Optimizing File Systems on Heterogeneous Memory by Integrating DRAM Cache with Virtual Memory Management

Yubo Liu, Yuxin Ren, Mingrui Liu, Hongbo Li, Hanjun Guo, Xie Miao, and Xinwei Hu, *Huawei Technologies Co., Ltd.;*
Haibo Chen, *Huawei Technologies Co., Ltd. and Shanghai Jiao Tong University*

https://www.usenix.org/conference/fast24/presentation/liu-yubo

This paper is included in the Proceedings of the 22nd USENIX Conference on File and Storage Technologies.

February 27–29, 2024 • Santa Clara, CA, USA

978-1-939133-38-0

Optimizing File Systems on Heterogeneous Memory by Integrating DRAM Cache with Virtual Memory Management

Yubo Liu[1], Yuxin Ren[1], Mingrui Liu[1], Hongbo Li[1], Hanjun Guo[1], Xie Miao[1], Xinwei Hu[1], and Haibo Chen[1,2]
[1]*Huawei Technologies Co., Ltd.* [2]*Shanghai Jiao Tong University*

Abstract

This paper revisits the usage of DRAM cache in DRAM-PM heterogeneous memory file systems. With a comprehensive analysis of existing file systems with cache-based and DAX-based designs, we show that both suffer from suboptimal performance due to excessive data movement. To this end, this paper presents a cache management layer atop heterogeneous memory, namely FLAC, which integrates DRAM cache with virtual memory management. FLAC is further incorporated with two techniques called zero-copy caching and parallel-optimized cache management, which facilitates fast data transfer between file systems and applications as well as efficient data synchronization/migration between DRAM and PM. We further design and implement a library file system upon FLAC, called FlacFS. Micro benchmarks show that FlacFS provides up to two orders of magnitude performance improvement over existing file systems in file read/write. With real-world applications, FlacFS achieves up to 10.6 and 9.9 times performance speedup over state-of-the-art DAX-based and cache-based file systems, respectively.

1 Introduction

Emerging persistent memory (*e.g.*, 3DXPoint [14, 26] and CXL-based SSD [18]) promise fast and byte-addressable accesses to large volume of data. This brings a trend of deploying heterogeneous memory of a volatile memory layer (DRAM) and a persistent memory layer (PM). However, it raises a natural question: how to maximize performance atop such a heterogeneous architecture?

State-of-the-art file systems for heterogeneous memory can mainly fall into two categories: using DRAM as a cache for PM (DRAM cache) or providing direct access (DAX) to PM. Caching pages in DRAM, such as the VFS page cache, is a common design in traditional file systems (*e.g.*, EXT4 and XFS [44]) to bridge the performance gap between fast DRAM and slow persistent storage devices (*e.g.*, HDD and SSD). However, many previous studies [10] show that DRAM cache incurs significant overhead under the fast, all-memory architecture. Therefore, most existing systems (*e.g.*, NOVA [51], SplitFS [20], and ctFS [31]) resort to DAX , which bypasses the DRAM cache and performs I/Os on PM directly.

However, DAX is still suboptimal for heterogeneous memory file systems. First, the performance gap between PM and DRAM cannot be ignored in the present and future (the PM latency may range from hundreds to thousands of nanoseconds [18], which is much higher than DRAM). Such high PM latency easily limits the file system performance. Second, DAX potentially loses the performance benefit of data locality provided by the DRAM cache. According to our analysis, the performance of DAX-based systems is inferior to that of DRAM cache systems in scenarios with high concurrency and strong data locality, even though the VFS page cache framework introduces high software overhead. Last but not least, instant persistence is the best scene of DAX; but it is an overkill in many real-world scenarios [49].

To this end, this paper revisits the usage of DRAM cache in heterogeneous memory architecture. According to our quantitative analysis, we summarize two challenges of building an efficient cache framework on heterogeneous memory:

Challenge 1. Data transfer overhead between application buffer and DRAM cache is high. Transferring data between the application and DRAM 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.

Challenge 2. The impact of "cache tax" is significant. In addition to data transfer, existing cache frameworks spend lots of effort to synchronize (flushing dirty data) and migrate data across DRAM cache and PM (moving data into/out of cache). Currently, such operations are implemented in a synchronous and sequential way and significantly increase performance penalty (more than 30%).

We argue that the main reason is that existing cache frameworks (*e.g.*, VFS page cache) are built upon the virtual memory subsystem, which makes it difficult to avoid the cache-application data copying and hide the overhead of cross-layer data synchronization/migration. Hence, this paper advocates an integration of DRAM page cache into virtual memory management of operating systems and proposes FLAC (**FLA**t **C**ache), a novel cache framework for heterogeneous memory. FLAC provides a single-level address space of heterogeneous memory. File system developers can leverage the exposed interfaces to the data store on FLAC to enjoy the efficient DRAM cache on data I/O paths (other modules of file system are independent of FLAC). FLAC further builds 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 [9, 30, 38, 40], simply adopting this idea in the file system cache faces some unique challenges. First, FLAC addresses the side-effects of page unaligned and expensive COW page fault by the sliding window buffer and batch faulting/detaching, respectively. Second, the zero-copy caching makes the page have multiple versions, and it requires FLAC to have a new cache management mechanism to ensure data consistency and high concurrency.

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

To demonstrate the effectiveness of FLAC, we design and implement FlacFS, a file system for building its data store on FLAC. Evaluation shows that FlacFS provides a performance increase of more than two orders of magnitude over state-of-the-art DAX-based and cache-based file systems in the micro benchmarks. With real-world applications, FlacFS achieves up to 10.6 and 9.9 times performance speedup over DAX-based and cache-based systems, respectively.

The contributions of this paper include:

- It quantitatively analyses the cache and DAX frameworks and summarizes the key challenges of cache framework design on heterogeneous memory.
- It designs and implements FLAC, a novel cache framework for heterogeneous memory file systems that including the techniques of zero-copy caching and parallel-optimized cache management.
- It implements a file system (FlacFS) based on FLAC, and demonstrate the benefits via micro/macro benchmarks and real-world applications.

