Redesigning Data and Metadata Updates in PM File Systems with Persistent CPU Caches

Congyong Chen[1], Shengan Zheng[2](✉), Yuhang Zhang[1], and Linpeng Huang[1](✉)

[1] Shanghai Jiao Tong University, Shanghai, China
{ndsffx304ccy,carlislefelix,lphuang}@sjtu.edu.cn

[2] MoE Key Lab of Artificial Intelligence, AI Institute, Shanghai Jiao Tong University, Shanghai, China
shengan@sjtu.edu.cn

Abstract. The recent advent of persistent memory has revolutionized file system design by enabling efficient, fast, and durable data updates. However, to cope with mismatched access granularities and volatile CPU caches, existing file systems resort to costly data and metadata update approaches, leading to high write-back latency and excessive PM bandwidth consumption. Recent cache persistence techniques, such as Intel's eADR, address this issue by automatically flushing data from CPU caches to PM during power failures, making CPU caches effectively persistent. Persistent CPU caches provide both cacheline access granularity and data durability, acting as a new storage medium that can greatly reduce the latency and throughput overhead of data persistence. We present FusionFS, a file system that leverages persistent CPU caches to redesign data and metadata update approaches. FusionFS employs an *adaptive data update* approach that chooses the most effective mechanism based on file access patterns, minimizing PM bandwidth consumption and update latency. FusionFS also adopts an *aggregated metadata update* approach that consolidates small entries in persistent log buffers before appending them to PM logs, minimizing small random writes to PM. Experimental results show that FusionFS outperforms existing PM file systems in terms of latency and throughput in various scenarios.

Keywords: Persistent memory · File system · CPU cache.

1 Introduction

Emerging byte-addressable persistent memory (PM) offers DRAM-like performance and disk-like durability, enabling in-memory system designs with high throughput and low persistence overhead. However, commercially available PM products, such as Intel Optane DC Persistent Memory, do not meet the expectations of system designers in terms of data flushing and access granularity[28, 7, 19]. Applications must issue flush instructions and memory barriers on platforms with volatile CPU caches (e.g., ADR-based platforms) to guarantee data persistence. This results in prolonged critical path latency and high consumption of PM write bandwidth. Moreover, PM's internal XPLine access granularity (256B) mismatches with CPU caches' cacheline access granularity (64B), causing small random PM writes to trigger additional read-modify-write traffic. Therefore, accessing PM with blocks aligned to 256B can maximize the utilization of PM bandwidth.

These challenges force existing systems to adopt expensive data and metadata update approaches. Existing data update approaches are mainly based on three types of mechanisms: i) *active-flush update* eagerly persists data synchronously with flush instructions and memory barriers[10, 20, 26], ii) *non-temporal update* directly writes data to PM bypassing CPU caches[27, 18, 8], iii) *asynchronous update* buffers data in DRAM and persists updates asynchronously[20, 9, 30, 25, 14]. The first two lead to high synchronous data persistence overhead and write amplification, while the third cannot guarantee immediate data consistency. Performance degradation occurs in access modes less conducive to these approaches. Similarly, metadata update approaches fall into two main categories: i) *in-place approach* involves direct modifications to persistent indexes, making it susceptible to write amplification[10, 9, 29, 25, 16, 22], ii) *log-structured approach* places append-only logs in PM and maintains volatile indexes in DRAM. However, the small update granularity in both methods introduces write amplification.

Fortunately, recent cache persistence techniques (e.g., Intel's eADR[17], battery-backed cache[1], CXL's Global Persistent Flush[6]) enable automatic data flushing from CPU caches to PM during power failures. This feature provides persistence to byte-addressable CPU caches, enabling them to operate as a novel storage medium that can absorb writes to the hot PM region. By leveraging persistent CPU caches, data update mechanisms that are usually meant for volatile data structures can be applied to update persistent data. These mechanisms include: i) *pinned-cache update* pins data to a designated cache space within persistent CPU caches[31] using cache allocation technologies[15, 2], ii) *in-place update* does not explicitly issue flush instructions after writes[29, 21]. In addition, hardware transactional memory (HTM) can now be used to transactionally update multiple cachelines in PM. Previously, the use of HTM was limited by the need to flush data from volatile CPU caches for persistence, causing HTM transactions to abort.

