Cache or Direct Access? Revitalizing Cache in Heterogeneous Memory File System

Yubo Liu, Yuxin Ren, Mingrui Liu, Hanjun Guo, Xie Miao, Xinwei Hu
Huawei, China

Abstract

This paper revisits the value of cache in DRAM-PM heterogeneous memory file systems. The first contribution is a comprehensive analysis of the existing file systems on heterogeneous memory, including cache-based and DAX-based (direct access). We find that the DRAM cache still plays an important role in heterogeneous memory. The second contribution is a cache framework for heterogeneous memory, called FLAC. FLAC integrates the cache with the virtual memory management and proposes two technologies of zero-copy caching and parallel-optimized cache management, which deliver the benefits of fast application-storage data transfer and efficient DRAM-PM data synchronization/migration. We further implement a library file system upon FLAC. Microbenchmarks show that FLAC provides a performance increase of up to two orders of magnitude over existing file systems in file `read/write`. With a real-world application, FLAC achieves up to 77.4% and 89.3% better performance than NOVA and EXT4, respectively.

CCS Concepts: • **Software and its engineering** → **File systems management**; • **Information systems** → *Storage class memory*.

Keywords: Heterogeneous Memory, File Systems, Page Cache

ACM Reference Format:
Yubo Liu, Yuxin Ren, Mingrui Liu, Hanjun Guo, Xie Miao, Xinwei Hu. 2023. Cache or Direct Access? Revitalizing Cache in Heterogeneous Memory File System. In *1st Workshop on Disruptive Memory Systems (DIMES '23), October 23, 2023, Koblenz, Germany*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3609308.3625272

1 Introduction

New persistent memory techniques (*e.g.*, 3DXPoint [12, 20] and CXL-based SSD [15]) and connection techniques (*e.g.*, CXL [4] and GenZ [5]) promise high performance, larger capacity, and energy efficiency. These bring a new trend of heterogeneous memory architecture which consists of a volatile memory layer (DRAM) and a persistent memory layer (PM, it can be the 3DXPoint or emerging CXL-based memory storage devices). This work investigates an important question: What kind of storage framework can maximize the potential of heterogeneous memory? Currently, using DRAM as cache and direct access (DAX) are two mainstream solutions for heterogeneous memory file systems.

Caching pages in DRAM, such as VFS page cache, is a common design in traditional file systems (*e.g.*, EXT4 and XFS) to bridge the performance gap between fast DRAM and slow persistent storage devices (*e.g.*, HDD and SSD). However, many previous studies [8] in the past decade argue that the DRAM cache incurs significant software overhead under the fast, full-memory architecture. Therefore, most existing systems (*e.g.*, NOVA [31], SplitFS [17], ctFS [21], and KucoFS [2]) resort to the DAX method, which bypasses the DRAM cache and performs I/Os on PM directly.

However, DAX is still suboptimal for heterogeneous memory file systems. First, the DAX method potentially loses the performance benefit of data locality provided by the DRAM cache and incurs mandatory data copy across DRAM and PM. Even worse, the performance upper bound of the DAX is limited by the PM hardware which is inevitably slower than DRAM. As we show in §2, the performance of DAX-based systems is inferior to that of cache-based systems in scenarios with high concurrency and strong data locality, even though the VFS page cache framework introduces high software overheads. Last but not least, instant persistence is the best scene of DAX; but it is overkill in many real-world scenarios [30].

Moreover, new characteristics in emerging hardware and customer demand motivate us to revisit the value of DRAM cache in heterogeneous memory architecture. **1)** The performance gap between PM and DRAM cannot be ignored, and multiple PMs will have different performances in the future (latency ranges from 170ns to 3000ns [14, 15]). **2)** Data locality exists in various real scenarios, and keeping the hot data in DRAM can undoubtedly improve I/O performance. **3)** Instant persistence in DAX is overkill in many real-world scenarios [30].