The rest of this paper is organized as follows: Section 2 introduces the background and motivation; Section 3 presents the key designs of FLAC; Section 4 introduces the implementation of FlacFS; Section 5 discusses the limitations and challenges; Section 6 shows the detailed evaluation of FLAC/FlacFS; Section 7 concludes the paper.

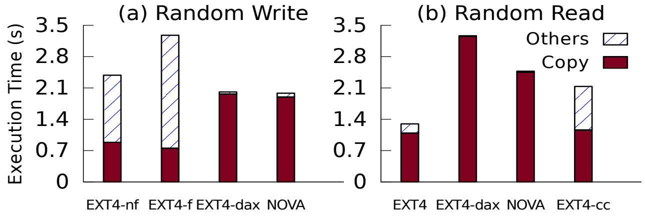


Figure 1: Traditional Cache *vs.* DAX. "-f/-nf": with/without background flushing; "-cc": cold cache.

2 Background and Motivation

2.1 Heterogeneous Memory

With the emergence of new persistent storage media (*e.g.*, 3DXPoint [14], CXL-based SSD [18, 23, 55, 56]), 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 [33]. *1) Latency Gap.* The latency of DRAM is about tens of nanoseconds, while the latency of low-level memory range from hundreds to thousands of nanoseconds [18, 34]. *2) Bandwidth Gap.* The bandwidth of DRAM can reach tens of GB, while it is only about a few GB of existing PM [52]. *3) Concurrency Gap.* The PM has lower concurrency than the DRAM [11, 20]. For example, existing PM hardware based on 3DXPoint is hard to scale beyond 4 concurrent [16] in the single channel. In summary, the performance gap between DRAM and PM and between different types of PMs cannot be ignored, which make it challenging to design efficient storage systems for heterogeneous memory.

2.2 Direct Access (DAX) *vs.* Cache

Heterogeneous memory raises an important question for file system designers: what kind of storage framework can take advantage of different memory devices? There are two typical storage frameworks are used on heterogeneous memory: 1) traditional page cache based on the DRAM-block device architecture (*i.e.*, VFS page cache) and 2) direct access (DAX). We quantitatively analyze three typical file systems with the VFS page cache (EXT4) and DAX (EXT4-DAX [7], NOVA [51]) by performing random writes/reads on a 10GB file with 2MB I/O (the testbed is introduced in §6). Three important observations are found from our experiments:

Observation 1*: Existing DAX and cache frameworks are sub-optimal, and 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, the performance gap between PM and DRAM cannot be ignored. Figure 1 shows that the VFS page cache still has better performance than DAX in some cases (*e.g.*, read). Second, taking advantage of data locality is an effective method of performance optimization, but DAX misses this opportunity. 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 [49].

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

Data I/Os (file read/write) need to transfer data between the application buffer and the storage system (cache space or persistent data space). Memory copy is the mainstream method to transfer data, but in our experiment, it takes up more than 23% and 96% of the total overhead in cache-based and DAX-based file systems, respectively. In particular, the performance bottleneck of data copy between cache and application is obvious in heterogeneous memory systems since the latency of PM is much lower than traditional block devices.

Observation 3*: The "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. Figure 2 shows the core processes of the typical DRAM page cache, which reveals the composition of the "cache tax". From our experiments, the data synchronization (background dirty flushing) and data migration (cache miss handling) lead to 37% and 65% performance declines, respectively.

2.3 Motivation

According to the previous analysis, an efficient heterogeneous memory cache framework needs to meet two requirements: 1) low application-cache transfer overhead and 2) low "cache tax" impact. However, exiting cache frameworks (*e.g.*, VFS page cache) do not fully exploit the potential of DRAM cache in heterogeneous memory systems. They are difficult to avoid the data copy between the DRAM cache and the application buffer. At the same time, they are difficult to transparently overlap the critical I/O paths and the cross-layer data synchronization/migration. The motivation of this work is to integrate the DRAM page cache with the virtual memory management subsystem, and it brings two key principles for our design.

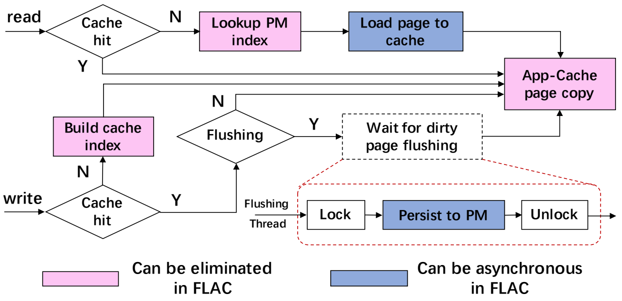


Figure 2: Typical Diagram of Page Cache.

Principle 1: *Optimizing data transfer between the cache and the application by zero-copy.* Traditional DRAM cache frameworks simply take advantage of the performance advantages of DRAM, but ignore another important advantage of DRAM cache: it is homogeneous with the application runtime. By co-designing the DRAM cache and the virtual memory subsystem, the data copy during application-cache transfer can be avoided by page mapping. FLAC proposes the zero-copy caching technique to avoid application-cache data copy and redundant indexes (red squares in Figure 2).

Principle 2: *Reducing the impact of "cache tax" by hiding the data synchronization/migration overhead.* The "cache tax" is difficult to eliminate, but their impact on the critical I/O paths can be reduced by improving the parallelism between the data synchronization/migration and the front-end I/Os. FLAC proposes the parallel-optimized cache management mechanism to amortize the data synchronization/migration overhead in the background (blue squares in Figure 2).

3 FLAC Design

3.1 Overview

This work proposes FLAC, a **FLA**t **C**ache framework integrated with the virtual memory subsystem to deeply explore the potential of cache for heterogeneous memory systems. As shown in Figure 3, FLAC maintains a range of contiguous virtual memory addresses, called FLAC space. The size of FLAC space is equal to the usable PM space, and it provides the data storage area with the built-in DRAM page cache for the heterogeneous memory file system. The FLAC space is indexed by the heterogeneous page table, which makes page physical locations transparent and exposes a single-level memory space to file system developers. Data is transferred between the application and the FLAC space with the zero-copy approach (§3.2), and synchronized/migrated between DRAM and PM with the parallel-optimized mechanism (§3.3). Table 1 shows the main APIs of FLAC.

