FusionFS: A Contention-Resilient File System for Persistent CPU Caches

CONGYONG CHEN, Shanghai Jiao Tong University, Shanghai, China
SHENGAN ZHENG, MoE Key Lab of Artificial Intelligence, AI Institute, Shanghai Jiao Tong University, Shanghai, China
YUHANG ZHANG, Shanghai Jiao Tong University, Shanghai, China
LINPENG HUANG, Shanghai Jiao Tong University, Shanghai, China

Byte-addressable storage (BAS), such as persistent memory and CXL-SSDs, does not meet system designers' expectations for data flushing and access granularity. Persistent CPU caches, enabled by recent techniques like Intel's eADR and CXL's Global Persistent Flush, can mitigate these issues without sacrificing consistency. However, the shared nature of CPU caches can lead to cache contention, which can result in cached data being frequently evicted to the BAS and reloaded into caches, negating the benefits of caching. If the BAS write granularity is larger than the cacheline eviction granularity, this can also lead to severe write amplification.

In this paper, we identify, characterize, and propose solutions to the problem of contention in persistent CPU caches, which is largely overlooked by existing systems. These systems either simply assume that cached data is hot enough to survive cache evictions or use unsupported cache allocation techniques without testing their effectiveness. We also present FusionFS, a contention-resilient kernel file system that uses persistent CPU caches to redesign data update approaches. FusionFS employs an *adaptive data update* approach that chooses the most effective mechanism based on file access patterns during system calls and memory mapping accesses, minimizing BAS media writes and improving throughput. FusionFS also employs *contention-aware cache allocation* to minimize various types of cache contention. Experimental results show that FusionFS outperforms existing file systems and effectively mitigates various types of cache contention.

CCS Concepts: • **Hardware** → **Non-volatile memory**; **Memory and dense storage**; • **Software and its engineering** → **File systems management**.

Additional Key Words and Phrases: Persistent memory, non-volatile memory, file system, CPU cache

1 Introduction

Byte-addressable storage (BAS), such as Intel Optane PM [28] and Compute Express Link (CXL)-SSDs [49, 59], combines the byte addressability of DRAM with the durability of disk storage, enabling system designs with high throughput and low persistence overhead. However, commercially available BAS products do not meet the

Extension of Conference Paper [9]. In this new manuscript, (1) we identify, characterize, and propose solutions to several types of cache contention that are largely overlooked by existing systems and can offset the benefits of persistent CPU caches. (2) Following our proposed guidelines, we introduce *contention-aware cache allocation* to mitigate various types of cache contention in the kernel space. (3) We extend *adaptive data update* with L3_CAT-based *dedicated-cache update* and a hotspot detector that automatically limits hot data within the dedicated cache capacity. (4) We add experiments to demonstrate that FusionFS is resilient to cache contention. (5) We discuss the generality of our work and compare it to related work.

Authors' Contact Information: Congyong Chen, Shanghai Jiao Tong University, Shanghai, China; e-mail: ndsffx304ccy@sjtu.edu.cn; Shengan Zheng, MoE Key Lab of Artificial Intelligence, AI Institute, Shanghai Jiao Tong University, Shanghai, China; e-mail: shengan@sjtu.edu.cn; Yuhang Zhang, Shanghai Jiao Tong University, Shanghai, China; e-mail: carlislefelix@sjtu.edu.cn; Linpeng Huang, Shanghai Jiao Tong University, Shanghai, China; e-mail: lphuang@sjtu.edu.cn.

ACM 1544-3973/2025/2-ART
https://doi.org/10.1145/3719656

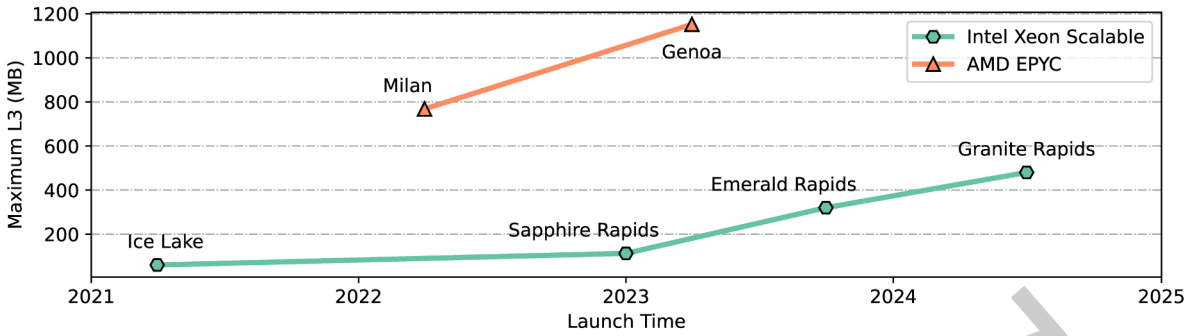

Fig. 1. Maximum L3 cache sizes in recent processor generations.

expectations of system designers in terms of data flushing and access granularity [11, 38, 58]. 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 limited BAS write bandwidth. Moreover, BAS often has a large access granularity (e.g., 256B for Optane PM [58] and 16KB for CXL-SSDs [59, 65]) that mismatches with CPU caches' cacheline access granularity (64B), causing small random writes to trigger additional read-modify-write traffic.

Existing systems often adopt expensive data update approaches to overcome these limitations. These approaches are mainly based on three types of mechanisms: 1) *active-flush update* eagerly persists data synchronously with flush instructions and memory barriers [15, 45, 56, 57], 2) *non-temporal update* writes data to BAS directly bypassing CPU caches [13, 35], 3) *asynchronous update* buffers data in DRAM and persists updates asynchronously [14, 19, 45, 67, 68]. Synchronous updates, including *active-flush update* and *non-temporal update*, lead to high data persistence overhead on the critical path and write amplification for accesses with mismatched granularity, while *asynchronous update* cannot guarantee immediate data consistency.

Fortunately, recent cache persistence techniques (e.g., Intel's eADR [27], battery-backed cache [1], CXL's Global Persistent Flush [10]) enable automatic data flushing from CPU caches to BAS during power failures. In addition, CPU L3 caches are large and have grown rapidly in recent years, as shown in Fig. 1. By leveraging persistent CPU caches, data update mechanisms originally designed for volatile data structures can be applied to update persistent data. These mechanisms include: 1) *dedicated-cache update* allocates a dedicated cache space within persistent CPU caches for BAS systems [70], 2) *in-place update* does not explicitly issue flush instructions after writes [33, 46, 60, 63].

Despite these advances, CPU caches are shared by BAS, DRAM, and I/O devices [26] and have limited cache ways, leading to **cache contention** that can offset or even reverse the benefits of persistent CPU caches. Cache contention can be divided into two main categories: *internal contention* and *external contention*. *Internal contention* occurs even when no other workloads are running and includes: 1) *BAS-BAS contention*, which occurs when the BAS working set size (WSS) exceeds the cache capacity [33, 63, 70], 2) *DRAM-BAS contention*, which occurs when the DRAM accesses of the BAS systems evict cached BAS data, 3) *set contention*, which occurs in set-associative caches when multiple memory accesses compete for the same cache set, resulting in cache evictions even when the cache isn't full. *External contention* occurs when other workloads are running alongside the BAS system and includes: 4) *interfering process contention*, which refers to the competition for cache space between BAS systems and other unrelated interfering processes, and 5) *I/O contention*, which occurs when I/O devices compete with BAS systems for cache space.

Table 1. Effectiveness of BAS Systems' Cache Contention Mitigation Techniques

	Kernel Space Support	Internal			External	
		BAS-BAS	DRAM-BAS	Set	Interfering Process	I/O
Remove All Flush [46, 60]	✓	✗	✗	✗	✗	✗
Flush Cold Data [33, 63]	✓	✓	✗	✗	✗	✗
Cache Pseudo-Locking [24, 70]	✗	✗	✗	✗	✗	✗
Naive L3_CAT [31]	✗	✗	✗	✗	✓	✗
FusionFS	✓	✓	✓	✓	✓	✓

However, existing BAS systems, including those based on cache persistence techniques, fail to fully harness the potential of persistent CPU caches. For data updates, systems with volatile caches use *active-flush/non-temporal update* to ensure consistency. On platforms with persistent caches, file systems completely switch to *in-place update* to avoid persistence overhead [60]. Such a fixed approach can lead to *BAS-BAS contention* due to the large BAS WSS of the system. Moreover, existing systems are ineffective at mitigating contention types other than *BAS-BAS contention*, as shown in Table 1. This results in cached data being frequently evicted to BAS and reloaded into caches, negating the benefits of caching. Worse yet, if the write granularity of BAS is larger than the cacheline eviction granularity, it can lead to severe 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) [18].

In this paper, we identify, characterize, and propose solutions to several types of cache contention that remain challenging for existing systems. We then propose FusionFS, a contention-resilient kernel file system that harnesses persistent CPU caches to optimize data updates. The key observation is that *different data update mechanisms have their own applicable file access patterns*. Therefore, FusionFS uses an *adaptive data update* approach to dynamically select the most appropriate data update mechanism based on file access patterns (e.g., data hotness, update size, and consistency requirements). Writes to large ($\geq$ 4KB) hot data and small ($\leq$ 64B) cold data are stored in persistent CPU caches to improve throughput and reduce BAS bandwidth consumption. For memory mapping accesses, FusionFS omits flush instructions for hot data during synchronization calls (e.g., *msync*) to minimize writes to BAS. For hot data with relaxed consistency, FusionFS buffers writes to DRAM and persists them during synchronization calls. Remaining writes go directly to BAS so that neighboring cacheline evictions can be aggregated in its internal buffer. FusionFS uses a *scalable 2Q-LRU* approach to transparently measure data hotness. For memory mapping accesses that bypass the file system to access BAS directly, FusionFS adopts a *page fault-based profiling* approach where data page permissions are periodically adjusted to catch the next access and add it to the appropriate queue.

FusionFS uses a *contention-aware cache allocation* approach to mitigate various types of cache contention. FusionFS offloads *dedicated-cache updates* to L3_CAT-protected kernel threads to mitigate *DRAM-BAS contention* and support L3_CAT in kernel space. To mitigate *set contention*, FusionFS leaves some headroom in dedicated caches. In addition, FusionFS spreads file data across kernel threads of different NUMA nodes to utilize CPU caches from multiple nodes, and preferentially allocates caches not used by DCA to avoid *I/O contention*.

In summary, the contributions of this paper include:

- We perform an in-depth analysis of the impact of persistent CPU caches on BAS, highlighting their potential in optimizing the data update approaches of existing systems.
- We identify, characterize, and propose solutions to several types of cache contention that are largely overlooked by existing systems and can offset the benefits of persistent CPU caches.
- We propose FusionFS, a kernel file system that integrates *adaptive data update* and *contention-aware cache allocation* to improve throughput and reduce BAS bandwidth consumption.