While the DRAM cache still plays an important role in the future heterogeneous memory architecture, simply reusing the current implementation, such as the VFS page cache, is insufficient. According to our quantitative analysis (§2),

DIMES '23, October 23, 2023, Koblenz, Germany

ACM ISBN 979-8-4007-0300-3/23/10…$15.00
https://doi.org/10.1145/3609308.3625272

we conclude two challenges of building an efficient cache framework on heterogeneous memory:

1) Reduce the data transfer overhead between application buffer and cache. Transferring data between the application and cache is the most critical fast-path operation, but existing cache frameworks use memory copy that introduces substantial performance overhead. Our experiments show that data copying occupies up to 84% of the overhead in the file system with the VFS page cache.

2) Reduce the impact of the "cache tax". In addition to data transfer, existing cache frameworks spend lots of effort to synchronize (flushing dirty data) and migrate data (evicting data into/out of cache) across DRAM cache and PM. Currently, these operations are implemented in a synchronous and serial way and significantly increase performance penalty (more than 30%). For instance, upon a cache miss, current systems block I/O operation and fetch data from lower-level storage synchronously.

This paper proposes FLAC (**FLA**t **C**ache), a novel cache framework for heterogeneous memory. The key idea of FLAC is to integrate the cache with the virtual memory management subsystem. FLAC provides a single-level view of heterogeneous memory and enables a transparent and efficient DRAM cache in the data I/O path. FLAC leverages two novel techniques to deal with the two challenges outlined above:

1) Zero-Copy Caching. FLAC proposes the heterogeneous page table that unifies heterogeneous memory into a single level. Virtual pages within FLAC can be dynamically mapped to physical pages on DRAM or PM according to their states (*i.e.* cached or evicted). We then design the page attaching mechanism, a set of tightly coupled management operations on the heterogeneous page table, which optimize the data transfer between applications and cache in a zero-copy manner. The core idea of page attaching is to map pages between source and destination addresses with enforced copy on write (COW). As a result, data read/write to/from FLAC is executed by page attaching to realize efficient and safe data transfer. While page remapping optimizations are also used in some systems to reduce the overhead of data copy [7, 10, 24, 25], using a similar idea in the file system cache faces some unique challenges that will be discussed in §3.1.

2) Parallel-Optimized Cache Management. The cache management mechanism of FLAC must ensure a low "cache tax" impact. Leveraging the multi-version feature that is brought by the zero-copy caching, FLAC can fully exploit the parallelism of data synchronization and migration with critical I/O paths. FLAC proposes a 2-Phase flushing mechanism that allows the expensive persistence phase in dirty data synchronization to be lock-free, and proposes asynchronous cache miss handling to amortize the overhead of loading data to cache in the background.

Furthermore, we leverage FLAC to implement a prototype of file system `read` and `write` operations. The evaluation shows that FLAC provides a maximum performance increase of more than an order of magnitude over existing systems. With a command line application (`tar`), FLAC outperforms NOVA and EXT4 by up to 77.4% and 89.3%, respectively.

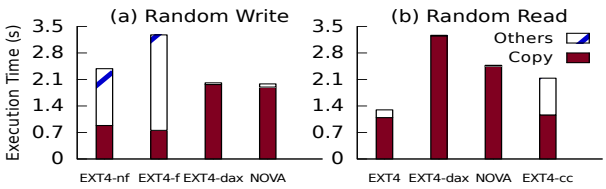

Figure 1. Traditional Cache vs. DAX (Lower is better). "-f/-nf": with/without background flushing; "-cc": cold cache.

2 Cache? or DAX?

With the emergence of new interconnection technologies (*e.g.*, CXL [4], GenZ [5]) and persistent storage medias (*e.g.*, 3DXPoint [12], CXL-based PM/SSD [15]), the storage architecture evolves from memory-block to all-memory. A typical heterogeneous memory architecture consists of a fast, volatile, small capacity layer (DRAM), and a slow, non-volatile, large capacity layer (PM). Different types of memories present heterogeneity in multiple aspects. The latency of DRAM is about 80ns to 140ns, while the latency of low tier memory ranges from 170ns to 3000ns [14, 15]. The PM layer also has lower bandwidth and concurrency than the DRAM layer [9, 17].

We deeply analyze the overhead of three typical file systems with cache (EXT4) and DAX (EXT4-DAX, NOVA) and discuss the way to efficiently utilize heterogeneous memory. We use a single test thread to randomly `write`/`read` a 10GB file with 2MB I/O (the testbed is introduced in §5) and come up with three observations from the experiments.