`init_flac`: This API is used to initialize and bind the given PM to the FLAC space for file data storage. If the FLAC space has already been created on the PM, it rebuilds the FLAC space from the last consistent state.

`zcopy_from/to_flac`: The file system based on FLAC internally uses these two APIs to transfer data and support

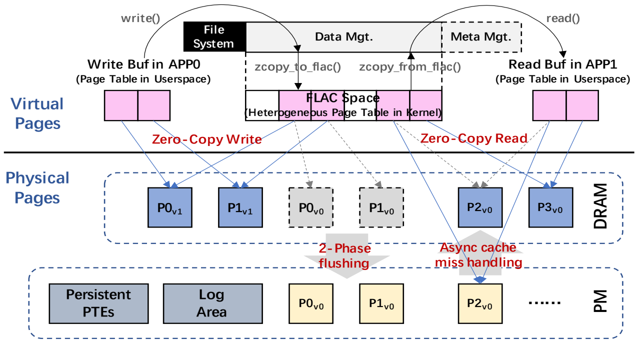


Figure 3: Architecture of FLAC. File system runs the data management on top of FLAC. Application accesses data by file read/write, and they are converted to the zero-copy transfer APIs in FLAC. Data is stored in a flat memory address, which is transparent to the physical locations of the pages through heterogeneous page table.

file read/write operations, which are similar to the action of `copy_to/from_user` in traditional kernel file systems.

`pflush_add/commit`: This pair of APIs are used to explicitly flush dirty data from DRAM to PM, and they give developers the flexibility to customize flushing policies. Dirty pages are added to a flush handle (`pflush_add`) and flushed to PM in a transaction (`pflush_commit`). File systems use the *fs_metalog* parameter to ensure the consistency of FS-level metadata during data flushing.

`pfree`: This API is used to atomically reclaim a range of FLAC space. It invalidates the page on the DRAM/PM and removes the page table mapping.

Architecture and Usage. FLAC runs under the file system as a development framework. Developers of heterogeneous memory file systems customize the file data management on the FLAC space (*e.g.*, file read/write logic and data flushing policy) by encapsulating the APIs above, and applications access the data on the FLAC space by normal file interfaces. The other modules of the file system (*e.g.*, metadata management) are independent to FLAC, which can be flexibly designed and implemented. FLAC's APIs can be called by `ioctl`s or kernel functions, which allows developers to flexibly implement file systems in the userspace or kernel.

3.2 Zero-Copy Caching

3.2.1 Heterogeneous Page Table

As Figure 3 shows, FLAC uses the heterogeneous page table, a customized sub-level table (including one or multiple PUDs) of the kernel page table, to maintain the FLAC space: it is a range of consecutive kernel virtual memory addresses and its size is equal to the usable PM size. The positions of pages (DRAM/PM) in the FLAC space are transparent for the file systems running upon it. This design has two meanings: **1)** 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. **2)** Page table entries (PTEs) belonging to the FLAC space are replicated in PM for fault recovery. The heterogeneous page table unifies the page indexes of cache and persistent storage and simplifies cache access and management.

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	Zero-copy transfer data between the application and the FLAC space
`pflush_add`	pflush_handle addr size	Attach (map) the pages to the flushing buffer and add to the handle
`pflush_commit`	pflush_handle fs_metalog	Flush the pages in the handle and update the metadata atomically
`pfree`	addr size fs_metalog	Reclaim the PM pages and update the metadata atomically

PM is divided into three areas. **1)** The persistent PTE area records the mapping information between the virtual addresses and the PM pages. All PTEs of the heterogeneous page table are mirrored on PM. When a page is flushed from DRAM to PM, FLAC records the related offset in the PM device to the persistent PTE for recovery. **2)** The log area logs the modifications of FLAC-level (*e.g.*, persistent PTE) and FS-level metadata (*e.g.*, inode) by the FS-FLAC collaboration logging mechanism when the persistent data modification APIs (`pflush_commit/pfree`) are called. **3)** The page area contains multiple 4KB units for file data storage. Data pages are persisted in this area during flushing.

Developers call `init_flac` to prepare the FLAC space and it is responsible for rebuilding the heterogeneous page table. First, FLAC checks the logs to determine whether the system exits abnormally, and if so, recovers it to the last consistent state. Then, the PGDs, PUDs, PMDs, and PTEs of the heterogeneous page table are created in DRAM. In particular, the locations in PTEs are rebuilt by translating the related offsets in the persistent PTEs (if have) to the physical location of the PM pages. After initialization, all valid pages on PM are mapped to the heterogeneous page table.

3.2.2 Transfer Data with Page Attaching

The core technique used to achieve the goal of zero-copy is a new virtual memory management operation – page attaching (Interface (1)). The `attach` includes four parameters: two address and their size, and the permission mode. Page attaching maps the pages of the source address (*from_addr*) to the destination address (*to_addr*) with the given size. The permission mode (*pmode*) allows users to set permissions on source and destination addresses after page attaching (*e.g.*, read-only).

Page attaching first searches the PTEs of source and destination addresses then maps the physical pages of the source address to the destination address (the permission and page reference counter are also set) and finally flushes the TLB. It will be aborted if the source addresses are not faulted (*i.e.*, mapped to the physical pages), but there is no restriction on the destination addresses. If the destination addresses are faulted (*e.g.*, overwrite), the reference counter of the old physical pages will be reduced and they are reclaimed by the memory subsystem when their counters reach 0.

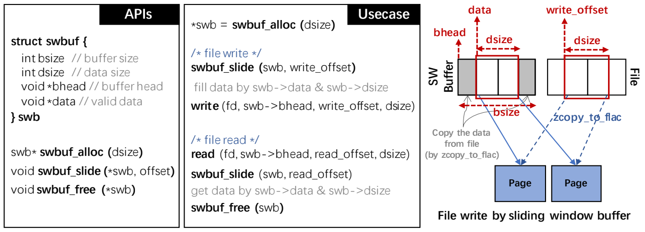
$$attach\,(to\_addr, from\_addr, size, pmode) \tag{1}$$

The APIs of `zcopy_from/to_flac` encapsulate `attach` for transferring data between the application and the FLAC space. For data security and isolation, the operated pages are set to read-only after attaching, so that subsequent writes on these pages will transparently trigger copy-on-write (COW) page fault, which ensures that the memory operations inside the application do not affect the data that have been mapped to the global cache and other applications. In particular, benefiting from the heterogeneous page table, pages can be attached whether they are cached or not and this feature delivers the design of asynchronous cache miss handling.

