. In: FAST, pp. 323–338 (2016)

19. Yoon, D.H., Chang, J., Schreiber, R.S., Jouppi, N.P.: Practical nonvolatile multilevel-cell phase change memory. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, p. 21. ACM (2013)
20. Zheng, S., Huang, L., Liu, H., Wu, L., Zha, J.: HMVFS: a hybrid memory versioning file system. In: 2016 32nd Symposium on Mass Storage Systems and Technologies (MSST), pp. 1–14. IEEE (2016)