Observation 1*: Existing DAX and cache frameworks are sub-optimal, but DRAM cache still has great value for heterogeneous memory file systems.*

The VFS page cache is a typical cache framework that is designed to bridge the performance gap between DRAM and block devices. However, the VFS page cache has a heavy software stack, which makes it unsuitable for the heterogeneous memory structure. Therefore, many heterogeneous memory file systems proposed in the past decade resort to the DAX method, *i.e.*, bypassing the DRAM cache in the data I/O path. However, we think DRAM cache still has a lot of value in heterogeneous memory file systems. First, PMs with different performances will emerge in the future, and the performance gap between PM and DRAM cannot be ignored. Figure 1 and 3 show that VFS page cache still has better performance than DAX in some cases (*e.g.*, read and concurrent write). Second, taking advantage of data locality is still the main method of performance optimization. Third, POSIX is still a mainstream semantics and it can tolerate cached I/Os, which makes instant persistence in DAX an overkill in many real-world scenarios [30].

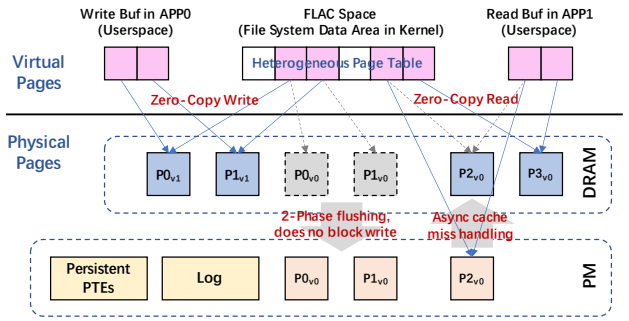

Figure 2. Architecture of FLAC. File data management is run on top of FLAC. The file read/write operation is converted to the zero-copy transfer API of FLAC. The pages are flushed to/loaded from the PM by the parallel-optimized mechanism.

***Observation 2**: Data transfer overhead between the file system and application buffer is significant but often overlooked, and it is one of the keys to unlocking the potential of the cache in heterogeneous memory.*

File data I/Os (`read`/`write`) need to transfer data between the application buffer and the storage system (*i.e.* cache or persistent data area in DAX). Memory copy is the mainstream method to transfer data and it takes up more than 23% and 96% of the total overhead in cache-based and DAX-based file systems, respectively (Figure 1). Since the latency of PM is much smaller than block devices, the performance bottleneck of data copy between cache and application is more obvious. This observation leads us to consider a zero-copy approach to transfer data between the cache and application buffer. Thanks to the byte-addressability, it is feasible to build a single-level address space of heterogeneous memory to achieve zero-copy.

***Observation 3**: "Cache Tax" in traditional cache frameworks is heavy, and it mainly includes the overhead of data synchronization and migration.*

Caching increases storage levels and brings extra data management overhead. Figure 1 shows that the "cache tax" (denoted as "other") takes up to 77% of the execution time in EXT4. Furthermore, the experiments reveal the composition of the "cache tax". The background dirty flushing (data synchronization) and cache miss (data migration) lead to 37% and 65% performance declines, respectively. In general, the "cache tax" is difficult to eliminate, but we can reduce its impact on the critical I/O paths by improving the parallelism between them and front-end I/Os.

3 Flat Cache

We propose FLAC, a **FLA**t **C**ache framework integrated with the virtual memory subsystem to deeply explore the potential of cache in heterogeneous memory. FLAC is designed for any heterogeneous memory architecture, while persistent memory can be existing (*e.g.*, 3DXPoint) or future (*e.g.*, CXL-based PM/SSD) devices.

Table 1. Main APIs of FLAC (for file system developer)

API	Main Para.	Description
`init_flac`	pm_path	Create/Recover the FLAC space
`zcopy_from_flac` `zcopy_to_flac`	from_addr to_addr size	Transfer data between application buffer and FLAC with the zero-copy approach
`pflush_add`	pflush_handle addr size	Attach (map) pages to a flushing buffer and add to the handle
`pflush_commit`	pflush_handle	Atomically flush dirty pages to PM
`pfree`	addr size	Atomically reclaim PM pages

As shown in Figure 2, FLAC maintains a range of contiguous virtual memory addresses, called FLAC space, which is as large as the available PM space used to store file data. FLAC space is indexed by the heterogeneous page table and it makes the physical location of the page transparent and exposes a single-level memory space to file system developers. Pages are cached in DRAM when they are accessed (the cache size can be adjusted). Data are transferred between the application and FLAC space with the zero-copy approach (§3.1) and synchronized/migrated between DRAM and PM with the parallel-optimized mechanism (§3.2).