Handling Page Unaligned. The file I/O (<fd, offset, size>) is translated to the address and size of the FLAC space by the upper-layer file system, and the data is transmitted between the FLAC space and read/write buffer by page attaching (`zcopy_from/to_flac`). Page attaching requires that the operated addresses and sizes are page aligned, but file I/Os and buffers are arbitrary, resulting in the page unaligned problem. Our solution is to attach all the pages containing the required data and use a cursor to locate valid data in the application buffer. FLAC requires the upper-layer file system to ensure that the start address of each file is page aligned. Given an unaligned file I/O and a buffer, we first need to extend the file I/O to the FLAC space range that contains all the required pages, which is achieved by the automatic alignment mechanism. In addition, we need to ensure that the buffer is large enough, page aligned, and can represent the valid data in it, which is achieved through the sliding window buffer technology.

Automatic Alignment. The `zcopy_from/to_flac` check whether the given FLAC address and size are page aligned, and if it doesn't, the access range (start and end addresses) is automatically extended to page aligned. Then, the page attaching is executed. In particular, if the destination space is already mapped to the (old) pages, the hole(s) caused by automatic alignment is filled with the data in the old page(s) through copy after attaching.

Sliding Window Buffer. As Figure 4 shows, the application allocates (`swbuf_alloc`) the sliding window (SW) buffer by using the read/write size (*dsize*). It includes a `swbuf` structure and a page unaligned space (**bhead*) in the size of $\lceil dsize/4096 \rceil$+1 pages (*bszie*). The application uses the **bhead* in SW buffer to serve file read/write with arbitrary offset and size. However, the valid data may not start with the **bhead* due to the automatic alignment in `zcopy_from/to_flac`. Before using the data in the read/write buffer, the application is asked to call the `swbuf_slide` to calculate the window of valid data in the SW buffer by using the file offset and **bhead*. The head of valid data is recorded in the **data*. It is worth noting that the SW buffer is not mandatory if the application can guarantee the offset and size of file read/write are page aligned.



```
APIs
struct swbuf {
   int bsize  // buffer size
   int dsize  // data size
   void *bhead // buffer head
   void *data // valid data
} swb

swb* swbuf_alloc (dsize)
void swbuf_slide (*swb, offset)
void swbuf_free  (*swb)
```

```
Usecase
*swb = swbuf_alloc (dsize)

/* file write */
swbuf_slide (swb, write_offset)
 fill data by swb->data & swb->dsize
write (fd, swb->bhead, write_offset, dsize)

/* file read */
read  (fd, swb->bhead, read_offset, dsize)
swbuf_slide (swb, read_offset)
get data by swb->data & swb->dsize
swbuf_free (swb)
```

data
write_offset
bhead
dsize
dsize
SW Buffer
File
bsize
zcopy_to_flac
Copy the data from file (by zcopy_to_flac)
Page
Page
File write by sliding window buffer

Figure 4: Sliding Window (SW) Buffer. Application uses SW buffer to serve page unaligned read/write. The sliding window is used to identify valid data in the buffer.

Reducing COW Page Fault Overhead. The zero-copy data transfer ensures security and isolation by setting the source and destination memory read-only, and this makes the first write operation (`store` instruction) to the source (write buffer) or destination (read buffer) memory after `attach` to trigger COW page fault. According to our analysis, the main overhead of COW page fault includes two aspects: TLB flush and data copy. FLAC proposes two techniques to reduce the impact of COW page fault for different use cases.

Batch Fault. In some scenarios, applications directly process data in the read/write buffer, which causes a large number of pages in the buffer to be faulted with COW. FLAC optimizes this unfriendly case by executing the COW page faults in batch, thus reducing the number of TLB flushes. Batch faulting copies the data from the original pages to the new pages in batch and only needs to flush the TLB once. The application can call the `bfault` API for the read/write buffer before the data in the buffer are processed.

Detach. In some scenarios, the application just wants to reuse the read/write buffer's space instead of its data, which is a false sharing scenario (*e.g.*, pre-allocating a log buffer and reusing it after it is written to the storage system). To avoid COW page fault in this scenario, FLAC provides the `detach` API to remap the addresses of the read/write buffer to some new anonymous pages to absorb subsequent memory operations. The application can call the `detach` before the read/write buffer is reused. After detaching, the subsequent memory writes to the buffer will not trigger COW page faults.

3.3 Parallel-Optimized Cache Management

Due to the zero-copy caching design, FLAC requires a cache management mechanism for its multi-version feature, while ensuring a low "cache tax" impact. Fortunately, the multi-version feature and the heterogeneous page table design of

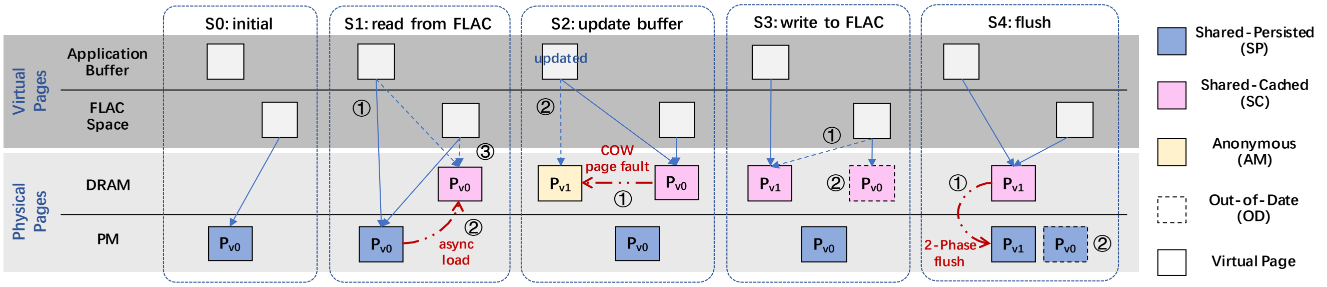


Figure 5: Page State/Version Transition. Solid blue arrow: current mapping; Blue dashed arrow: future mapping; Red dashed arrow: data copy.