- We implement FusionFS as a POSIX-compliant kernel file system for Linux. Performance results show that FusionFS outperforms existing file systems and effectively mitigates various types of cache contention. The source code of FusionFS is publicly available at https://github.com/SJTU-DDST/FusionFS.

2 Background and Motivation

2.1 Impact of Persistent CPU Caches on BAS

Byte-addressable storage (BAS) offers many attractive features that are changing the design of storage systems, including file systems, databases and key-value stores. It can be accessed by the CPU in a DRAM-like byte-addressable manner, but has lower bandwidth and higher latency. It also offers disk-like endurance and large capacity. There are several types of BAS, such as Intel's Optane PM [28] and CXL-SSDs [49, 59]. Apart from these fundamental features, we highlight two other observations on platforms with persistent CPU caches that prevent existing systems from fully leveraging BAS performance.

Observation 1: Access pattern determines how flush operations affect performance. On platforms with volatile CPU caches (e.g., ADR-based platforms [21]), it is necessary to proactively flush data into BAS or bypass CPU caches (e.g., `clwb` and `ntstore`) and use memory barriers to guarantee data persistence. Recent cache persistence techniques allow automatic flushing of data from CPU caches to the BAS during power failures. By omitting flush instructions, writes to the hot BAS region can be absorbed by CPU caches, effectively reducing BAS bandwidth consumption without compromising data consistency.

However, simply removing the flush instructions does not fully exploit the potential of persistent CPU caches because the BAS WSS can exceed the cache capacity, so cached data is often evicted to the BAS and reloaded into caches. In addition, CPU caches are shared by BAS, DRAM and I/O devices [26], further exacerbating cache contention. In contrast, explicitly issuing flush instructions for cold data not only allows BAS to aggregate large sequential writes to the cold region in its internal buffer [58], but also reduces BAS WSS.

Consequently, with the presence of persistent CPU caches, applications gain the flexibility to decide whether to use flush instructions based on access patterns.

Observation 2: Data access granularity affects BAS bandwidth consumption. The BAS often has a large access granularity (e.g., 256B for Optane PM [58] and 16KB for CXL-SSDs [59, 65]) that does not match the access granularity of CPU caches (64B). We refer to these access granularity blocks as XPLine. This mismatch makes small random writes to the BAS inefficient because they are converted to accesses with access granularity, resulting in write amplification [58]. Therefore, accessing the BAS with blocks matched to its access granularity can maximize the use of BAS bandwidth. In addition, persistent CPU caches allow multiple small writes to be aggregated before being written to BAS, effectively mitigating write amplification.

As a result, the two observations that hinder systems from fully harnessing BAS performance can be mitigated through the use of persistent CPU caches.

2.2 Data Update Approaches

With the introduction of persistent CPU caches, mechanisms originally designed for updating volatile data can now be repurposed for updating persistent data. Fig. 2 shows five existing data update mechanisms.

Dedicated-Cache Update. *Dedicated-cache update* allocates a dedicated cache space within persistent CPU caches for BAS systems using cache allocation technologies (e.g., Intel's Cache Allocation Technology [20], AMD's Platform Quality of Service [2], and ARM's Memory Partitioning and Monitoring [4]) [70]. This reduces BAS bandwidth usage by mitigating cache contention, and cached data is automatically persisted to BAS during a power outage.

Cache allocation technologies include two related features: Cache Pseudo-Locking and L3_CAT. Cache Pseudo-Locking [24], supported only by Intel, is not architecturally supported after the Broadwell generation released

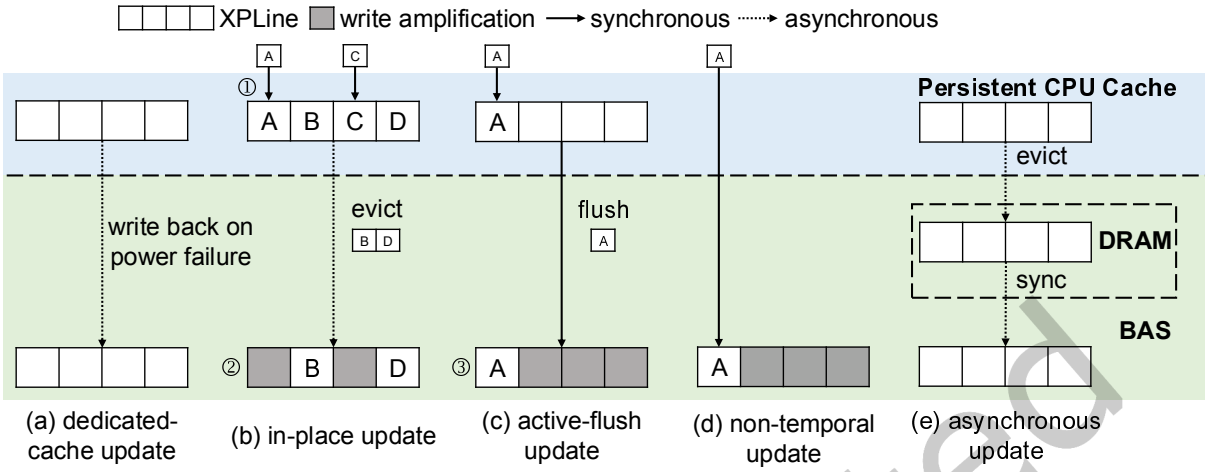

Fig. 2. BAS Data update mechanisms.

in 2014 [22, 23]. L3_CAT supports recent CPUs of various brands (e.g., Intel CPUs since Xeon E5 v3 [31], AMD CPUs since EPYC Rome [44], and ARM CPUs [5]) through Linux's `resctrl` interface [25], but has different characteristics. To our knowledge, no other BAS system has used L3_CAT to mitigate cache contention.

Unlike Cache Pseudo-Locking, which locks memory segments in the caches, L3_CAT allocates a portion of the caches to L3_CAT groups that contain specific PIDs or CPU cores. Processes can only allocate cachelines to their assigned LLC ways, but can still load/update cachelines from all LLC ways. Programmers can take advantage of L3_CAT by simply accessing the appropriate Model-Specific Registers (MSRs) or using high-level libraries [31]. Furthermore, dynamic mechanisms can be built on top of it [61].

However, L3_CAT can only isolate cache usage from other processes and does not completely avoid cache contention. In addition, due to ID-based allocation, L3_CAT cannot directly affect kernel-level operations. This is because system calls run on the CPU core that initiates the call, which is not fixed, and they do not belong to any PID. A workaround is to use L3_CAT to protect all processes that initiate system calls, but this greatly expands the system's WSS by including the WSS of all writer processes, potentially exceeding cache capacity. *Dedicated-cache update is suitable for updating hot data when there are concurrent interfering processes, but its effectiveness for kernel and other types of cache contention remains to be addressed.*

In-Place Update. *In-place update* omits issue flush instructions after writes, as these operations are no longer needed to ensure consistency with cache persistence techniques. This mechanism minimizes update latency by reducing synchronous data persistence overhead [33, 46, 60, 63]. The bandwidth consumption of this mechanism varies depending on the access pattern. On the one hand, when updating hot data, it uses CPU caches to buffer multiple writes to the same XPLine (Step 1 in Fig. 2). On the other hand, for cold data exceeding one cacheline (> 64B), it can result in write amplification due to the cacheline eviction policy (Step 2 in Fig. 2). Worse, it is susceptible to interference from offending applications that frequently request large amounts of data but rarely reuse the cached data, such as file hosting and video streaming programs [20]. *In-place update is suitable for updating small (≤ 64B) cold data, as well as for updating hot data when there are no interfering processes.*

Active-Flush Update. *Active-flush update* explicitly issues flush instructions after writes to proactively flush data into BAS, coupled with memory barriers for immediate data persistence. It is a widely adopted mechanism in ADR-based systems [15, 45, 56, 57] to ensure consistency. Although flushing becomes unnecessary for consistency, *active-flush update* prevents random cacheline eviction by aggregating large writes in the internal buffer and

writing them to BAS sequentially, mitigating write amplification. However, it cannot alleviate write amplification if the access size is within a cacheline (Step 3 in Fig. 2). *Active-flush update is suitable for updating cold data with more than one cacheline (> 64B).*

Non-Temporal Update. *Non-temporal update* directly writes data to BAS, bypassing CPU caches using non-temporal write instructions. Like *active-flush update*, it is widely used by ADR-based systems [13, 35] to ensure data consistency. Compared to *active-flush update*, it exhibits lower latency and higher bandwidth by avoiding loading data into CPU caches [58]. *Non-temporal update is suitable for updating cold data with more than one cacheline (> 64B).*

Asynchronous Update. *Asynchronous update* buffers writes in DRAM and persists them asynchronously. This mechanism effectively reduces critical path latency and BAS bandwidth usage, but also introduces the risk of data loss during power failures [45, 55, 67, 68]. To obtain immediate persistence in file system access, applications must explicitly call `fsync()` to synchronously persist previously buffered writes to BAS. *Asynchronous update is only suitable for scenarios with relaxed consistency requirements.*

These update mechanisms provide different choices for updating data in BAS, and their effectiveness depends on the access pattern. To optimize performance, it is essential to dynamically choose the most suitable mechanism according to the access pattern. However, current BAS systems do not adaptively determine the appropriate data update mechanism according to the access pattern, instead adhering to a rigid approach.

3 Characterization of BAS-related Cache Contention

In this section, we specifically identify and characterize several types of cache contention that remain challenging for existing BAS systems to mitigate, including *DRAM-BAS contention* (Section 3.1), *set contention* (Section 3.2), *interfering process contention* (Section 3.3), and *I/O contention* (Section 3.4). Such contention 1) causes cached data to be frequently evicted to BAS and reloaded into caches, negating the benefits of caching, and 2) can lead to write amplification if the write granularity of the BAS is large. Despite its importance, the issue of cache contention has been largely overlooked in existing BAS systems. Therefore, we present a set of guidelines aimed at mitigating each type of contention.

3.1 DRAM-BAS Contention

DRAM-BAS contention occurs when the combined cache requirements of a BAS system's DRAM and BAS exceed the available cache capacity, resulting in competition for cache resources. Some BAS systems flush cold data to limit BAS WSS to prevent *BAS-BAS contention* [33, 63], but the systems' DRAM accesses can also compete for cache space. To understand the effects of *DRAM-BAS contention*, we perform an experiment on the platform described in Section 5.1. In the experiment, the process simultaneously overwrites both a BAS buffer of the same size as the allocated caches and a DRAM buffer of variable size. The BAS buffer is allocated via memory mapping on the ext4-DAX file system. We monitor results using `perf` for cache miss rates and `ipmctl` for BAS media writes. L3_CAT with flush is used as the baseline for I/O amplification. Test results are consistent for both sequential and random access patterns and for 32B/4KB access sizes.