FLAC is a development framework for heterogeneous memory architecture that allows the file system developers to customize the data management (*e.g.*, read/write logic) on FLAC space. The other modules of the file system (*e.g.*, metadata management) are independent of FLAC and they can be flexibly implemented. Table 1 shows the APIs of FLAC. The file system developers initialize FLAC by calling `init_flac`, which creates FLAC space and binds the PM to it. The file system internally uses `zcopy_to/from_flac` to transfer data and support `read` and `write` operations. Data accesses on FLAC space are transparent to applications, so applications use standard `read` and `write` operations to access files. The file system developers are asked to explicitly flush dirty data from DRAM to PM, which gives them the flexibility to customize flushing policies. A pair of APIs, `pflush_add` and `pflush_commit`, provide a high concurrency and atomic manner to flush data. At the same time, FLAC space can be atomically reclaimed by calling `pfree`.

3.1 Zero-Copy Caching

Heterogeneous page table. As Figure 2 shows, FLAC uses the heterogeneous page table, a customized sub-level table (*e.g.*, one or multiple PUDs) of the kernel page table, to maintain FLAC space. The heterogeneity of the page table has two meanings. **1)** Page table entries (PTEs) belonging to the page table are replicated in PM for fault recovery. **2)** The address indexed in the page table is dynamically mapped to DRAM

or PM as the page is cached or evicted, and a bit in the PTE is used to indicate the location of the page. The heterogeneous page table unifies the page indexes of cache and persistent storage and simplifies cache access and management.

Page attaching. The core technique of optimizing the data transfer in FLAC is a new mechanism for virtual memory management – page attaching. Given a piece of data, the page attaching maps its source address (in either DRAM cache or PM) to the destination address in the application buffer without copying the data. Page attaching allows users to set permissions (*e.g.*, read-only) on source and destination addresses after remapping. `Read/write` operations can then be implemented using page attach between the cache and application buffer without copying. FLAC guarantees secure data mapping and sharing under concurrent accesses by enforcing read-only mapping and copy-on-write (COW).

In contrast to existing works that utilize page remapping to realize zero-copy, as far as we know, FLAC is the first work to use attaching to optimize data transfer between application and file system cache. Furthermore, there are some unique challenges in the file system cache to employ page remapping. First, the remapping-based data transfer makes the page have multiple versions, which needs to work with a new cache management mechanism to ensure data consistency and high concurrency (detailed in §3.2). Second, FLAC uses COW page fault to ensure security and isolation when pages are attached from/to FLAC space, which may bring extra overhead in some workloads. Third, zero-copy introduces the limitation of page alignment, which may reduce the applicability of the file system implemented based on FLAC. The second and third challenges are discussed in §6.

3.2 Parallel-Optimized Cache Management

FLAC requires a cache management mechanism for its multi-version feature, while ensuring low "cache tax" impact. Existing cache frameworks execute cache flushing and cache miss handling with large synchronization overhead: Cache flushing locks the dirty pages until they are completely flushed, which blocks the foreground writes and reduces the performance; cache miss handling blocks the foreground I/O until the pages are loaded to DRAM cache. The multi-version feature and the heterogeneous page table design of FLAC give us an opportunity to fully exploit the parallelism of these operations with critical I/O paths. Figure 2 shows the examples of cache flushing and cache miss handling in FLAC.

2-Phase flushing. FLAC splits the dirty pages flushing into two phases: collection phase (`pflush_add`) and persistence phase (`pflush_commit`). The collection phase allocates a fresh VA area as a temporary flush buffer and attaches the dirty pages to it. This phase requires a lock to prevent concurrent writes from modifying the mapping. Then, the persistence phase stores the dirty pages to PM, which is lock-free since there are no concurrent accesses to the temporary buffer. Because the page mapping in the collection phase is much faster than cross-memory copy, FLAC significantly reduces the blocking time on foreground writes. In addition, the persistence phase is atomic – the consistency of pages and persistent PTEs are protected by log-structured flushing and logging, respectively.