However, existing PM systems, including those based on eADR, fail to fully harness the potential of persistent CPU caches.For data updates, ADR-based systems use *active-flush/non-temporal update* to ensure consistency. On eADR-enabled platforms, they completely switch to *in-place update* to avoid persistence overhead[29, 21]. However, such a fixed approach can lead to write amplification as writes larger than a cacheline may be split into multiple cacheline-sized random writes due to the cacheline eviction policy (e.g., LRU algorithm)[13]. For metadata updates, *in-place approach* often writes with atomic instructions, whose granularity mismatches with PM's internal access granularity, resulting in write amplification. Even when using *log-structured approach* to write sequentially and taking advantage of PM's internal buffer[27, 3], massive metadata (e.g., NOVA's per-inode logs, KucoFS's per-partition-tree logs) can overwhelm the small buffers (about 16KB per Optane DIMM)[28].

In this paper, we present FusionFS, a file system that harnesses persistent CPU caches to optimize data and metadata operations. FusionFS uses an *adaptive data update* approach to select the most suitable data update mechanism based on file access patterns (e.g., data hotness, update size, and consistency requirements). FusionFS also includes a hotspot detector that considers access counts and last access timestamps, supporting both system calls and memory mapping accesses. Moreover, FusionFS proposes an *aggregated metadata update* approach to consolidate small metadata log entries within log buffers located in persistent CPU caches. In summary, the contributions of this paper include:

- We perform an in-depth analysis of the impact of persistent CPU caches on PM, highlighting their potential in optimizing the data and metadata update approaches of existing PM systems.
- We propose FusionFS, a PM file system that integrates both *adaptive data update* and *aggregated metadata update* to leverage the capabilities of persistent CPU caches, thereby reducing latency and minimizing PM bandwidth consumption.
- We implement FusionFS as a POSIX-compliant file system for Linux. Performance results show that FusionFS achieves significantly lower latency and higher throughput than state-of-the-art PM file systems.

2 Design and Implementation

In this section, we introduce FusionFS, a novel PM file system that exploits persistent CPU caches to redesign data and metadata update approaches. We first propose an *adaptive data update* approach to select the optimal data update mechanism from the five update mechanisms based on access patterns (Section 2.1). We then design an *aggregated metadata update* approach to minimize small metadata writes to PM (Section 2.2).

2.1 Adaptive Data Update

Update Policy. FusionFS employs an adaptive approach that dynamically selects the most suitable data update mechanism during system calls and memory mapping accesses, as shown in Fig. 1. FusionFS uses radix trees as file indexes for efficient file management. In addition, FusionFS uses HTM transactions to replace inode locks, ensuring consistency of in-place writes and allowing simultaneous writes to different sections of the same file.

For large ($\geq 256B$) hot data, FusionFS employs *in-place update* to buffer writes in persistent CPU caches. This is especially effective for hot data that is likely to be accessed again before eviction, reducing writes to PM and leveraging the benefits of cache hits. For hot data that is frequently updated in small granularity ($< 256B$), FusionFS uses *pinned-cache update.* It protects these small data

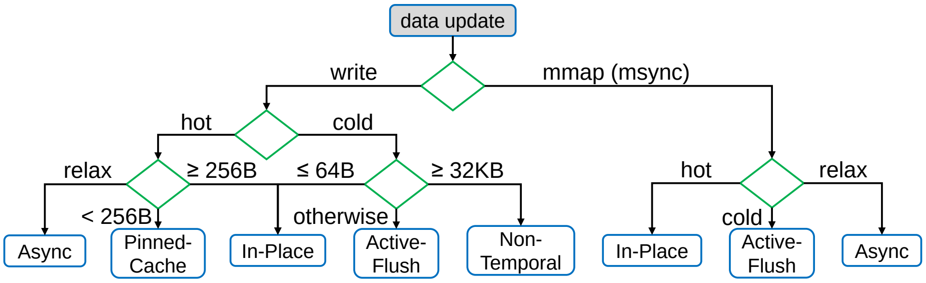

Fig. 1. Adaptive Data Update

segments from cacheline eviction, ensuring their persistence in CPU caches while minimizing CPU cache usage. For hot data with relaxed consistency, including recoverable data and data flagged as relaxed consistent by the user, FusionFS chooses *asynchronous update* to buffer writes in DRAM and persist them asynchronously. This is because it is free of write amplification on cacheline eviction and allows the high performance of DRAM to be exploited.