Our results, shown in Fig. 3, show a direct correlation between the DRAM buffer size and the increase in I/O amplification. Specifically, when the DRAM buffer size reaches 7MB, I/O amplification is about 1, meaning that BAS writes catch up with the bandwidth to flush all writes. The results highlight a critical challenge for BAS systems: they often store recoverable metadata in DRAM to speed up accesses, but as the metadata grows, DRAM accesses can often evict cached BAS data. In addition, using L3_CAT to isolate kernel file systems by protecting all processes that initiate system calls becomes impractical due to the DRAM accesses of these processes. Instead, proactively calling flush instructions after DRAM accesses (*active-flush update*) or using `ntstore` to access DRAM (*non-temporal update*) can effectively limit DRAM WSS and thus mitigate *DRAM-BAS contention.*

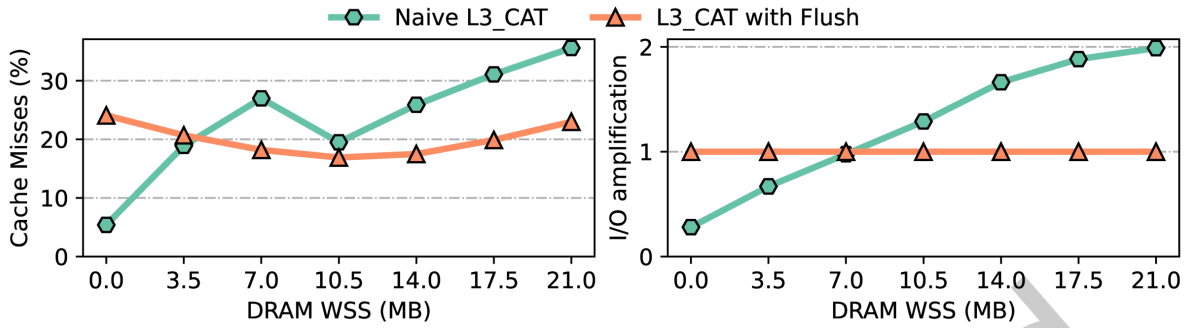

Fig. 3. Cache miss rates and I/O amplification when the process overwrites a BAS buffer of the same size as the allocated caches (21MB) and a DRAM buffer of variable size.

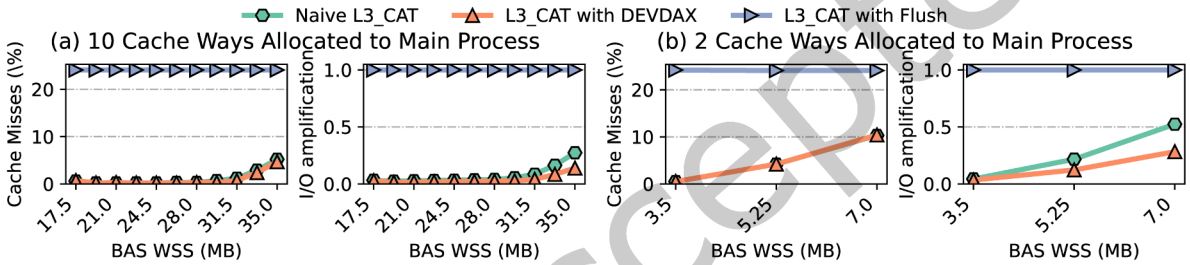

Fig. 4. Cache miss rates and I/O amplification when the process writes to a BAS buffer of variable size for various L3_CAT configurations.

Guideline 1: BAS systems can use *active-flush/non-temporal updates* for DRAM accesses to minimize the eviction of cached BAS data caused by DRAM accesses.

3.2 Set Contention

Set contention occurs in set-associative caches when multiple memory accesses compete for the same cache set, resulting in cache evictions even when the cache isn't full. In modern CPUs, the L3 caches are segmented into numerous sets, with each set consisting of multiple ways. For example, Intel Xeon Gold 6348 equipped on our platform has 57,344 sets with 12 ways per set [29]. One cache way corresponds to one twelfth of the L3 caches, or 3.5MB caches. Physical addresses are mapped to free ways in the corresponding set using a hash function. This architecture is intended to speed up cache access, but severely limits the number of ways per set, which can lead to *set contention* as the WSS approaches cache capacity. *Set contention* exists even without L3_CAT, and is exacerbated in L3_CAT-protected systems because L3_CAT imposes an additional limit on the number of cache ways the system can access.

We design an experiment to evaluate the impact of *set contention* on BAS systems. Specifically, we allocate either 10 cache ways (35MB) or 2 cache ways (7MB) of cache to the process. This setup allows us to highlight the increased severity of *set contention* when fewer cache ways are available. The process then overwrites a variable-sized BAS buffer. Our results, shown in Fig. 4, show that the cache miss rate remains low as long as the size of the cached data is smaller than the allocated cache size by at least one cache way (3.5MB). However, as the

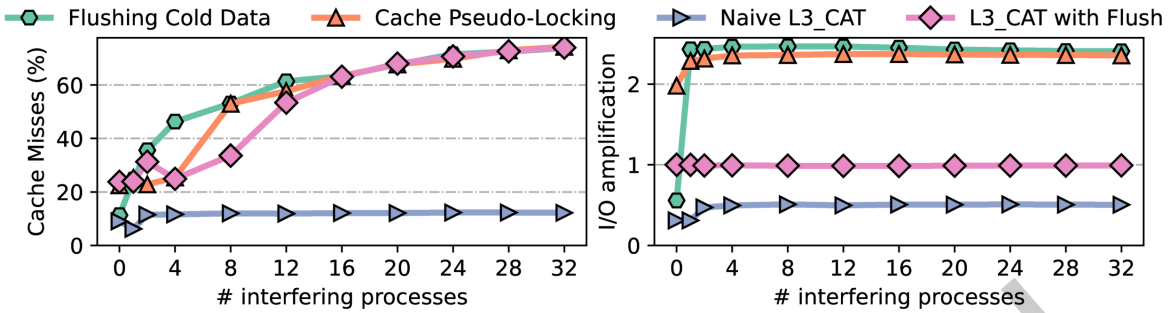

Fig. 5. Cache miss rates and I/O amplification when the BAS system process overwrites a BAS buffer and various numbers of interfering processes overwrite private DRAM buffers under different cache contention mitigation approaches.

available cache ways continue to decrease and there is less than one cache way available, the cache miss rate begins to increase, indicating increased *set contention*. This contention increases as the available cache capacity decreases. Fig. 4(b) shows that when only two cache ways are allocated and the size of the cached data is equal to the allocated cache size, BAS writes can reach half the bandwidth to flush all writes.

Another of our findings is that using devdax mode to allocate BAS buffers can reduce BAS writes caused by *set contention* by approximately 50%. This significant reduction is due to devdax's ability to provide more raw access to BAS, making it easier for an application to guarantee alignment for large pages. This finding is crucial because *set contention* can be exacerbated by an unfavorable access pattern to the cache's hash function, leading to an imbalance in set utilization [16, 50].

Guideline 2: BAS systems can mitigate *set contention* by reserving at least one cache way in the dedicated cache, or by using devdax mode to balance set usage.

3.3 Interfering Process Contention

Interfering process contention refers to the competition for cache space between BAS systems and other interfering processes unrelated to BAS. Early systems based on persistent CPU caches [46, 60] omit flush instructions after writes without considering that interfering processes may evict cached BAS data. Recent systems [33, 63, 70] attempt to address this by flushing cold data and assuming that cached data is hot enough to survive cache contention. However, cache eviction strategies on Intel CPUs are undocumented and not simply hotness-based [8], and it is difficult to guarantee that cached BAS data is hotter than data from other processes. The deprecated Cache Pseudo-Locking and its successor L3_CAT are intended to reserve cache space for BAS systems, but their effectiveness remains to be tested.

We conduct an experiment to evaluate the impact of *interfering process contention* on BAS systems. In our tests, the BAS system process overwrites a BAS buffer 100 times, while multiple interfering processes overwrite their 100MB private DRAM buffers. The BAS buffer size is set to match the maximum available cache size. For flushing cold data, this is the entire cache size (42MB). Otherwise, it is 11 of the 12 cache ways (38.5MB), since we must reserve at least one cache way (3.5MB) for other processes. This setup is designed to allow the BAS system to get the most out of the cache space while simulating a situation where the BAS data is hotter than the DRAM data due to the smaller BAS data.

As shown in Fig. 5, our results indicate that neither flushing cold data nor using Cache Pseudo-Locking effectively mitigates cache contention. Flushing cold data can reduce BAS writes in the absence of interfering processes, but even a single interfering process can significantly increase BAS writes, even exceeding the value

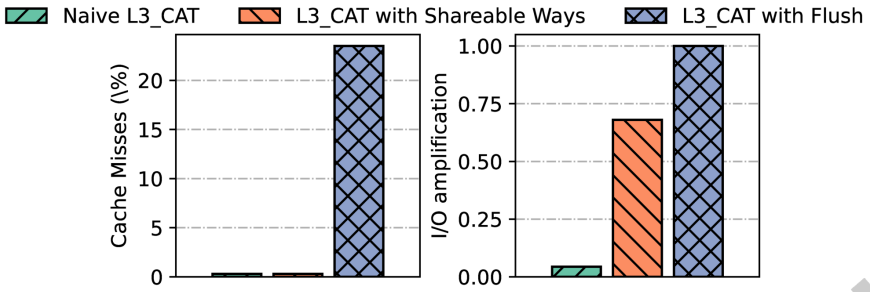

Fig. 6. Cache miss rates and I/O amplification when the BAS system process writes to a 3.5MB BAS buffer with 7MB of different cache ways allocated.

when hot data is also flushed. Conversely, the deprecated Cache Pseudo-Locking tends to increase BAS writes even in the absence of interfering processes. As for L3_CAT, cache miss rates and I/O amplification are not significantly affected by interfering processes, but have a non-zero lower bound due to *set contention*.

Guideline 3: BAS systems should use L3_CAT rather than unsupported Cache Pseodo-Locking to isolate cached BAS data from cached DRAM data of interfering processes.

3.4 I/O Contention

I/O contention occurs when I/O devices compete with BAS systems for cache space, causing cache evictions even when cache is allocated to BAS systems with L3_CAT. This contention is caused by Direct Cache Access (DCA) technologies, such as Intel's Data Direct I/O (DDIO) [32] and ARM's Cache Stashing [6], which allow I/O devices to inject incoming I/O traffic directly into CPU caches instead of memory. In this paper, we focus on Intel's DDIO because it is the most widely used DCA technology. By default, DDIO can only allocate on the two leftmost LLC ways[1] [61] (i.e., the shareable ways), while it can update or read data from all ways. For DRAM systems, DDIO can improve system performance by reducing memory access latency and memory bandwidth consumption [54]. However, it inevitably evicts cachelines when caches are full and causes write amplification for BAS systems. Here we categorize *I/O contention* into two types, 1) *intra-I/O contention*: multiple I/O requests compete for cache space in systems that access BAS with I/O devices (e.g., RDMA NICs [54], DMA engines [41]), and 2) *system-I/O contention*: BAS systems that do not use I/O devices compete for cache space with concurrent I/O requests.