FLAC allow us to fully exploit the parallelism of data synchronization/migration with critical I/O paths.

3.3.1 Parallel-Optimized Synchronization/Migration

Existing cache frameworks execute cache flushing and cache miss handling with large synchronization and migration overhead: Cache flushing locks the dirty pages until they are completely flushed, which blocks the front-end writes and dramatically reduces the performance; Cache miss handling blocks the I/Os until the pages are loaded to the DRAM cache. They are optimized by the following two techniques.

2-Phase Flushing. FLAC splits the dirty pages flushing into two phases: collection (`pflush_add`) and persistence (`pflush_commit`). The collection phase adds the given dirty pages to a flush handle, which allocates a fresh virtual memory address space as a temporary flush buffer and attaches the dirty pages to it. This phase requires a lock to prevent concurrent writes from modifying the target pages. The persistence phase is responsible for persisting the dirty pages in the flush handle to PM. This phase 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-layer copy, the 2-Phase flushing mechanism significantly reduces the blocking time on concurrent writes due to dirty page synchronization (*e.g.*, background flushing).

The persistence phase is atomic. It flushes data pages by the log-structured method, *i.e.*, dirty data is written to the new PM pages and the out-of-date PM pages are reclaimed. The persistent PTEs are updated after dirty data is successfully flushed. The modifications of the PM page allocator and persistent PTEs are logged to ensure crash consistency. In addition, file systems may require FS-level metadata updates and data persistence to be the same transaction. FLAC provides the FS-FLAC collaboration logging mechanism to meet this goal (§3.3.4).

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), but it is expensive on read operation. Benefiting from the heterogeneous page table, FLAC can directly attach the PM pages to the read buffer (returns immediately) and handle the cache miss asynchronously. A background thread in FLAC is responsible for loading the missed pages to DRAM and remapping the PTEs of FLAC space and application buffer(s) pointing to those PM pages to the cached DRAM pages. The page may have been modified to trigger the COW page fault before it is loaded to DRAM, which means it has the newest version in DRAM. The asynchronous cache miss handling checks if the page already has a new version in DRAM and skips if it does. This design makes it possible for the overhead of handling cache misses to be amortized in the background, thereby reducing the data I/O latency.

3.3.2 Page State/Version Transition

The page may have different states and versions in FLAC. There are four states of a page in FLAC: shared-persistent (SP), shared-cached (SC), anonymous (AM), and out-of-date (OD). SP and SC pages are stored in the global areas (PM data page area/DRAM cache) and are read-only; AM pages are readable and writable, which is the same as the normal anonymous page in processes; OD pages are invisible to the file system on FLAC and are managed by FLAC's reclamation mechanism. Figure 5 shows an example of page state/version transition through a sequence of operations. As the initial stage (S0), we assume that there is a page on the FLAC space and it is in the PM data page area.

Stage 1: Read from FLAC. The target page is an SP page, so cache miss happens when the application reads the page from the FLAC space to the application buffer (by the file system interface). FLAC first maps the buffer to the SP page and the read operation is returned (①). Then, the target page is asynchronously loaded to DRAM as an SC page (②) and the virtual pages of application buffer and FLAC space are remapped to the new SC page (③).

Stage 2: Update the buffer. When the application tries to update (by `store` instruction) the data in the buffer, a new version AM page is created by COW (①), and then it is mapped to the buffer address to absorb the updates (②). COW page fault is only triggered at the first time the SP/SC page (depending on if it is cached) is updated by the application, and subsequent memory accesses will directly perform on the AM page. In addition, the old version page in FLAC is still in its original state (SP/SC).

Stage 3: Write to FLAC. When the application writes (attaches) the page back to FLAC, the state of the page mapped by the buffer is changed from AM to SC (①). The state of the old page that is mapped to the target address in FLAC will be changed to OD and reclaimed by FLAC when it is clean (②).

Stage 4: Background flush. The page will be synchronized to PM when the background flushing is triggered, which is implemented by the upper-layer file system. During background flushing, a new SP page is created (①) and the old version of the SP page is reclaimed by FLAC after the data is successfully synchronized (②).

Page State Semantics. Page state is at the process granularity as FLAC maintains the state by manipulating the process' page table. The FLAC-based file system has the same semantics in file read/write operations as traditional file systems. After one thread attaches (by file read/write) a page with SC/SP state, other threads in the same process can read it consistently. Once the attached page is modified and causes COW to generate an AM page, it can be shared within the process. FLAC currently does not support read/write shared pages between different processes (*e.g.*, using the FLAC space as inter-process shared memory). FLAC enforces mandatory isolation and COW in different processes will generate separate AM pages. However, FLAC may support this case by considering reverse page mapping during COW.

Durability. FLAC does not restrict the durability model, which is defined and implemented by the upper-layer file system. Our prototype FLAC-based system (FlacFS) uses the same durability model as traditional file systems. The cached data is persisted to PM under two cases, *i.e.*, the background flushing is triggered and the `fsync` is called by the file system user. FlacFS implements background flushing and `fsync` by encapsulating the data synchronization operations of FLAC (`pflush_add/commit`), and FLAC guarantees that these operations are atomic and recoverable by the FS-FLAC collaboration logging.

3.3.3 Cache Policy

The size of FLAC space is equal to the usable PM size, but the maximum DRAM cache usage is controllable and page eviction is triggered when the cache is full. Due to the zero-copy design naturally brings the advantage of deduplication, FLAC counts the pages that are *only* mapped by the FLAC space into the used size, and the pages that are mapped by both the FLAC space and application buffer are treated as an in-process pages (without taking up the cache space).

As this work mainly focuses on the cache framework, we just design a simple cache policy in our prototype, *i.e.*, it selects pages for eviction with the round-robin approach. Existing cache algorithms [17, 35, 41, 53, 54] can also be used for FLAC. For the sake of simplicity, a page can be evicted only when two conditions are met. First, the page is clean, *i.e.*, it has been synchronized to PM by background flushing or `fsync`. Second, the reference counter of the page is 1, which means that the page is only mapped by the FLAC space and not used by any application. After a page is evicted to PM, the target PTE of the FLAC space is remapped to the PM page and the DRAM page will be reclaimed.