Asynchronous cache miss handling. Cache miss has less impact on write operation because it does not require pages to be loaded into the cache (except in the case of page misalignment). Benefiting from the heterogeneous page table, FLAC can directly attach the PM pages to the read buffer (and returns immediately) and handle the cache miss asynchronously. A background thread in FLAC is responsible for loading the cache missed pages to DRAM and remapping page tables pointing to those PM pages to the cached pages on DRAM. As a result, the overhead of handling cache misses can be amortized in the background.

4 Case Study: File Read/Write on FLAC

We introduce a simple library file system based on FLAC to show its usage and benefits. As FLAC focuses on optimizing data I/O, the prototype only implements `read/write` and a few necessary metadata operations (*e.g.*, open and close).

Architecture. File data are stored in the FLAC space to get benefits from the flat cache design, while metadata are stored in the hash table on the normal shared memory space. The inode table is stored on both the DRAM and PM. The prototype borrows the page index design from ctFS [21], *i.e.*, it allocates a segment of consecutive VAs (virtual memory address) on FLAC space for each file, and the start VA and size of the file are recorded in the corresponding metadata.

Read and Write. The `read/write` interface is similar to that in the traditional POSIX file. The main parameters include file handle, file offset, access length, and read/write buffer provided by the application. Currently, our prototype assumes that the file offset and read/write buffer are 4KB page aligned (the applicability is discussed in §6). After opening the file, `read/write` gets the start VA on FLAC space from the file's metadata and performs `zcopy_from_flac` (read) / `zcopy_to_flac` (write) to transfer data. In addition, the file system launches a background thread to periodically flush the dirty pages to PM by using the 2-Phase flushing mechanism (by calling `pflush_add` and `pflush_commit`).

Compared to Related Works. FLAC allows file systems based on it to benefit from the DRAM cache while reducing the effects of "cache tax" as much as possible We compare FLAC to a wide range of existing works (shown in Table 2).

vs. Cache-based File Systems/mmap. There are many file systems designed based on VFS. Although VFS page cache can improve the performance in some scenarios in heterogeneous memory file systems, these systems suffer from heavy "cache tax" and fail to optimize the application-FS data transfer. In addition, some existing works optimize

Table 2. Comparison with Related Works

Type	Typical System	Data Cache	Low/Non Cache Tax Impact	App-Storage Zero-Copy	App-Storage Decouple
Cache-based FS	VFS page cache FSes (ETX4, SPFS [30], *etc.*)	✔	✗	✗	✔
Cache-based mmap	mmap in VFS page cache FSes (EXT4, *etc.*)	✔	✗	✔	✗
DAX-based FS	NOVA [31], Strata [19], SplitFS [17], WineFS [16], ctFS [21], KucoFS [2], PMFS [8], libnvmmio [3], EXT4-DAX [6], XFS-DAX, HTMFS [32]	✗	✔	✗	✔
DAX-based Runtime	Twizzler [1], Mnemosyne [28], PMDK [13] zIO [27], SubZero [18]	✗	✔	✔	✗
Flat Cache	FLAC-based FS	✔	✔	✔	✔

DRAM page cache [22, 23] for PM, but they are built on top of the virtual memory subsystem and therefore fail to exploit the full potential of cache. Although some cache-based file systems also provide the `mmap` method to avoid the data transfer overhead, it makes application design and storage backend to be coupled, which is complementary to the file semantics.

vs. DAX-based File Systems. DAX-based file systems bypass the DRAM cache in data I/O, making them suffer from high application-storage transfer overhead. Also, the latency and concurrency of PM hardware greatly limit their performance. In particular, some DAX-based file systems also use remapping: SplitFS [17] proposes relink, an operation to atomically move a contiguous extent from one file to another, which is used to accelerate appends and atomic data operations; ctFS [21] proposes pswap to swap the page mapping of two same-sized contiguous virtual addresses, which is used to reduce the overhead of maintaining file data in contiguous virtual addresses. However, neither SplitFS nor ctFS uses remapping to optimize data copying between applications and file systems, and FLAC optimizes this part with the zero-copy caching technique.

vs. DAX-based Runtime. This type of work usually provides a memory management library or programming framework for applications. Although the overhead of data transfer between the application and storage system can be avoided, they require the application to be co-designed with the storage backend (*e.g.*, use customized interfaces or object abstraction). Some of these works provide zero-copy PM I/O libraries [18, 27]. However, they require applications to allocate read/write buffers on PM to avoid data copy, and thus force to ship the data processing from DRAM to PM, which is not friendly for some cases [29]. DAX-based runtime focuses on programming directly on PM and can be seen as complementary to the file system.