We evaluate the impact of *system-I/O contention* through experiments since we focus on monolithic systems. Specifically, we allocate 7MB of cache to the BAS system process, which overwrites a 3.5MB BAS buffer, while simultaneously performing DDIO-enabled DMA requests to write DRAM data on the same NUMA node. Our results, shown in Fig. 6, indicate that allocating the shareable ways to the BAS system process significantly increases BAS writes, approaching the bandwidth to flush all data. In contrast, allocating other non-shareable ways results in near-zero cache miss rates and BAS writes. This finding suggests that even when cache ways are made exclusive to BAS systems using L3_CAT, there is still *I/O contention* if the assigned ways overlap with shareable ways. Fortunately, L3_CAT can still be used to prohibit BAS systems from allocating cachelines in shareable ways to prevent *I/O contention*.

A common approach in BAS systems is to disable DDIO for BAS-related I/O devices, which also persists data on platforms with volatile CPU caches. However, disabling DDIO is ineffective for *system-I/O contention* and

[1]This value is defined in a Model Specific Register (MSR) called "IIO LLC WAYS" at address 0xC8B and can be read/written using msr-tools (e.g., rdmsr and wrmsr).

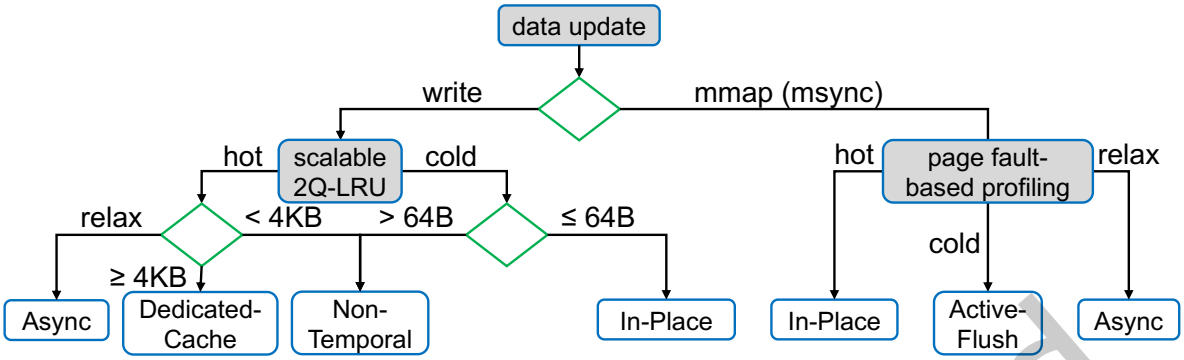

Fig. 7. Update policy of FusionFS.

platforms with persistent CPU caches. On the one hand, if DDIO is disabled, incoming data will still be in the cache initially and will be flushed to memory immediately, potentially evicting cached BAS data when the cache is full [61]. On the other hand, on platforms since Ice Lake, all DDIO-related registers are read-only [47]. Some vendors offer a workaround to globally disable DDIO by disabling Intel Virtualization Technology (Intel VT) in the BIOS. However, this workaround has significant drawbacks: 1) it severely degrades I/O performance when accessing DRAM, 2) it disables other virtualization features, potentially impacting system functionality, and 3) not all vendors support this option. As a result, simply disabling DDIO is an impractical solution for mitigating *system-I/O contention* or adapting to modern platforms.

Guideline 4: When DCA is enabled, BAS systems should use L3_CAT to avoid using shareable ways.

4 Design and Implementation

In this section, we introduce FusionFS, a contention-resilient kernel file system that exploits persistent CPU caches to redesign data update approaches. We propose an *adaptive data update* approach to select the optimal data update mechanism from the five update mechanisms based on access patterns (Section 4.1). Following our guidelines, we propose a *contention-aware cache allocation* approach to prevent various types of cache contention (Section 4.2).

4.1 Adaptive Data Update

In this section, we first introduce the update policy of FusionFS during system call accesses (Section 4.1.1) and memory mapping accesses (Section 4.1.2), as shown in Fig. 7. We then present the *scalable 2Q-LRU* and *page fault-based profiling* approaches, which transparently measure data hotness even for memory mapping accesses that bypass the file system to access BAS directly (Section 4.1.3).

4.1.1 System Call Accesses. FusionFS selects the most suitable data update mechanism based on access patterns during system call accesses (e.g., *write*). For large (≥ 4KB) hot data, FusionFS uses *dedicated-cache 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 BAS and leveraging the benefits of cache hits. In addition, *dedicated-cache update* can protect these data segments from *interfering process contention*. For small (< 4KB) hot data, FusionFS uses *non-temporal update* because it has higher bandwidth for I/O sizes less than 4KB. 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, avoiding write amplification and leveraging DRAM's high performance.

For small (≤ 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. Conversely, for large (> 64B) cold data, FusionFS employs *non-temporal update* rather than *active-flush update* due to its lower latency and higher bandwidth.

4.1.2 Memory Mapping Accesses. Memory-mapped file support is critical because *mmap* is an important method for accessing BAS that allows applications to achieve near-native BAS performance. While some BAS-aware applications use flush instructions during *mmap* accesses, legacy applications still rely on synchronization calls to flush data.

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 to reduce writes to BAS. *Dedicated-cache updates* cannot be used because memory-mapped data is accessed by userspace processes, not the file system. For cold data, FusionFS uses *active-flush update* to aggregate the sequential writes in BAS's internal buffer. For hot data with relaxed consistency, FusionFS uses *asynchronous update* by buffering data pages in DRAM and persisting them to BAS during synchronization calls. Since applications often synchronize the entire file regardless of the *mmap* write area, if a strictly consistent file has multiple full-file sync calls within a second, FusionFS will batch them into a single full-file sync to further reduce overhead. This does not affect consistency thanks to persistent CPU caches.

4.1.3 Hotspot Detector. We design a lightweight yet effective hotspot detector to measure the hotness of data pages during system call and memory mapping accesses without requiring code changes to user applications.

Scalable 2Q-LRU. For system call accesses, we design a *scalable 2Q-LRU* algorithm to minimize cache thrashing. The 2Q-LRU algorithm uses two queues, the recent queue and the frequent queue. Data pages enter the recent queue on the first access and move to the frequent queue on subsequent visits. The algorithm uses a read-write semaphore to provide thread safety. Changes to the 2Q-LRU queues are cached in per-CPU buffers and bulked into the queues until they are full. The size of identified hot data is limited to avoid *set contention*.

Page fault-based profiling. For memory mapping accesses, obtaining access information transparently is challenging because applications bypass the file system to access BAS directly. To address this, FusionFS adopts a *page fault-based profiling* approach. FusionFS periodically adjusts the permissions of data pages in a background kernel thread. This allows the next access to be captured by the page fault handler function. The kernel thread then performs a translation lookaside buffer (TLB) shootdown and sends inter-processor interrupts (IPIs) to synchronize the TLB across all CPUs. Finally, FusionFS appends captured mmap accesses to appropriate 2Q-LRU queues for hot data detection. This process is only triggered once per profiling period for each page, so the overhead is negligible.

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 BAS bandwidth utilization and lower latency across various scenarios, addressing the limitations observed in conventional approaches.

4.2 Contention-Aware Cache Allocation

Following our guidelines in Section 3, we propose *isolated data access* (Section 4.2.1), *associativity-friendly data layout* (Section 4.2.2), and *DCA-aware way allocation* (Section 4.2.3) to address *DRAM-BAS*, *set*, and *I/O contention*, respectively. These designs also make L3_CAT effective for kernel space and allow FusionFS to use CPU caches of all NUMA nodes.

4.2.1 Isolated Data Access. FusionFS addresses the challenges of *DRAM-BAS contention* and unsupported L3_CAT in kernel space by offloading *dedicated-cache updates* to L3_CAT-protected kernel threads. Kernel threads are actually processes cloned from process 0 (the swapper). Unlike system calls and user threads, kernel threads not only have their own PIDs, which are required for L3_CAT, but also have access to both the kernel and user address spaces [30, 48, 66, 71]. The unique feature of kernel threads allows them to copy data from user address space buffers to BAS located in the kernel address space, while protecting the cached BAS data from being evicted by other processes via L3_CAT, and isolating the kernel file system's BAS WSS from the system and calling processes' DRAM WSS.

FusionFS uses several methods to reduce the overhead of offloading *dedicated-cache updates* to kernel threads. First, FusionFS assigns a fixed kernel thread to each BAS region according to the hash function of the address. This ensures that the same BAS region is always accessed by the same kernel thread, reducing the cache coherence overhead caused by multiple kernel threads on different CPU cores accessing the same BAS region. Second, FusionFS allocates private ring buffers to kernel threads to improve scalability. Third, FusionFS uses offloaded *dedicated-cache updates* only for large (≥ 4KB) hot data, since the overhead of offloading small data updates is non-trivial according to the experimental results in Section 5.4. Finally, FusionFS performs the 2Q-LRU algorithm outside of the kernel threads to reduce kernel thread access to DRAM, thus mitigating *DRAM-BAS contention*. We also test a workaround that uses L3_CAT-protected kernel threads to load BAS data into dedicated CPU caches on the first access, and then perform subsequent accesses directly without offloading. However, we find that after non-offloaded accesses, the cached data is no longer protected by L3_CAT and is vulnerable to *interfering process contention*.

4.2.2 Associativity-Friendly Data Layout. FusionFS leaves a one-way gap between the WSS and the cache capacity to mitigate *set contention*. As mentioned in Section 3.2, using `devdax` mode to get more raw access can reduce *set contention*. However, we also note that the kernel already provides raw access to BAS, so the physical addresses of pages with contiguous kernel virtual addresses are also contiguous. Therefore, FusionFS can mitigate *set contention* without relying on `devdax`, thus providing better support for other BAS devices such as CXL-SSDs.

Additionally, when FusionFS is mounted on multiple NUMA nodes, FusionFS spreads the file data across the kernel threads of different NUMA nodes in a RAID0-like fashion. This allows FusionFS to use the CPU caches of other NUMA nodes by offloading *dedicated-cache updates* to kernel threads on those nodes, even if the processes initiating the write calls are running on a single NUMA node. Because the data is copied from the local DRAM of the initiating NUMA node to the CPU caches of the kernel threads on other NUMA nodes during write operations, it does not suffer from BAS's poor cross-NUMA access performance. With this optimization, FusionFS can use the CPU caches of all NUMA nodes to extend the dedicated cache capacity.

4.2.3 DCA-Aware Way Allocation. FusionFS addresses the challenge of *I/O contention* caused by Direct Cache Access (DCA) by preferentially allocating cache ways that are not used by DCA for *isolated data accesses*. This approach ensures that incoming I/O traffic that DCA allows to be injected directly into CPU caches will not evict cached BAS data without disabling DCA. In fact, disabling DCA does not work on monolithic systems and is not feasible for eADR platforms (Section 3.4). While other processes and I/O operations share the shareable ways, they are less affected by *I/O contention* due to the higher bandwidth and matched access granularity of DRAM.