In particular, the multi-version feature of FLAC does not incur additional space overhead compared to traditional page cache. The new version of the page being created in the application process by COW page fault, the new version does not take up space in the page cache before it is overwritten to FLAC. After overwriting, the virtual address of FLAC space is mapped to the new version and the old version is reclaimed.

3.3.4 FS-FLAC Collaboration Logging

For normal shutdown, the recovery process of FLAC only needs to rebuild the heterogeneous page table according to the persistent PTEs. For an unexpected shutdown, FLAC must recover the system to the last consistent state. As described in the 2-Phase flushing, persistent data modifications in FLAC (`pflush_commit/pfree`) are atomic. However, along with data modifications, the file system upon FLAC may need to update the related FS-level metadata (*e.g.*, page index) on PM in the FS-level atomic operation (*e.g.*, append).

To ensure complete consistency, FLAC provides the FS-FLAC collaboration logging mechanism to allow the data modifications in FLAC and FS-level metadata updates in a transaction. It requires the file system to make two efforts: **1)** File system should provide the self-formatted metadata log (*fs_metalog* parameter) when the persistent data modification APIs are called. FLAC concatenates the internal (FLAC-level) and external (FS-level) metadata log into an entry and appends it to the log area after a successful persistent data modification operation. **2)** File system should overload an external metadata recovery function provided by FLAC. During recovery, FLAC first commits the internal metadata log and then calls the external recovery function to commit the external metadata. After all logs are committed, FLAC can be recovered as normal shutdown.

4 Case Study: FlacFS

We implement FlacFS, a file system based on FLAC to show the usage and benefits of FLAC. FlacFS contains three additional designs, the metadata management, data management, and mechanism of security and consistency. FlacFS is a library file system implemented through memory semantics. Figure 6 shows the architecture of FlacFS. As FlacFS focuses on the cache framework, we draw from existing works on designs in some aspects.

4.1 Metadata Management

The metadata area is mapped as traditional shared memory on userspace. It includes two separate virtual memory addresses for DRAM and PM, while they are created by `shmget` and `mmap`, respectively. The metadata (inode) of the directory/file

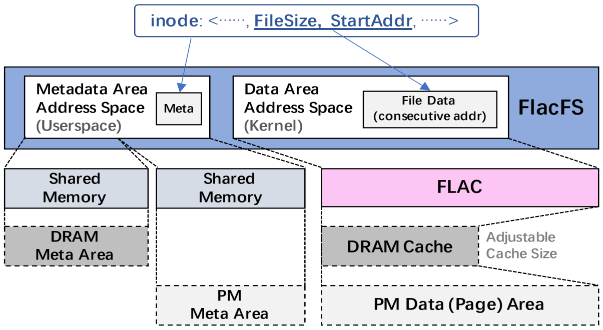


Figure 6: Implementation of FlacFS. The data space is built on FLAC, and file read/write are implemented by encapsulating FLAC's APIs. The metadata space is built on traditional shared memory. All inodes are cached in the DRAM hash table using their paths as keys, and their copies are persistently stored on PM.

is treated as a KV pair and stored in the inode hash table on shared memory using its full path as the key. The inode table is stored on both DRAM and PM to accelerate metadata operations. The metadata operations are performed in the DRAM inode table immediately and flushed to the PM inode table when the dirty data of the related files are flushed by background flushing or `fsync`. The metadata consistency is guaranteed by the FS-FLAC collaboration logging (§4.3).

Inspired by SCMFS [50] and ctFS [31], FlacFS allocates consecutive virtual memory addresses on the FLAC space for each file to store data. File inode only needs to record the start virtual address and the file size. FlacFS uses a buddy-like allocator for it. When the file size increases, a new range of consecutive virtual addresses is allocated, then the pages (existing and new) are attached to the new virtual addresses, and finally the old virtual addresses are reclaimed. This design allows FlacFS to leverage MMU to accelerate page indexing without the need for complex index structures (*e.g.*, B-tree).

4.2 Data Management

The data area is run on top of FLAC, which is created by `init_flac`. It appears to FlacFS as a range of consecutive kernel virtual memory addresses, and the data I/Os on the FLAC space are transparently cached.

File Read/Write. After the file is successfully opened, FlacFS calculates the target address range on FLAC space of the request by the start virtual address of the file (recorded in the inode) and the offset. Read and write are executed by `zcopy_from_flac` and `zcopy_to_flac` respectively, which makes the data transferring between file system and application is zero-copy.

Background Flushing. FlacFS launches a background thread periodically (10ms by default) to traverse the opened files and flush the dirty pages and related metadata to PM. It uses the 2-Phase flushing mechanism of FLAC for efficient data synchronization. For each dirty file, FlacFS creates a flush handle and collects the dirty pages according to the per-file dirty bitmap, and then uses `pflush_add` to add them to the handle (*i.e.*, attach to a temporary flush buffer). After collecting, FlacFS calls `pflush_commit` to atomically persist the dirty data to PM.

File Synchronization. Similar to traditional file systems, FlacFS provides `fsync` for users to flush data from DRAM to PM immediately. FlacFS uses the 2-Phase flushing mechanism to synchronize dirty data in `fsync`, which is similar to the background flushing. Following the semantics of `fsync`, the operation is returned after the data is persisted.

4.3 Security and Consistency

Data is protected by the kernel mode. FLAC is implemented in the kernel, and userspace applications can access it only through syscall/ioctl. 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. The metadata security can be solved by using the userspace security mechanisms or putting metadata management in the kernel. For example, the mechanism of existing systems [31, 57] can be used to ensure the metadata security, *i.e.*, the metadata area is protected by MPK [13, 42] and access permission only is granted to the user process during the metadata operation.