5 Preliminary Results

We compare the `read/write` performance in FLAC with cache-based (FLAC and EXT4) and DAX-based (EXT4-DAX and NOVA) file systems. FLAC and EXT4 are run on a hot cache. The period of background flushing is 10ms in FLAC while it is 100ms in EXT4. The experiments are run on a server with two Intel Xeon Platinum 8380 CPU @ 2.30GHz, 256GB RAM, and 1TB (128GB×8) Intel 3DXPoint DCPMM. Porting FLAC to more complicated applications and comparing with more recent file systems, such as ctFS [21], are ongoing work.

Microbenchmark. The benchmark uses 2MB I/O to concurrently and randomly write/read 64GB data on 64 non-empty files (1GB per file), and no data contention in the experiments. As Figure 3 (a) and (b) show, FLAC outperforms other tested systems by more than 25.9 times and 13.7 times in write and read, respectively. The zero-copy design of FLAC contributes to the main performance gain. In addition, data copy during background flushing does not block foreground writes, which significantly reduces the impact of background flushing on performance in write-intensive scenarios.

Real-World Application. We port `tar` (v1.34) to FLAC, a commonly used archiving application. Its main process reads the input file, archives it, and writes the data to an output file. The `tar` contains little computation and represents an I/O-intensive case. Figure 3 (c) plots the execution time of archiving a file with increasing file size. On average, FLAC improves NOVA, EXT4, and EXT4-DAX by 48.5%, 54%, and 41.2%, respectively.

6 Discussion and Future Work

Although FLAC promises attractive performance improvement in data I/Os, it still leaves some limitations and open challenges for our feature work.

1) Reduce COW page fault overhead. Data transfer between FLAC space and application buffer is implemented by page attaching. After attaching, the source and destination memory are set to read-only for security, which makes the first `store` instruction to the source (write case) and destination (read case) memory after attaching to trigger a COW page fault. Our analysis shows that TLB flushing and data copy are two of the main overheads in the COW page fault. We have two ideas to reduce the COW page fault overhead: The first idea is to flush TLB in batch (the default is once per page). The second idea is to provide a new interface that allows the application to detach the original page mappings, thus completely avoiding COW page fault.

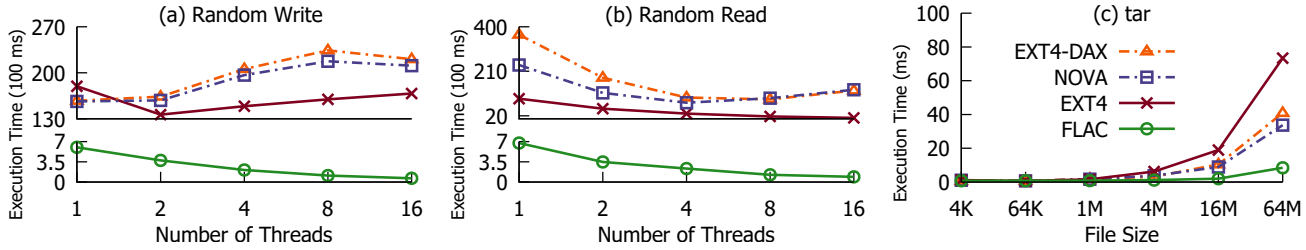

Figure 3. Microbenchmark and Application Performance (Lower is better).