By mitigating *DRAM-BAS*, *set*, and *I/O contention* and making L3_CAT effective for kernel space, FusionFS addresses the limitations of L3_CAT and ensures that the dedicated caches are used effectively. In addition, FusionFS can use the CPU caches of all NUMA nodes, allowing more data to be stored in the caches. This further reduces the need to access BAS, improving the overall performance of the file system.

5 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 effectively mitigate various types of cache contention with *contention-aware cache allocation*?
- Can FusionFS exhibit optimal performance in application scenarios of system calls and memory mapping?

5.1 Evaluation Methodology

Experimental platform. We perform our evaluation on a server equipped with two 28-core Intel Xeon Gold 6348 (42MB cache), four 128GB Intel Optane PM, which is a type of BAS, and four 16GB DDR4 DRAM on each node. The server is running Linux kernel v5.13.13.

FusionFS implementation and configuration. We modify and extend the PMFS-based [15] ODINFS [71] to design and implement FusionFS. ODINFS is a PM file system that introduces a data movement delegation mechanism used in non-PM systems [30, 48, 66], where background threads access PM on behalf of applications to limit concurrent PM accesses and use the PM bandwidth of multiple NUMA nodes. As a result, FusionFS provides the same level of consistency as PMFS and ODINFS, i.e. all metadata operations are synchronous and atomic, and all data operations are synchronous but not atomic. Many applications that write to the file system, such as SQLite and LevelDB, ensure consistency through their own logging mechanisms and do not require the file system to provide atomicity for data updates. If needed, we can extend FusionFS to provide atomicity of data operations using logging [33] or hardware transactional memory (HTM) [60, 63]. For a fair comparison, no *asynchronous updates* are used. Unless otherwise mentioned, we configure FusionFS to have 12 kernel threads and 21MB of dedicated cache per NUMA node. FusionFS can also be extended with cache management mechanisms [61] to dynamically adjust the dedicated cache capacity.

Target comparisons. We evaluate and compare FusionFS with five file systems: ext4, PMFS [15], NOVA [57], WineFS [34], and ODINFS [71]. We configure ext4 with the DAX option and all other file systems with the default setup. These file systems provide weaker or the same level of consistency as FusionFS. Unless otherwise mentioned, all file systems run on a single NUMA node.

5.2 Micro-benchmarks

5.2.1 Fio. We evaluate the file write performance of FusionFS using Fio [17], testing both 4KB and 2MB access sizes with 1 and 8 threads. We vary the Zipf parameter θ to present results with localities ranging from 90/10 (90% of accesses go to 10% of data) to 50/50 (uniform distribution). We configure fio to let each thread access a 16MB private file.

Fig. 8 shows the throughput, average latency, and I/O amplification of all file systems evaluated. We report the I/O amplification as the number of bytes written to the underlying BAS media divided by the number of bytes issued by the CPUs. For single-threaded 4K writes, FusionFS outperforms the throughput of other file systems by an average of 67.4%. FusionFS also shows near-zero latency and I/O amplification because the single 16MB file can fit into dedicated cache capacity. In contrast, other file systems have higher latency and I/O amplification around 1 because they flush every write to BAS. In particular, ODINFS has significantly lower throughput and higher latency due to the communication overhead with kernel threads and the inability to use the BAS bandwidth of multiple NUMA nodes when only one node is available, while FusionFS avoids the cache coherency overhead and benefits from the high bandwidth of CPU caches. For single-threaded 2M writes, the average throughput advantage for FusionFS increases to 343.3% because the offloading overhead for large writes is negligible.

For 8-threaded 4K writes, FusionFS's performance metrics improve with increasing locality, with throughput averaging 42.9% higher than the other filesystems. This is because the size of the eight 16MB files far exceeds the dedicated cache capacity, and the higher locality brings a higher probability that the target data block is located

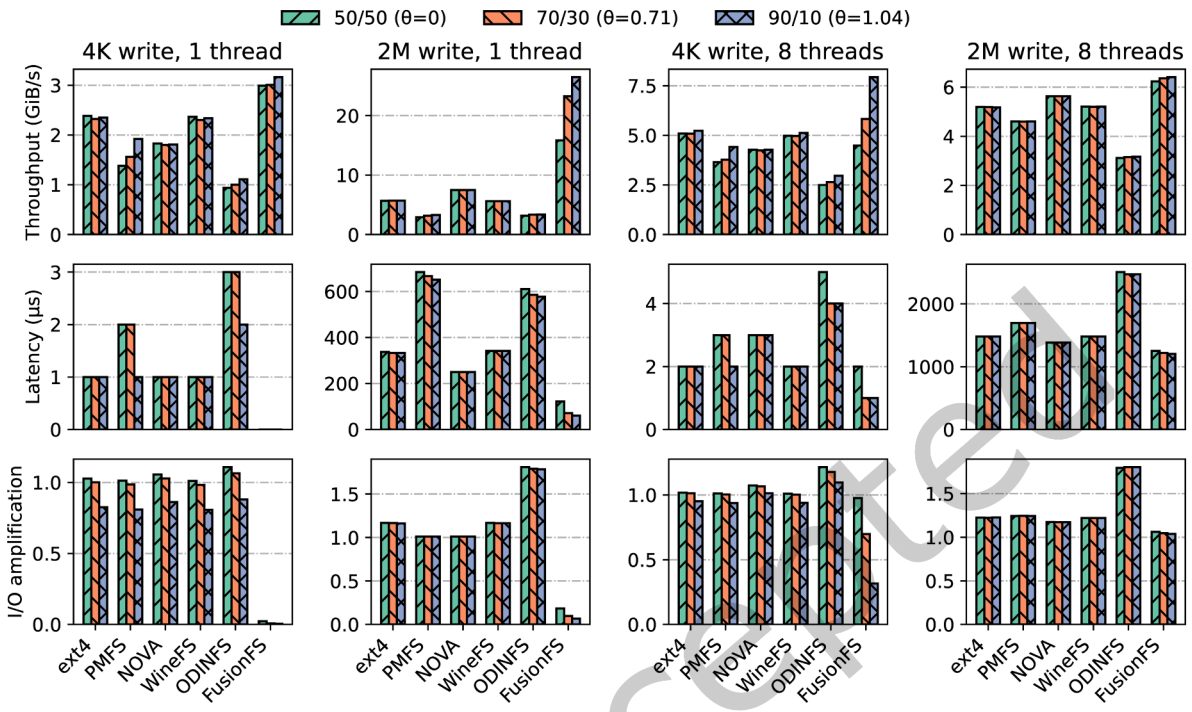

Fig. 8. Throughput, latency, and I/O amplification of evaluated file systems in Fio.

Name	Description
DRBL	Each thread reads a private block in a private file.
DRBM	Each thread reads a private block in a shared file.
DRBH	Each thread reads a shared block in a shared file.
DWOL	Each thread overwrites a private block in a private file.
DWOM	Each thread writes to a private block in a shared file.

Table 2. Summary of used FxMark workloads. Each thread repetitively performs the corresponding operations in each workload.

in the CPU caches. For 8-threaded 2M writes, although large file sizes and write sizes make random accesses nearly uniformly distributed regardless of θ, FusionFS throughput is still 33.2% higher on average due to the higher bandwidth and lower latency of *non-temporal updates* compared to *active-flush updates* used by other file systems.

5.2.2 FxMark. We use the FxMark [43] workloads described in Table 2 to evaluate the performance and scalability of FusionFS data operations. Fig. 9 shows the scalability results of the evaluated file systems. Among them, PMFS and NOVA can only scale the DRBL workload. Instead, FusionFS and ODINFS can scale all the benchmarks with the readers-writer range lock [12]. For read workloads, FusionFS and ODINFS are 16.9% slower than PMFS in

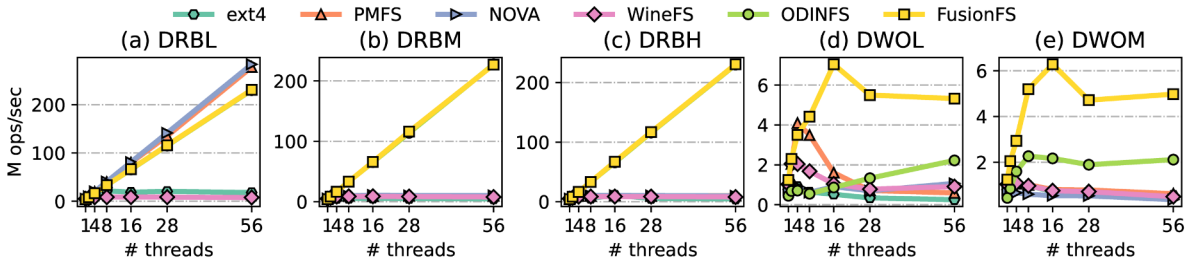

Fig. 9. Results of FxMark workloads.

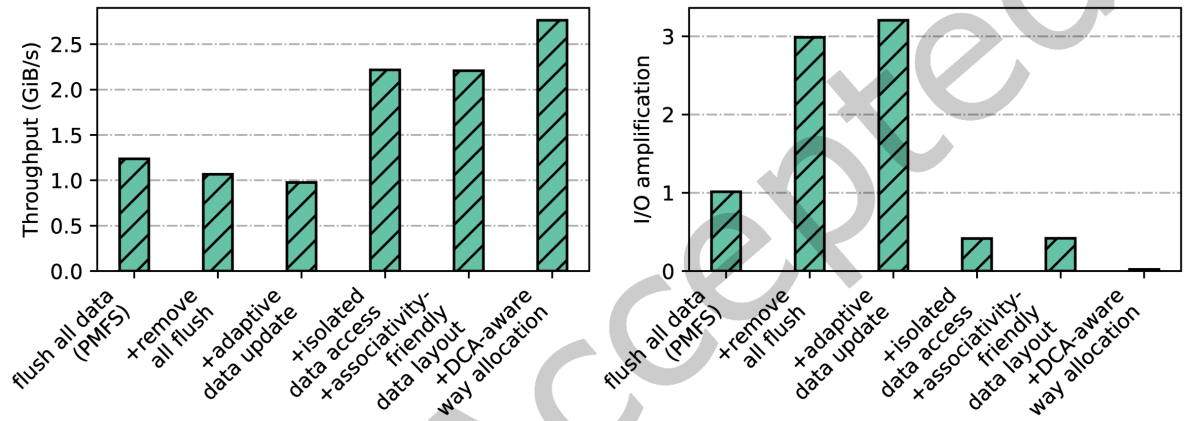

Fig. 10. Breakdown analysis of FusionFS with Fio workloads as optimizations are gradually enabled.

DRBL. However, they outperform other file systems by about 23.2× and 22.8× in DRBM and DRBH, respectively, because other file systems are limited by the readers-writer semaphore implementation in the Linux kernel.