The FS-FLAC collaboration logging mechanism requires the upper-layer file system to provide formatted metadata modification and corresponding metadata recovery functions. FlacFS uses the newest inode as the *fs_metalog* parameter in persistent data modification APIs (`pflush_commit`, `pfree`), and overloads the external recovery function to overwrite the original inode by its newest version. FlacFS calls `init_flac` to recover the FLAC space when system restarts. After the success of `init_flac`, FlacFS rebuilds the metadata area in DRAM and the system is recovered from the crash.

4.4 Advantages of FlacFS/FLAC

FLAC allows file systems based on it to benefit from the DRAM cache while reducing the effects of "cache tax" as much as possible. Table 2 gives a comparison between FlacFS/FLAC and existing systems.

vs. Cache-based File Systems/mmap. There are many file systems designed based on the VFS. Although the 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-storage data transfer. These file systems also provide the mmap method to avoid the data transfer overhead, but it makes application design and storage backend to be coupled. Therefore, they cannot fully exploit the potential of cache in heterogeneous memory architecture.

vs. DAX-based File Systems. DAX-based systems bypass the DRAM cache in data I/O, making them suffer from high application-storage transfer overhead. Also, the latency and

Table 2: Comparison with Related Work

Type	Typical System	Data Cache	Low/Non Cache Tax Impact	App-Storage Zero-Copy	App-Storage Decouple
Cache-based FS	VFS page cache FSes (*e.g.*, ETX4, XFS [44], SPFS [49])	✔	✗	✗	✔
Cache-based mmap	mmap in VFS page cache FSes (*e.g.*, EXT4, XFS [44])	✔	✗	✔	✗
DAX-based FS	NOVA [51], SplitFS [20], WineFS [19], ctFS [31], KucoFS [5], PMFS [10] libnvmmio [6], EXT4-DAX [7], HTMFS [57], OdinFS [62], ZoFS [8]	✗	✔	✗	✔
DAX-based Runtime	Twizzler [4], Mnemosyne [46], PMDK [15] zIO [43], DaxVM [1], SubZero [22]	✗	✔	✔	✗
Flat Cache	FlacFS	✔	✔	✔	✔

concurrency of PM hardware greatly limit their performance. In particular, some DAX-based file systems also use remapping: SplitFS [20] 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 [31] 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. Some DAX-based systems focus on special design objectives, such as NUMA optimization (*e.g.*, OdinFS [62]), userspace optimization (*e.g.*, KucoFS [5], ZoFS [8], Trio [61]), and aging problem (*e.g.*, WineFS [19]). They are complementary to FlacFS.

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 [22, 43]. 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 [48]. DAX-based runtime focuses on programming directly on PM and can be seen as complementary to the file system.

vs. Other Related Work. Some PM-based file systems try to use DRAM as a cache (*e.g.*, HiNFS [37] and HasFS [32]). However, these works do not exploit the potential of the virtual memory subsystem in the cache and are designed for the simulated PM. Some file systems (*e.g.*, Strata [24], Ziggurat [60]) are optimized for other multi-layer storage architectures (DRAM-PM-SSD). Some work focuses on data management in tiered memory (*e.g.*, HeMem [39] and Johnny Cache [29]), which are complementary to FLAC.

5 Discussion

Although FLAC/FlacFS offers promising performance, it also encounters some new challenges, which we discuss below.

Page Fault Overhead. COW page fault doesn't happen at every write, and is only triggered at the first time to overwrite the buffer. The natural COW page fault overhead is high and our evaluation shows that it can reduce performance by about 30 times in the worst case without specific optimization. Our optimizations (`bfault/detach`) precisely address two key bottlenecks in COW page fault and they are easy to adapt to applications (shown below). According to our evaluation, `bfault/detach` can reduce more than 78.3% COW page fault overhead in the worst cases (§ 6.2.4) and can be used effectively in real-world scenarios (§ 6.3).

Application Adaptation. We think adaptation is simple and straightforward: First, it requires only a few code changes. We intercepted the POSIX interface to transparently adopt the file operations (open, read, write, *etc*) to FlacFS. The code changes are related only to buffer allocation and page fault optimization. Second, it needs no change to the original application code logic. The code changes are alternative (replacing buffer allocation) and/or incremental (adding `bfault/detach` before reusing the buffer). This allows applications to be "trivially" adapted to FLAC/FlacFS.

Target I/O Workloads. FLAC/FlacFS is more friendly to large I/Os, especially I/Os larger than 64KB (§ 6.2.2). Large I/Os are important in production scenarios. For example, LLM (Large Language Model) training usually makes checkpoints in the file system for recovery. Take GPT3-NEOX [12] as an example, the average I/O size generated during checkpointing is at MB-level; As another example, SQL databases (*e.g.*, openGauss [36]) typically aggregate data into large blocks (*e.g.*, 64KB) and write to the file system by large I/Os.

Design Universality. Although this work mainly focuses on the cache framework of file systems, FLAC is possible to be adapted to other storage systems. For example, KV stores (*e.g.*, [2, 3, 21, 27, 28, 47, 58, 59]) can build their DRAM cache upon the FLAC space to enjoy the benefits of zero-copy caching and efficient cache management.

6 Evaluation

We compare FlacFS to a wide range of heterogeneous memory file systems to demonstrate the benefits of FLAC framework.

Cache-based Systems. Systems of this type include EXT4 and FlacFS. EXT4 is representative of file systems using the