2) Improve the applicability. Zero-copy (page attaching) can be performed only when the application buffer and the target memory in FLAC are aligned. Currently, FLAC asks applications to perform file accesses following this rule. For example, one way to adapt an application to FLAC is to ensure that the read/write buffer and target file offset are 4KB page aligned. To improve the applicability of FLAC, a possible solution is to provide a customized buffer management mechanism for applications. It allocates a buffer larger (and aligned) than the required the file I/O size and always perform file access as page aligned. In this way, the buffer may contain more data than the application needs, so the mechanism maintains a sliding window to represent the valid data in the buffer, and the application calls an explicit interface to move the window after each file I/O.

3) FLAC-optimized cache policy. FLAC permits more powerful cache policies for heterogeneous memory. First, because FLAC is embedded in the VM subsystem, it can be aware of more memory access behavior about the applications (*e.g.*, allocation/free, reference count), which makes it possible for FLAC to make better caching decisions. Second, the lower layer of heterogeneous memory is fast and byte-addressable, thus FLAC can investigate more trade-offs between data locality and miss ratio. In future work, we aim to collaborate these new insights brought by FLAC with traditional hotness-based methods to design an efficient cache policy.

4) Ensure security. For data security, FLAC is implemented in the kernel, and userspace applications can use it only through syscalls. Pages are always mapped to the application as read-only, which ensures that local operations of the application do not affect the data in the cache and other applications as they are handled by COW page fault. Implementing a storage system on top of FLAC brings security considerations for metadata, which can be solved by using the userspace security mechanisms (*e.g.*, MPK [11, 26]) or putting metadata management in the kernel.

5) Ensure crash consistency. FLAC ensures that data modification operations (`pflush_commit` and `pfree`) are atomic. However, along with flushing the data, the file system upon FLAC may need to update the related FS-level metadata (*e.g.*, page index) on PM. We plan to design a FS-FLAC collaboration logging mechanism, which ensures that data flushing and FS-level metadata updates are in a transaction. The basic idea is to allow the file system to provide the updated FS-level metadata to FLAC, and they are logged with updated FLAC-level metadata in the same entry during data modification operations. The file system is also required to overload the FS-level recovery function that FLAC calls to commit the FS-level metadata log during recovery.

6) Space overhead. Compared to traditional page cache, the multi-version design of FLAC does not incur additional space overhead. In FLAC, the new version of the page is created in the application runtime by COW page fault, so the new version does not take up space in the page cache before it is overwritten to FLAC. After overwriting, the virtual address in the FLAC space is mapped to the new version, and the old version is reclaimed. Furthermore, the zero-copy caching design naturally brings the deduplication benefit in some cases (*e.g.*, the application reuses the write buffer and only a small number of pages have been modified). We plan to leverage this advantage to improve the space efficiency of the DRAM cache and PM.

7 Conclusion

Heterogeneous memory requires innovations of effective software architecture to maximize its potential of various advantages. We analyze the shortcomings of existing cache-based and DAX-based file systems, and conclude that DRAM cache still has great potential in fast all-memory architectures. We propose FLAC, a flat cache framework for heterogeneous memory that embeds the cache into virtual memory management. FLAC unlocks the potential of cache through two new techniques: zero-copy caching and parallel-optimized cache management. We implement a file system prototype based on FLAC and show that FLAC has significantly better performance than existing cache and DAX solutions.

Acknowledgments

We thank the anonymous reviewers for their constructive comments and feedback. We also thank our colleagues in the Huawei OS Kernel Lab for their help.

References

[1] Daniel Bittman, Peter Alvaro, Pankaj Mehra, Darrell D. E. Long, and Ethan L. Miller. 2020. Twizzler: A Data-Centric OS for Non-Volatile Memory. In *Proceedings of the USENIX Annual Technical Conference (ATC'20)*.

[2] Youmin Chen, Youyou Lu, Bohong Zhu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Jiwu Shu. 2021. Scalable Persistent Memory File System with Kernel-Userspace Collaboration. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST'21)*.

[3] Jungsik Choi, Jaewan Hong, Youngjin Kwon, and Hwansoo Han. 2020. Libnvmmio: Reconstructing Software IO Path with Failure-Atomic Memory-Mapped Interface. In *Proceedings of the USENIX Annual Technical Conference (ATC'20)*.