For small ($\leq$ 64B) cold data, FusionFS uses *in-place update* to minimize the cost of explicit flush instructions, since actively flushing a single cacheline will not mitigate write amplification caused by cacheline eviction. Conversely, for large ($\geq$ 32KB) cold data, FusionFS employs *non-temporal update* due to its lower latency and higher bandwidth. FusionFS ensures compatibility by writing shadow pages outside of HTM transactions and achieves consistency by modifying indexes within HTM transactions. Recognizing that the overhead of index modification and copying unaligned parts is non-trivial for smaller updates, FusionFS switches to *active-flush update* for medium-sized cold data. FusionFS ensures compatibility by delaying flushes until HTM transactions are complete and achieves consistency by performing in-place writes to data pages within HTM transactions.

FusionFS extends support for the *adaptive data update* approach to memory-mapped files. For hot data, FusionFS uses *in-place update* by omitting flush instructions in synchronization calls (e.g., *fsync*, *fdatasync*, and *msync*) to reduce writes to PM. For cold data, FusionFS uses *active-flush update* to aggregate the sequential writes in XPBuffer. For hot data with relaxed consistency, FusionFS uses *asynchronous update* by buffering data pages in DRAM and persisting them to PM during synchronization calls to take advantage of the high performance of DRAM.

By dynamically selecting the most effective mechanism based on access patterns, FusionFS optimizes data updates during system calls and memory mapping accesses. The adaptive approach of FusionFS ensures better PM bandwidth utilization and lower latency across various scenarios, addressing the limitations observed in conventional approaches.

Hotspot Detector. We design a lightweight yet effective hotspot detector to measure the hotness of data pages during system calls and memory mapping accesses. For system call accesses, the detector relies on the number of accesses to each data page within a designated period. After profiling the access counts for all data pages during this period, FusionFS classifies pages that exceed a certain threshold as hot data. The size of the identified hot data set matches the specified CPU cache size.

For memory mapping accesses, obtaining access count information directly is challenging. To address this, FusionFS proposes an approach inspired by the kernel's automatic NUMA balancing. FusionFS periodically adjusts the permissions of data pages in the background. This allows the next access to be captured and recorded by the page fault handler function. By recording the first access time of all pages during a scan period, FusionFS categorizes pages whose difference between access time and scan time is less than a certain threshold as hot data.

By analyzing both the number of accesses to data pages and the timestamp of the last access, FusionFS effectively identifies hot data in both system call and memory mapping scenarios. The lightweight design of this detector avoids the use of locks, minimizing update overhead.

2.2 Aggregated Metadata Update

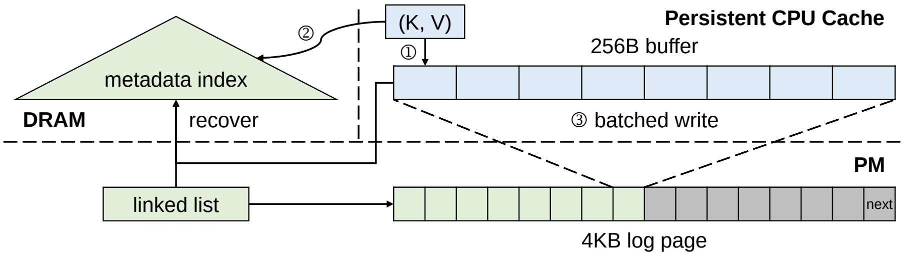

Fig. 2. Aggregated Metadata Update

FusionFS introduces the *aggregated metadata update* approach to increase the size of metadata writes to PM, as shown in Fig. 2. FusionFS maintains individual logs for each directory. During directory system calls (e.g., *create*, *mkdir*, *unlink*, and *rmdir*) that modify directory hash tables, FusionFS inserts corresponding log entries into the directory inode's log. In contrast to *log-structured approach*, FusionFS caches small metadata log entries ($< 256B$) in buffers pinned to persistent CPU caches (Step ①). FusionFS also maintains metadata indexes in DRAM to speed up lookup operations (Step ②). These entries are batched and copied to PM logs with *active-flush update* when the accumulated size exceeds 256B (Step ③). While FusionFS writes to CPU caches, DRAM, and PM during metadata updates, the benefit of writing in XPLine granularity to PM outweighs the small latency caused by redundant writes to the high-speed CPU cache and DRAM. PM logs in FusionFS are designed to be extendable and organized as linked lists of 4KB pages. To ensure crash consistency, FusionFS records the used space of each buffer and log page, and the pointer to the next log page. In the event of system failures, these indexes can be reconstructed by replaying log entries from both PM logs and buffers. This aggregated approach significantly reduces the frequency of small random writes to PM during metadata updates, while ensuring immediate consistency of log entries.