For write workloads, FusionFS outperforms other file systems by about 1.6× and 1.5× on average in DWOL and DWOM, respectively. This is because in FusionFS, writing to hot data is likely to hit CPU caches, while in other file systems, throughput is limited by BAS bandwidth as they eagerly flush the written data to BAS or use non-temporal write instructions to store data. Compared to ODINFS, which limits the number of BAS write threads to no more than 8 to avoid performance collapse, FusionFS does not limit access to cached data and achieves better scalability.

5.3 Breakdown Analysis

We use a Fio workload to investigate the throughput and I/O amplification improvements of each of FusionFS's optimizations when updating cached data under cache contention. The workload performs 4KB I/Os on a 16MB file with a single thread. We also spawn 8 interfering processes and I/O requests as described in Section 3.

Fig. 10 shows the throughput and I/O amplification for the tests. Removing all flushes results in a 13.7% decrease in throughput and a 194.4% increase in I/O amplification compared to flushing all data (i.e., PMFS). This is because writes larger than a cacheline can be split into multiple cacheline-sized random writes due to the cacheline

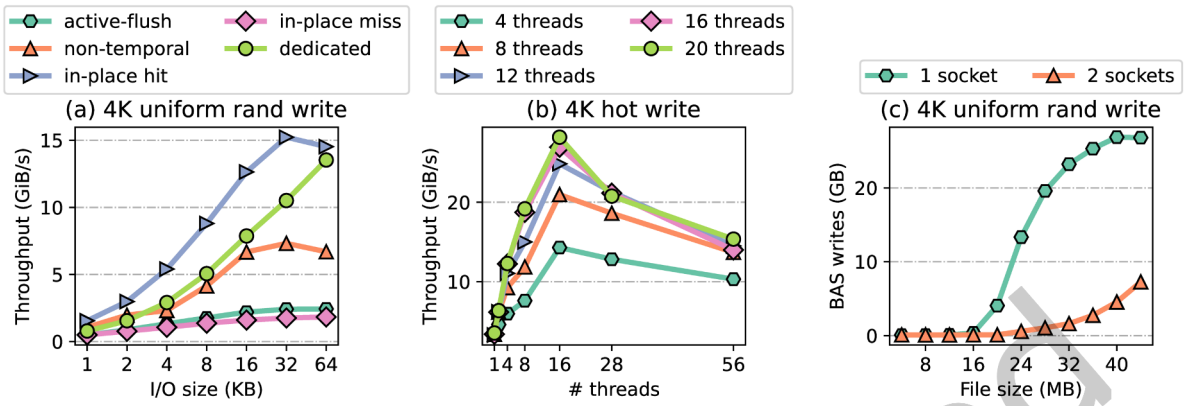

Fig. 11. Results of sensitivity analysis.

eviction policy, which does not match the access granularity of BAS. Cacheline evictions are further exacerbated by *BAS-BAS contention* and *interfering process contention*, causing severe write amplification.

When we start mitigating *BAS-BAS contention* with *adaptive data update*, throughput drops by 8.5% due to persistence overhead. However, I/O amplification even increase by 7.3% because cached data is still vulnerable to *interfering process contention*. This suggests that *adaptive data update* alone is not effective in mitigating *interfering process contention* because there is no guarantee that detected hot data is hotter than data from other processes.

Next, we offload *dedicated-cache updates* to L3_CAT-protected kernel threads to mitigate *interfering process contention*. The *isolated data access* optimization allows FusionFS to outperform PMFS under cache contention, with throughput increases of 127.2% and I/O amplification reductions of 87.0%. The *associativity-friendly data layout* does not significantly change the results because the addresses of the files created by Fio in the freshly initialized FusionFS are contiguous and the file size is smaller than the dedicated cache capacity. Finally, *DCA-aware way allocation* further mitigates *I/O contention* by preventing FusionFS from sharing caches with DCA-enabled I/O operations, increasing throughput by 25.1% and reducing I/O amplification by 95.2%.

In summary, combining all optimizations results in a throughput increase of 123.5% and a reduction in I/O amplification of 98.0%. This indicates that FusionFS's cached data is virtually immune to various types of cache contention and consumes near-zero BAS bandwidth when writing cached data due to *contention-aware cache allocation*.

5.4 Sensitivity Analysis

This section describes how I/O size thresholds, kernel thread count, and NUMA node count affect FusionFS performance.

FusionFS with varying I/O sizes. We run Fio workloads to evaluate the performance of different data update mechanisms under different I/O sizes. We generate single-threaded uniform random write requests with different I/O sizes ranging from 1KB to 64KB. We use a 16MB file to simulate a scenario where BAS accesses can hit CPU caches, and a 1GB file to simulate a scenario where BAS accesses cannot hit CPU caches.

Fig. 11(a) shows the results. *In-place updates* that hit CPU caches have the best throughput at small I/O sizes ($\leq$ 64KB), and the throughput decreases as the I/O size increases. However, if they cannot hit CPU caches due to cache contention, the throughput will be worst due to write amplification caused by random cacheline evictions and mismatched access granularities. *Dedicated-cache updates* mitigate *interfering process contention*, but also incur

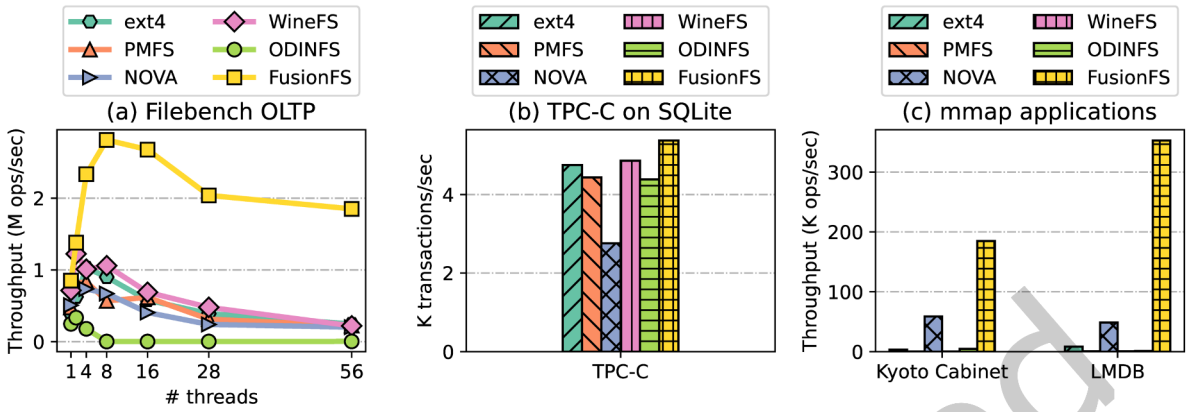

Fig. 12. Results of application benchmarks.

communication overhead. By applying a variety of optimizations, they outperform other options under cache contention for I/O sizes greater than or equal to 4KB. Unlike NOVA, which only uses *non-temporal updates* when the platform does not support `clflushopt` and `clwb`, FusionFS prefers *non-temporal updates* for system calls because they always have higher throughput than *active-flush updates*. For memory mapped cold data, FusionFS flushes data already written to CPU caches into the BAS during synchronization calls, so only *active-flush updates* can be used.

FusionFS with varying kernel threads. We run a multithreaded Fio workload with 4KB I/O size and 4KB file size and vary the number of kernel threads to find the optimal number for FusionFS's *isolated data access*, which is a common practice for data copy offloading [36, 41, 71, 72]. As Fig. 11(b) shows, the throughput approaches saturation at 12 kernel threads and continues to increase slightly up to 20 threads. However, beyond 12 threads, the incremental gains in throughput are minimal compared to the increase in CPU usage. Therefore, we choose 12 kernel threads as the default setup for FusionFS because it balances throughput and CPU usage.

FusionFS with varying NUMA nodes. Fig. 11(c) shows FusionFS's BAS media writes when running the uniform random write Fio workload with different file sizes on different numbers of NUMA nodes. When running on two NUMA nodes, the BAS writes begin to increase with larger file sizes. The results show that FusionFS can use CPU caches on other NUMA nodes by binding kernel threads to them, even if the process that initiated the write call is not on them.

5.5 Application Benchmarks

This section tests FusionFS in real-world application scenarios of system calls and memory mapping.

5.5.1 Filebench OLTP. Filebench [53] OLTP contains a single log writer process writing a 10MB log file with an I/O size of 256KB and multiple database writer/reader processes writing/reading a 10MB data file with an I/O size of 2KB. We configure OLTP to initiate transactions at a high frequency to better match the high performance of BAS.

Fig. 12(a) shows that FusionFS outperforms the second-ranked WineFS by up to 3.4×. This is because for 256KB log writes, FusionFS uses *dedicated-cache updates*, which are likely to hit CPU caches, rather than flushing all writes to BAS as other file systems do, thus taking advantage of the high performance of CPU caches. For 2KB database writes, FusionFS uses *non-temporal updates* to avoid the performance degradation typically associated

with small I/O sizes. In addition, FusionFS achieves good scalability with the readers-writer range lock, which ensures efficient concurrent access within the same file.

5.5.2 TPC-C on SQLite. SQLite [51] is a lightweight database that stores data in a single B+-Tree file, with other auxiliary files for logging. We run the OLTP benchmark TPC-C on SQLite in Write-Ahead-Logging (WAL) mode, which contains three types of files: the main database files (~350MB), WAL files (~4MB), and memory mapped SHM files (~32KB). The I/O size for write calls is 4KB.

Fig. 12(b) shows that FusionFS outperforms other file systems by up to 2.0× in the TPC-C on SQLite workload by adaptively choosing the most appropriate data update mechanism for the access pattern. While 4KB writes to the main database files can be offloaded to kernel threads, the lower locality due to the larger file size compared to Filebench OLTP leads FusionFS to use *non-temporal updates* for these writes. In contrast, the high access frequency of the WAL files allows FusionFS to use *dedicated-cache updates* to store the data CPU caches.

5.5.3 Kyoto Cabinet. Kyoto Cabinet (KC) [39] is a database library that stores the database in a single file. KC memory maps the first 64 MB of the file and frequently calls *msync* to ensure that updates to memory-mapped data are persistent. KC also uses write system calls to append new records to the file and uses WAL to provide failure atomicity. We issue sequential SET requests to KC from a single thread for 30 seconds. The key size is 8B and the value size is 1KB.

Fig. 12(c) shows that FusionFS outperforms other file systems with a throughput of 184.6K ops/sec. This is because other file systems iterate over each page within the *msync* range and use a series of flush instructions followed by a memory fence to ensure that the data is flushed to BAS. However, KC synchronizes the entire file even if only a small portion of the file is modified, which introduces unnecessary flushes and memory fences that degrade performance. NOVA shows the second best performance because it uses `generic_file_fsync` without flushing data to BAS. Unlike them, FusionFS uses *in-place updates* for the hot header and data, and batches full-file syncs within a second to reduce the overhead of frequent *msync* calls. In addition, FusionFS's page fault-based profiling mechanism can detect the hot data during *mmap* without any code changes to KC.