[4] CXL Consortium. 2023. Compute Express Link Specification Revision 3.0. https://www.computeexpresslink.org/download-the-specification

[5] Gen-Z Consortium. 2022. Gen-Z Final Specifications. https://genzconsortium.org/specifications/

[6] Johnathan Corbet. 2020. EXT4-DAX. https://lwn.net/Articles/717953

[7] Peter Druschel and Larry L. Peterson. 1993. Fbufs: A High-Bandwidth Cross-Domain Transfer Facility. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'93)*.

[8] 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 Eleventh European Conference on Computer Systems (EuroSys'14)*.

[9] Subramanya R. Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. 2016. Data Tiering in Heterogeneous Memory Systems. In *Proceedings of the European Conference on Computer Systems (EuroSys'16)*.

[10] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. 2012. MegaPipe: A New Programming Interface for Scalable Network I/O. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*.

[11] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. 2019. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries. In *Proceedings of the USENIX Annual Technical Conference (ATC'19)*.

[12] Intel. 2022. 3D XPoint Breakthrough Non-Volatile Memory. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-micron-3d-xpoint-webcast.html

[13] Intel. 2022. Persistent Memory Development Kit. https://pmem.io/pmdk

[14] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. 2019. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. *CoRR* (2019). https://doi.org/arXiv:1903.05714

[15] Myoungsoo Jung. 2023. Hello bytes, bye blocks: PCIe storage meets compute express link for memory expansion (CXL-SSD). In *Proceedings of Conference on Hot Topics in Storage and File Systems (HotStorage'23)*.

[16] 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 Symposium on Operating Systems Principles (SOSP'21)*.

[17] 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 ACM Symposium on Operating Systems Principles (SOSP'19)*.

[18] Juno Kim, Yun Joon Soh, Joseph Izraelevitz, Jishen Zhao, and Steven Swanson. 2020. SubZero: Zero-copy IO for Persistent Main Memory File Systems. In *Proceedings of Asia-Pacific Workshop on Systems (APSys'20)*.

[19] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. 2017. Strata: A cross media file system. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'17)*.

[20] Benjamin C. Lee, Ping Zhou, Jun Yang, Youtao Zhang, Bo Zhao, Engin Ipek, Onur Mutlu, and Doug Burger. 2010. Phase-Change Technology and the Future of Main Memory. *IEEE Micro* 30, 1 (2010), 143–143.

[21] Ruibin Li, Xiang Ren, Xu Zhao, Siwei He, Michael Stumm, and Ding Yuan. 2022. ctFS: Replacing File Indexing with Hardware Memory Translation through Contiguous File Allocation for Persistent Memory. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST'22)*.

[22] Yubo Liu, Hongbo Li, Yutong Lu, Zhiguang Chen, Nong Xiao, and Ming Zhao. 2020. HasFS: optimizing file system consistency mechanism on NVM-based hybrid storage architecture. *Cluster Computing* 23 (2020), 2510–2515.

[23] 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 (EuroSys'16)*.

[24] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. 1999. IO-Lite: A Unified I/O Buffering and Caching System. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI'99)*.

[25] Yuxin Ren, Gabriel Parmer, Teo Georgiev, and Gedare Bloom. 2016. CBufs: Efficient, System-Wide Memory Management and Sharing. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management (ISMM'16)*.

[26] Vasily A. Sartakov, Lluís Vilanova, and Peter Pietzuch. 2021. CubicleOS: A Library OS with Software Componentisation for Practical Isolation. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'21)*.

[27] Timothy Stamler, Deukyeon Hwang, Amanda Raybuck, Wei Zhang, and Simon Peter. 2022. zIO: Accelerating IO-Intensive Applications with Transparent Zero-Copy IO. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI'22)*.

[28] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'11)*.

[29] Yongfeng Wang, Yinjin Fu, Yubo Liu, Zhiguang Chen, and Nong Xiao. 2022. Characterizing and Optimizing Hybrid DRAM-PM Main Memory System with Application Awareness. In *Proceedings of Design, Automation & Test in Europe Conference & Exhibition (DATE'22)*.

[30] 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 *Proceedings of USENIX Conference on File and Storage Technologies (FAST'23)*.

[31] Jian Xu and Steven Swanson. 2016. NOVA: A Log-Structured File System for Hybrid Volatile/Non-Volatile Main Memories. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST'16)*.

[32] 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 *Proceedings of USENIX Conference on File and Storage Technologies (FAST'22)*.