3 Evaluation

In this section, we evaluate the performance of FusionFS and answer the following questions:

- Can FusionFS leverage the *adaptive data update* approach to optimize file updates?
- Can FusionFS utilize the *aggregated metadata update* approach to improve metadata updates?
- Can FusionFS exhibit optimal performance in various application scenarios?

3.1 Experimental Platform

We evaluate FusionFS on a server equipped with two 28-core Intel Xeon Gold 6348 (42M cache), with four 128GB Intel Optane and four 16GB DDR4 DRAM on each node. All experiments are performed on one NUMA node to avoid NUMA effects. To evaluate the performance of FusionFS, we compare it to PMFS, NOVA, SoupFS, and EXT4-DAX. For micro-benchmarks, we use Fio[11] to generate various access patterns to test the effectiveness of FusionFS's design. For macro-benchmarks, we use several workloads of Filebench[24]. For application benchmarks, we use YCSB[4] on LevelDB[12] and TPC-C[5] on SQLite[23] to evaluate the performance of FusionFS.

3.2 Micro-benchmarks

We use Fio to demonstrate how FusionFS benefits from the *adaptive data update* approach in file writes. We vary the Zipf parameter θ to issue uniform or skewed ($\theta = 0.99$) random write requests. We use 20 threads for the experiments to saturate the PM bandwidth, each thread performs 4KB I/Os to a private file of 64MB, and all writes are synchronous.

Fig. 3(a) shows that FusionFS achieves good performance in both scenarios by using the optimal data update mechanism for the access pattern during system calls. Fig. 3(b) shows similar results using memory mapping. PMFS and NOVA exhibit near-zero throughput due to their suboptimal synchronization performance. In contrast, FusionFS maintains high performance by selectively excluding hot data flushes from `fdatasync()` calls to minimize PM writes, while retaining them for cold data to prevent write amplification caused by random cacheline evictions.

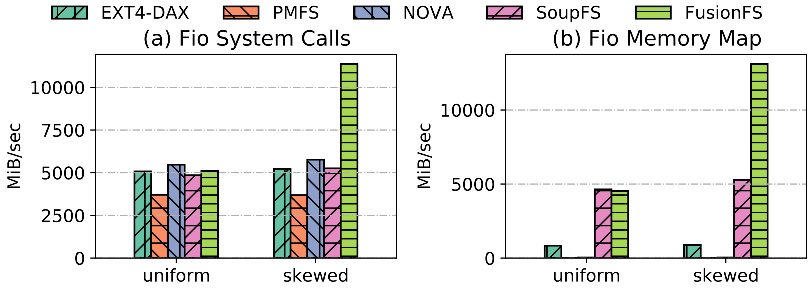

Fig. 3. Results of Fio

3.3 Macro-benchmarks

We evaluate the performance of FusionFS through Filebench, using three workloads to evaluate the performance of data and metadata updates. The results are shown in Fig. 4.

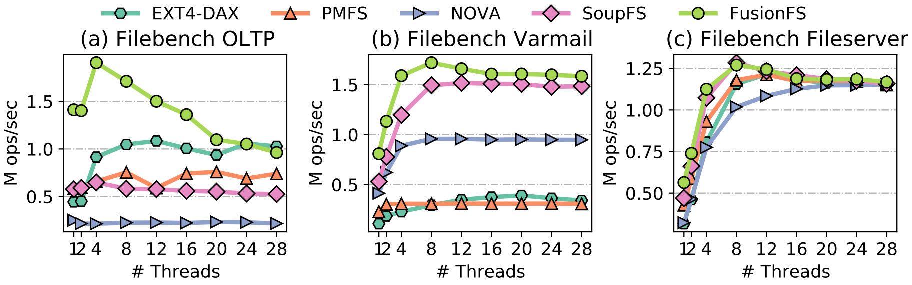

Fig. 4. Results of Filebench Workloads

OLTP[3] emulates database transactions at the file system level by issuing 2KB asynchronous writes to a small data file and periodic 256KB synchronous writes to a small log file. Given the high hotness of the files, FusionFS uses *in-place update*, allowing writes to hit CPU caches with high probability and outperforming other file systems by up to 8.9×.