5.5.4 LMDB. Lightning Memory-Mapped Database Manager (LMDB) [52] is a B-Tree-based database library. Unlike KC, LMDB memory maps the entire database, so that all data accesses directly load and store the mapped memory region and ensures atomicity with copy-on-write. LMDB also synchronizes the entire file during *msync*, regardless of the mmap write range. We repeat the same workload as KC to evaluate the performance of FusionFS with pure *mmap*. Fig. 12(c) shows that FusionFS outperforms other file systems with a throughput of 352.9K ops/sec. This is because FusionFS uses *in-place updates* for hot data and batches full-file syncs within a second.

5.6 Emulated CXL-SSD Performance

We re-run the Fio workload in Section 5.3 to evaluate the performance of FusionFS on emulated CXL-SSDs. Similar to existing work [3, 40, 42], we emulate CXL-SSDs using Optane PM on a remote NUMA node because they are not yet in mass production and their access latency is comparable to the remote latency on a dual-socket system. Previous studies [59, 65] also indicate that CXL-SSDs have 16KB access granularity. Therefore, when flushing data from CPU caches to the emulated CXL-SSDs, if the flush size is not an integer multiple of the BAS granularity, we adjust it to the nearest larger multiple. We then symbolically modify 1 byte for each cacheline within the aligned region and revert back to ensure that the entire region is marked as dirty and flushed to BAS. Based on previous experiments, FusionFS's cached data is virtually immune to cache contention, so we only consider write amplification caused by explicit flushes and ignore write amplification caused by random cache evictions. To further validate the versatility of FusionFS, we test access granularities of 64B, 256B, and 4KB, corresponding to DRAM, Optane PM, and SSD, respectively, in addition to 16KB.

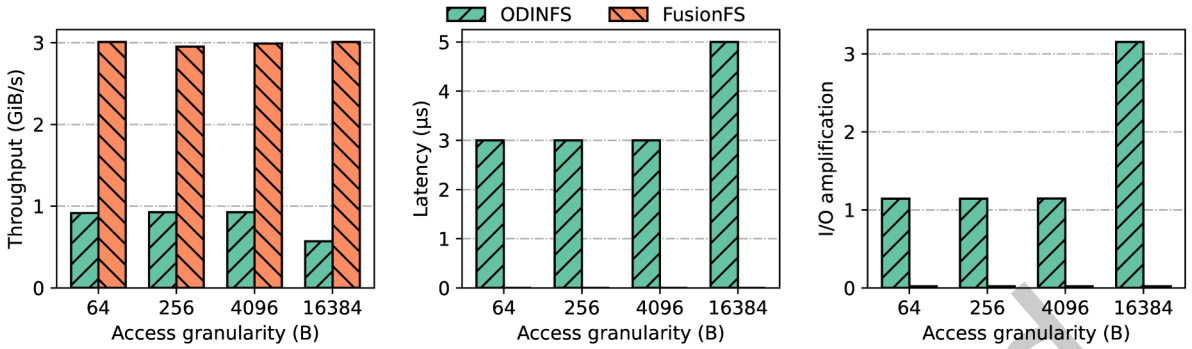

Fig. 13. Throughput, latency, and I/O amplification of evaluated file systems on emulated CXL-SSDs with different access granularities.

Fig. 13 shows that FusionFS achieves optimal performance regardless of access granularity. This is because FusionFS stores hot data in persistent CPU caches, protecting it from cache contention, and writes to the BAS only when the data gets cold. ODINFS, on the other hand, eagerly flushes all data to PM. While this makes it immune to random cache evictions, it will inevitably suffer from write amplification if the access granularity is larger than the write size (e.g., 16KB). For BAS with 64B access granularity (e.g., battery-backed memory), flush instructions can be safely omitted because random cache evictions do not cause write amplification. However, avoiding cache contention still helps reduce data movement between caches and the BAS, thus saving scarce BAS bandwidth. In addition, we find that the increased latency of CXL has minimal impact on kernel file systems that require system calls.

6 Discussion and Related Work

To the best of our knowledge, this is the first work to identify, characterize, and propose solutions to different types of cache contention for BAS systems, and the first to use L3_CAT to mitigate contention for persistent CPU caches and address its challenges. We first discuss the generality of our work and its applicability to other BAS devices, persistent CPU cache implementations, and BAS systems. We then compare FusionFS to related work.

Applicability to other BAS devices and persistent CPU cache implementations. Our implementation of FusionFS is based on Intel Optane PM and eADR, which is a type of BAS and its persistent CPU cache implementation. Although Intel has discontinued its Optane product, research on it is still useful. There are two main reasons for this. First, there is an obvious need for new storage technologies to bridge the gap between DRAM and SSD [7, 69]. Other BAS products such as CXL-SSDs [49, 59] are promising solutions. Second, the development of FusionFS is guided by the general byte-addressability and durability characteristics of BAS, rather than being customized for a specific BAS device. As discussed in Section 5.6, FusionFS achieves good performance across BAS devices with different access granularities and latency characteristics, and avoiding cache contention is still beneficial with matched access granularity.

Our findings can also be applied to other persistent CPU cache implementations. CXL 3.0 [10] introduces the hardware-based Global Persistent Flush (GPF) to provide functionality similar to eADR. Battery-Backed Buffer [1] uses batteries to make caches persistent. These alternatives also suffer from the same cache contention problems as eADR.

Applicability to other BAS systems. The ideas of FusionFS can be applied to other types of BAS systems, such as userspace file systems, key-value stores, and indexes. For example, in userspace file systems (e.g., HTMFS [60]),

we can minimize cache pollution from cold data by choosing *active-flush updates* for cold data. Since flushes cause HTM transactions to abort, we can ensure HTM compatibility by delaying flushes until transactions are complete. In addition, we can use *contention-aware cache allocation* to protect cached data from cache contention.

The Exploration of Persistent CPU Caches. Researchers have been looking for ways to optimize BAS systems with persistent CPU caches. Gugnani [18] proposes lock-free algorithms for linked lists and ring buffers based on atomic CPU hardware primitives. NBTree [62, 64] is a lock-free persistent B+-Tree designed for eADR-enabled platforms. HTMFS [60] and Spash [63] use HTM to simplify concurrency control and provide strong consistency at low cost. Falcon [33] maintains a reusable log window in persistent CPU caches. In terms of cache contention, Falcon [33] and Spash [63] selectively flush cold data to mitigate *BAS-BAS contention*. CacheKV [70] redesigns LSM-Tree's MemTables to optimize its write performance with persistent CPU caches, but it uses unsupported Cache Pseudo-Locking without verifying its effectiveness and therefore cannot mitigate *interfering process contention*. In contrast, FusionFS 1) mitigates various types of cache contention to use persistent CPU caches as a dedicated storage medium, 2) uses L3_CAT to effectively mitigate *interfering process contention* instead of unsupported Cache Pseudo-Locking, 3) optimizes memory mapping accesses through *page fault-based profiling*, and 4) uses range locks for concurrency control instead of relying on HTM, which has security vulnerabilities and is disabled by default [37].

7 Conclusions

Persistent CPU caches can mitigate the shortcomings of BAS in terms of data flushing and access granularity. However, the shared nature of CPU caches can lead to cache contention, negating the benefits of caching and even leading to write amplification. In this paper, we identify, characterize, and propose solutions to persistent CPU cache contention. We also propose FusionFS, a contention-resilient kernel file system that uses persistent CPU caches to redesign data update approaches. FusionFS employs an *adaptive data update* approach that chooses the most effective mechanism based on file access patterns during system calls and memory mapping accesses. FusionFS also employs *contention-aware cache allocation* to mitigate various types of cache contention. Performance results show that FusionFS outperforms existing file systems and effectively mitigates various types of cache contention.

Acknowledgments

This work is supported by National Key Research and Development Program of China (Grant No. 2023YFB4502902), National Natural Science Foundation of China (NSFC) (Grant No. 62332012, 62227809, 62302290), 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 Key Research and Development Program of Xinjiang Uygur Autonomous Region (Grant No. 2023B01027, 2023B01027-1).

References

[1] Mohammad Alshboul, Prakash Ramrakhyani, William Wang, James Tuck, and Yan Solihin. 2021. Bbb: Simplifying persistent programming using battery-backed buffers. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 111–124.

[2] AMD. 2022. AMD64 Technology Platform Quality of Service Extensions. https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/other/56375_1_03_PUB.pdf

[3] Moiz Arif, Kevin Assogba, M Mustafa Rafique, and Sudharshan Vazhkudai. 2022. Exploiting cxl-based memory for distributed deep learning. In *Proceedings of the 51st International Conference on Parallel Processing*. 1–11.

[4] ARM. 2023. Configuring MPAM via resctrl file-system. https://developer.arm.com/documentation/108032/0100/A-closer-look-at-MPAM-software/Linux-MPAM-overview/Configuring-MPAM-via-resctrl-file-system

[5] ARM. 2023. L3 cache partitioning. https://developer.arm.com/documentation/100453/0401/L3-cache/L3-cache-partitioning/

[6] ARM. 2024. Arm DynamIQ shared unit technical reference manual. https://developer.arm.com/documentation/100453/0300/functional-description/l3-cache/cache-stashing