Varmail emphasizes metadata updates by creating and deleting many small files in a single directory. In this workload, SoupFS, which is optimized for metadata updates, outperforms EXT4-DAX, PMFS, and NOVA. However, SoupFS

[3] We increase the frequency of OLTP-initiated transactions to better match the high performance of PM.

executes data persistence in background threads, while Varmail frequently calls `fsync()`, incurring synchronized metadata persistence overhead that affects its throughput. FusionFS, in contrast, employs the *aggregated metadata update* approach, guaranteeing immediate consistency of metadata updates. This approach leads to superior performance, outperforming SoupFS by up to 1.5× and other PM file systems by up to 7.5×.

Fileserver focuses on both file and metadata updates by keeping the number of files in each directory small and setting small file sizes. FusionFS and SoupFS show slightly better performance in this workload because they do not have to persist small writes immediately. While ADR-based SoupFS does not provide immediate consistency after writes, FusionFS outperforms other PM file systems by up to 1.2× and offers protection against data loss during power failures thanks to persistent CPU caches enabled by eADR.

3.4 Application Benchmarks

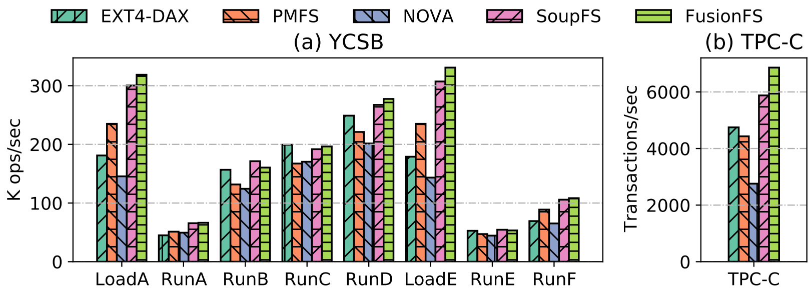

Fig. 5. Results of Application Benchmarks

Fig. 5 summarizes real-world application performance across different file systems. We run YCSB workloads on LevelDB by issuing five million requests with a single thread, where each request contains a 16B key and a 64B value. We then measure the performance of TPC-C on SQLite in the Write-Ahead-Logging (WAL) mode. Overall, FusionFS achieves up to 2.5× the performance of other PM file systems in write-intensive workloads, including LoadA, LoadE, and TPC-C, while consistently maintaining optimal performance across all scenarios. This is realized by adaptively choosing the most appropriate data update mechanism for the access pattern.

4 Conclusions

Recent cache persistence techniques allow CPU caches to be considered persistent. In this paper, we propose FusionFS, a PM file system that leverages persistent CPU caches to redesign data and metadata update approaches. FusionFS employs an *adaptive data update* approach that chooses the most effective mechanism based on file access patterns. FusionFS also adopts an *aggregated metadata update* approach that consolidates small entries in persistent log buffers before appending them to PM logs. These novel approaches help reduce PM bandwidth consumption and update latency. Performance results show that FusionFS outperforms existing PM file systems in terms of latency and throughput in various scenarios.

Acknowledgements. This work is supported by National Key Research and Development Program of China (Grant No. 2022YFB4500303), National Natural Science Foundation of China (NSFC) (Grant No. 62227809), the Fundamental Research Funds for the Central Universities, Shanghai Municipal Science and Technology Major Project (Grant No. 2021SHZDZX0102), Natural Science Foundation of Shanghai (Grant No. 22ZR1435400), and Huawei Innovation Research Plan.

References

1. Alshboul, M., Ramrakhyani, P., Wang, W., Tuck, J., Solihin, Y.: Bbb: Simplifying persistent programming using battery-backed buffers. In: 2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA). pp. 111–124. IEEE (2021)
2. AMD: Amd64 technology platform quality of service extensions (2022), https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/other/56375_1_03_PUB.pdf
3. Chen, Y., Lu, Y., Zhu, B., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H., Shu, J.: Scalable persistent memory file system with {Kernel-Userspace} collaboration. In: 19th USENIX Conference on File and Storage Technologies (FAST 21). pp. 81–95 (2021)
4. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with ycsb. In: Proceedings of the 1st ACM symposium on Cloud computing. pp. 143–154 (2010)
5. Council, T.P.P.: Tpc benchmark c, standard specification version 5 (2001)
6. Compute express link™: The breakthrough cpu-to-device interconnect, https://www.computeexpresslink.org/
7. Daase, B., Bollmeier, L.J., Benson, L., Rabl, T.: Maximizing persistent memory bandwidth utilization for olap workloads. In: Proceedings of the 2021 International Conference on Management of Data. pp. 339–351 (2021)
8. Dong, M., Bu, H., Yi, J., Dong, B., Chen, H.: Performance and protection in the zofs user-space nvm file system. In: Proceedings of the 27th ACM Symposium on Operating Systems Principles. pp. 478–493 (2019)
9. Dong, M., Chen, H.: Soft updates made simple and fast on non-volatile memory. In: 2017 uSENIX Annual Technical Conference (uSENIX ATC 17). pp. 719–731 (2017)
10. Dulloor, S.R., Kumar, S., Keshavamurthy, A., Lantz, P., Reddy, D., Sankaran, R., Jackson, J.: System software for persistent memory. In: Proceedings of the Ninth European Conference on Computer Systems. pp. 1–15 (2014)

11. Fio (flexible i/o tester), https://github.com/axboe/fio
12. Google: Leveldb, https://github.com/google/leveldb
13. Gugnani, S., Kashyap, A., Lu, X.: Understanding the idiosyncrasies of real persistent memory. Proceedings of the VLDB Endowment **14**(4), 626–639 (2020)
14. He, K., An, Y., Luo, Y., Liu, X., Wang, G.: Flatlsm: Write-optimized lsm-tree for pm-based kv stores. ACM Transactions on Storage **19**(2), 1–26 (2023)
15. Intel: Improving real-time performance by utilizing cache allocation technology (2015), https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/cache-allocation-technology-white-paper.pdf
16. Intel: Persistent memory development kit (2020), https://pmem.io/
17. Intel: eadr: New opportunities for persistent memory applications (2021), https://www.intel.com/content/www/us/en/developer/articles/technical/eadr-new-opportunities-for-persistent-memory-applications.html
18. Kadekodi, R., Lee, S.K., Kashyap, S., Kim, T., Kolli, A., Chidambaram, V.: Splitfs: Reducing software overhead in file systems for persistent memory. In: Proceedings of the 27th ACM Symposium on Operating Systems Principles. pp. 494–508 (2019)
19. Kim, W.H., Krishnan, R.M., Fu, X., Kashyap, S., Min, C.: Pactree: A high performance persistent range index using pac guidelines. In: Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles. pp. 424–439 (2021)
20. Ou, J., Shu, J., Lu, Y.: A high performance file system for non-volatile main memory. In: Proceedings of the Eleventh European Conference on Computer Systems. pp. 1–16 (2016)
21. PMDK: libpmem, https://pmem.io/pmdk/libpmem/
22. Schwalb, D., Berning, T., Faust, M., Dreseler, M., Plattner, H.: nvm malloc: Memory allocation for nvram. Adms@ Vldb **15**, 61–72 (2015)
23. SQLite: Sqlite transactional sql database engine, https://www.sqlite.org/
24. Vasily Tarasov, E.Z., Shepler, S.: Filebench: A flexible framework for file system benchmarking. USENIX; login **41**(1), 6–12 (2016)
25. Woo, H., Han, D., Ha, S., Noh, S.H., Nam, B.: On stacking a persistent memory file system on legacy file systems. In: 21st USENIX Conference on File and Storage Technologies (FAST 23). pp. 281–296 (2023)
26. Wu, X., Reddy, A.N.: Scmfs: A file system for storage class memory. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 1–11 (2011)
27. Xu, J., Swanson, S.: {NOVA}: A log-structured file system for hybrid {Volatile/Non-volatile} main memories. In: 14th USENIX Conference on File and Storage Technologies (FAST 16). pp. 323–338 (2016)
28. Yang, J., Kim, J., Hoseinzadeh, M., Izraelevitz, J., Swanson, S.: An empirical guide to the behavior and use of scalable persistent memory. In: 18th USENIX Conference on File and Storage Technologies (FAST 20). pp. 169–182 (2020)
29. Yi, J., Dong, M., Wu, F., Chen, H.: {HTMFS}: Strong consistency comes for free with hardware transactional memory in persistent memory file systems. In: 20th USENIX Conference on File and Storage Technologies (FAST 22). pp. 17–34 (2022)
30. Zheng, S., Hoseinzadeh, M., Swanson, S.: Ziggurat: A tiered file system for {Non-Volatile} main memories and disks. In: 17th USENIX Conference on File and Storage Technologies (FAST 19). pp. 207–219 (2019)
31. Zhong, Y., Shen, Z., Yu, Z., Shu, J.: Redesigning high-performance lsm-based key-value stores with persistent cpu caches. In: 2023 IEEE 39th International Conference on Data Engineering (ICDE). pp. 1098–1111. IEEE (2023)