[7] Vinay Banakar, Kan Wu, Yuvraj Patel, Kimberly Keeton, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2023. WiscSort: External Sorting For Byte-Addressable Storage. *arXiv preprint arXiv:2307.06476* (2023).
[8] Samira Briongos, Pedro Malagón, José M Moya, and Thomas Eisenbarth. 2020. {RELOAD+ REFRESH}: Abusing Cache Replacement Policies to Perform Stealthy Cache Attacks. In *29th USENIX Security Symposium (USENIX Security 20)*. 1967–1984.
[9] Congyong Chen, Shengan Zheng, Yuhang Zhang, and Linpeng Huang. 2024. Redesigning Data and Metadata Updates in PM File Systems with Persistent CPU Caches. In *International Conference on Database Systems for Advanced Applications*. Springer, 453–462.
[10] CXL [n. d.]. Compute express link™: The breakthrough cpu-to-device interconnect. https://www.computeexpresslink.org/
[11] Björn Daase, Lars Jonas Bollmeier, Lawrence Benson, and Tilmann Rabl. 2021. Maximizing persistent memory bandwidth utilization for OLAP workloads. In *Proceedings of the 2021 International Conference on Management of Data*. 339–351.
[12] Dave Dice and Alex Kogan. 2019. {BRAVO—Biased} Locking for {Reader-Writer} Locks. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 315–328.
[13] Mingkai Dong, Heng Bu, Jifei Yi, Benchao Dong, and Haibo Chen. 2019. Performance and protection in the ZoFS user-space NVM file system. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 478–493.
[14] Mingkai Dong and Haibo Chen. 2017. Soft updates made simple and fast on non-volatile memory. In *2017 uSENIX Annual Technical Conference (uSENIX ATC 17)*. 719–731.
[15] Subramanya R Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems*. 1–15.
[16] Alireza Farshin, Amir Roozbeh, Gerald Q Maguire Jr, and Dejan Kostić. 2019. Make the most out of last level cache in intel processors. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–17.
[17] Fio [n. d.]. FIO (Flexible I/O Tester). https://github.com/axboe/fio
[18] Shashank Gugnani, Arjun Kashyap, and Xiaoyi Lu. 2020. Understanding the idiosyncrasies of real persistent memory. *Proceedings of the VLDB Endowment* 14, 4 (2020), 626–639.
[19] Kewen He, Yujie An, Yijing Luo, Xiaoguang Liu, and Gang Wang. 2023. FlatLSM: Write-Optimized LSM-Tree for PM-Based KV Stores. *ACM Transactions on Storage* 19, 2 (2023), 1–26.
[20] Intel. 2015. Improving Real-Time Performance by Utilizing Cache Allocation Technology. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/cache-allocation-technology-white-paper.pdf
[21] Intel. 2016. Deprecating the pcommit instruction. https://www.intel.com/content/www/us/en/developer/articles/technical/deprecate-pcommit-instruction.html
[22] Intel. 2016. Pseudo-locking on Skylake processors. https://github.com/intel/intel-cmt-cat/issues/215
[23] Intel. 2016. README for PSEUDO LOCK Sample Code. https://github.com/intel/intel-cmt-cat/blob/master/examples/c/PSEUDO_LOCK/README#L39
[24] Intel. 2016. User Interface for Resource Control feature. https://www.kernel.org/doc/html/v5.9/x86/resctrl_ui.html#cache-pseudo-locking
[25] Intel. 2016. User Interface for Resource Control feature. https://www.kernel.org/doc/html/v5.9/x86/resctrl_ui.html
[26] Intel. 2020. How to Offload Compute-Intensive Code to Intel®GPUs. https://www.intel.com/content/www/us/en/developer/articles/technical/dpcpp-for-intel-processor-graphics-architecture.html
[27] Intel. 2021. eADR: New Opportunities for Persistent Memory Applications. https://www.intel.com/content/www/us/en/developer/articles/technical/eadr-new-opportunities-for-persistent-memory-applications.html
[28] Intel. 2021. Intel®optane™persistent memory 200 series brief. https://www.intel.la/content/www/xl/es/products/docs/memory-storage/optane-persistent-memory/optane-persistent-memory-200-series-brief.html
[29] Intel. 2021. Intel®Xeon®Gold 6348 Processor (42M Cache, 2.60 GHz) Specifications. https://www.intel.com/content/www/us/en/products/sku/212456/intel-xeon-gold-6348-processor-42m-cache-2-60-ghz/specifications.html
[30] Intel. 2021. Storage Performance Development Kit. https://spdk.io/
[31] Intel. 2024. Intel-cmt-cat. https://github.com/intel/intel-cmt-cat
[32] Intel. 2024. Intel®Data Direct I/O Technology. https://www.intel.com/content/www/us/en/io/data-direct-i-o-technology.html
[33] Zhicheng Ji, Kang Chen, Leping Wang, Mingxing Zhang, and Yongwei Wu. 2023. Falcon: Fast OLTP Engine for Persistent Cache and Non-Volatile Memory. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 531–544.
[34] Rohan Kadekodi, Saurabh Kadekodi, Soujanya Ponnapalli, Harshad Shirwadkar, Gregory R Ganger, Aasheesh Kolli, and Vijay Chidambaram. 2021. WineFS: a hugepage-aware file system for persistent memory that ages gracefully. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 804–818.
[35] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. 2019. SplitFS: Reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 494–508.
[36] Anuj Kalia, David Andersen, and Michael Kaminsky. 2020. Challenges and solutions for fast remote persistent memory access. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. 105–119.

[37] The kernel development community. 2019. TAA - TSX Asynchronous Abort. https://docs.kernel.org/admin-guide/hw-vuln/tsx_async_abort.html
[38] Wook-Hee Kim, R Madhava Krishnan, Xinwei Fu, Sanidhya Kashyap, and Changwoo Min. 2021. Pactree: A high performance persistent range index using pac guidelines. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 424–439.
[39] FAL Labs. 2010. Kyoto Cabinet: a straightforward implementation of DBM. http://fallabs.com/kyotocabinet/
[40] Huaicheng Li, Daniel S Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, et al. 2023. Pond: Cxl-based memory pooling systems for cloud platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 574–587.
[41] Jiahao Li, Jingbo Su, Luofan Chen, Cheng Li, Kai Zhang, Liang Yang, Sam Noh, and Yinlong Xu. 2024. Fastmove: A Comprehensive Study of On-Chip DMA and its Demonstration for Accelerating Data Movement in NVM-based Storage Systems. *ACM Transactions on Storage* (2024).
[42] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. 2023. Tpp: Transparent page placement for cxl-enabled tiered-memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 742–755.
[43] Changwoo Min, Sanidhya Kashyap, Steffen Maass, and Taesoo Kim. 2016. Understanding manycore scalability of file systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. 71–85.
[44] Babu Moger. 2023. x86/resctrl : Support AMD QoS RMID Pinning feature. https://lore.kernel.org/lkml/90bb280e-402a-a3e9-2db4-c08f3762fdb1@amd.com/T/
[45] Jiaxin Ou, Jiwu Shu, and Youyou Lu. 2016. A high performance file system for non-volatile main memory. In *Proceedings of the Eleventh European Conference on Computer Systems*. 1–16.
[46] PMDK. [n. d.]. libpmem. https://pmem.io/pmdk/libpmem/
[47] PMDK. 2024. Direct Write to PMem. https://pmem.io/rpma/documentation/basic-direct-write-to-pmem/
[48] T. L. F. Projects. 2021. DPDK. https://www.dpdk.org/
[49] Samsung. [n. d.]. MS-SSD. https://samsungmsl.com/ms-ssd/
[50] Parul Sohal, Michael Bechtel, Renato Mancuso, Heechul Yun, and Orran Krieger. 2022. A closer look at intel resource director technology (rdt). In *Proceedings of the 30th International Conference on Real-Time Networks and Systems*. 127–139.
[51] SQLite. [n. d.]. SQLite transactional SQL database engine. https://www.sqlite.org/
[52] Symas. 2017. Lightning Memory-Mapped Database (LMDB). https://symas.com/lmdb/
[53] Erez Zadok Vasily Tarasov and Spencer Shepler. 2016. Filebench: A flexible framework for file system benchmarking. *USENIX; login* 41, 1 (2016), 6–12.
[54] Xingda Wei, Xiating Xie, Rong Chen, Haibo Chen, and Binyu Zang. 2021. Characterizing and optimizing remote persistent memory with {RDMA} and {NVM}. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 523–536.
[55] Hobin Woo, Daegyu Han, Seungjoon Ha, Sam H Noh, and Beomseok Nam. 2023. On stacking a persistent memory file system on legacy file systems. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. 281–296.
[56] Xiaojian Wu and AL Narasimha Reddy. 2011. SCMFS: A file system for storage class memory. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–11.
[57] Jian Xu and Steven Swanson. 2016. {NOVA}: A log-structured file system for hybrid {Volatile/Non-volatile} main memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*. 323–338.
[58] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. 2020. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. 169–182.
[59] Shao-Peng Yang, Minjae Kim, Sanghyun Nam, Juhyung Park, Jin-Yong Choi, Eyee Hyun Nam, Eunji Lee, Sungjin Lee, and Bryan S Kim. 2023. Overcoming the Memory Wall with {CXL-Enabled}{SSDs}. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 601–617.
[60] Jifei Yi, Mingkai Dong, Fangnuo Wu, and Haibo Chen. 2022. {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)*. 17–34.
[61] Yifan Yuan, Mohammad Alian, Yipeng Wang, Ren Wang, Ilia Kurakin, Charlie Tai, and Nam Sung Kim. 2021. Don't forget the I/O when allocating your LLC. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 112–125.
[62] Bowen Zhang, Shengan Zheng, Liangxu Nie, Zhenlin Qi, Hongyi Chen, Linpeng Huang, and Hong Mei. 2024. Revisiting PM-Based B$^+$-Tree With Persistent CPU Cache. *IEEE Transactions on Parallel and Distributed Systems* 35, 5 (2024), 796–813. doi:10.1109/TPDS.2024.3372621
[63] Bowen Zhang, Shengan Zheng, Liangxu Nie, Zhenlin Qi, Linpeng Huang, and Hong Mei. 2024. Exploiting Persistent CPU Cache for Scalable Persistent Hash Index. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 3851–3864.
[64] Bowen Zhang, Shengan Zheng, Zhenlin Qi, and Linpeng Huang. 2022. NBTree: a Lock-free PM-friendly Persistent B+-Tree for eADR-enabled PM Systems. *Proceedings of the VLDB Endowment* 15, 6 (2022), 1187–1200.

[65] Rui Zhang, Yukai Huang, Sicheng Liang, Shangyi Sun, Shaonan Ma, Chengying Huan, Lulu Chen, Zhihui Lu, Yang Xu, Ming Yan, et al. 2024. Revisiting Learned Index with Byte-addressable Persistent Storage. In *Proceedings of the 53rd International Conference on Parallel Processing*. 929–938.
[66] Da Zheng, Randal Burns, and Alexander S Szalay. 2013. Toward millions of file system IOPS on low-cost, commodity hardware. In *Proceedings of the international conference on high performance computing, networking, storage and analysis*. 1–12.
[67] Shengan Zheng, Morteza Hoseinzadeh, and Steven Swanson. 2019. Ziggurat: A tiered file system for {Non-Volatile} main memories and disks. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. 207–219.
[68] Shengan Zheng, Morteza Hoseinzadeh, Steven Swanson, and Linpeng Huang. 2023. TPFS: A High-Performance Tiered File System for Persistent Memories and Disks. *ACM Transactions on Storage* 19, 2 (2023), 1–28.
[69] Ying Zheng and Kian-Lee Tan. 2024. Sorting on Byte-Addressable Storage: The Resurgence of Tree Structure. *Proceedings of the VLDB Endowment* 17, 6 (2024), 1487–1500.
[70] Yijie Zhong, Zhirong Shen, Zixiang Yu, and Jiwu Shu. 2023. Redesigning High-Performance LSM-based Key-Value Stores with Persistent CPU Caches. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 1098–1111.
[71] Diyu Zhou, Yuchen Qian, Vishal Gupta, Zhifei Yang, Changwoo Min, and Sanidhya Kashyap. 2022. {ODINFS}: Scaling {PM} performance with opportunistic delegation. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 179–193.
[72] Bohong Zhu, Youmin Chen, and Jiwu Shu. 2024. Exploring the Asynchrony of Slow Memory Filesystem with EasyIO. In *Proceedings of the Nineteenth European Conference on Computer Systems*. 624–640.

Received 28 August 2024; revised 29 December 2024; accepted 6 February 2025