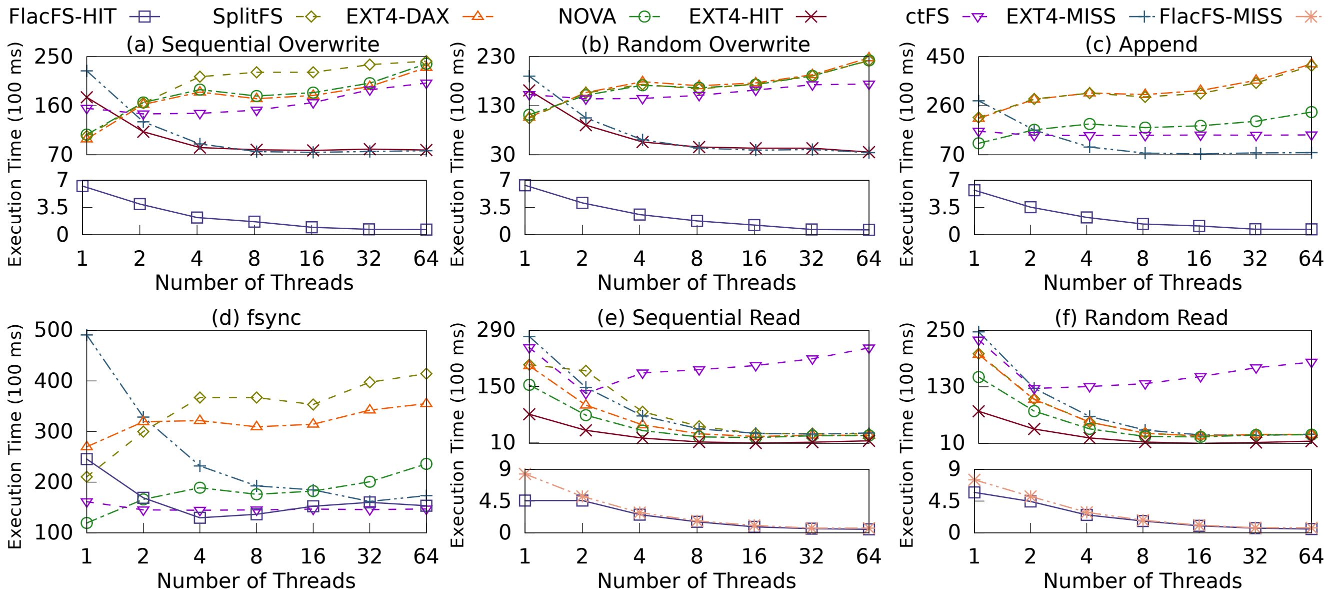


Figure 7: Micro Benchmark Performance.

VFS page cache (*e.g.*, XFS [44] and SPFS [49]). The dirty data flushing period is set to 10ms and 100ms for FlacFS and EXT4, respectively. FlacFS ensures the consistency of metadata and data, while EXT4 only ensures metadata consistency (ordered mode). If not specified, EXT4 and FlacFS trigger page eviction unless memory allocation fails, which is the default policy in Linux.

DAX-based Systems. Systems of this type includes EXT4-DAX [7], NOVA [51], SpiltFS [20], and ctFS [31]. Data I/Os of these systems bypass the VFS page cache and perform on PM directly. NOVA is set to sync mode, while SplitFS and ctFS are set to POSIX mode. All tested DAX file systems only ensure the metadata consistency, while FLAC ensures both metadata and data consistency.

Testbed. All experiments are run on a server with two Intel Xeon CPUs, 256GB RAM, and 1TB (128GB×8) PM. FlacFS and EXT4 use Ubuntu 20.04 with Linux 5.1, and others file systems use the kernel versions they can support.

6.1 Benchmark Performance

6.1.1 Micro Benchmark

We evaluate the duration of performing append, overwrite, read, and fsync-after-append (`fsync` is called after each write) on 64 1GB files with random and sequence patterns. The I/O size is 2MB and there is no contention for accesses between files in these experiments. Figure 7 shows the results. For the cache-based file systems, "*-HIT" and "*-MISS" represent cache hits and misses, respectively (analyzed in §6.2.1).

In the write scenarios, FlacFS provides a maximum performance increase of more than two orders of magnitude over other tested systems. In the read scenarios, FlacFS outperforms other tested systems by more than 200 times. The zero-copy caching in FLAC significantly reduces the data copy overhead between the application's write buffer and the file system, while all other systems suffer from this copying overhead. Compared with another cache-based system, EXT4, the data persisting phase during background flushing in FlacFS does not block the front-end writes, which significantly improves the performance in write-intensive scenarios. In the fsync-after-append scenario, FlacFS is comparable to the best of the DAX file systems and better than EXT4. Although dense `fsync` is not friendly to FlacFS, it still performs well due to the lightweight nature of FLAC.

At the framework level, we observe that the DAX-based systems have lower scalability than cache-based systems (EXT4 and FlacFS) under write-intensive workloads. The DAX approaches are difficult to scale beyond even 2 concurrent threads in Figure 7 (a) - (d) because they reach the bandwidth and concurrency limitation of PM. In summary, these results demonstrate that FLAC can fully exploit the potential of DRAM cache in heterogeneous memory file systems.

6.1.2 Macro Benchmark

We use two I/O intensive workloads in Filebench [45] to evaluate the performance of FlacFS in the scenarios with mixed operations (including many types of data and metadata operations). All workloads use 128MB file and 2MB I/O. The main process of Filserver is to create files, write data to the files, and then read data from the files. The main process of Webserver is to create and append files, and then read the files repeatedly. In particular, read operations have stronger locality than write operations in these workloads.

Figure 8 shows that the throughput of FlacFS is higher than other tested file systems by more than 40 times and 20 times in Fileserver and Webserver, respectively. At the same

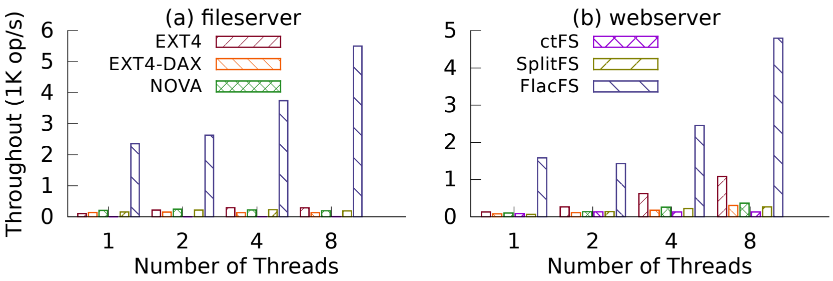


Figure 8: Filebench Performance.

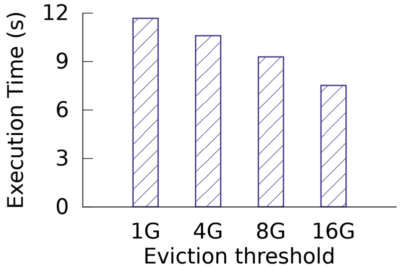


Figure 9: Cache Eviction Overhead.

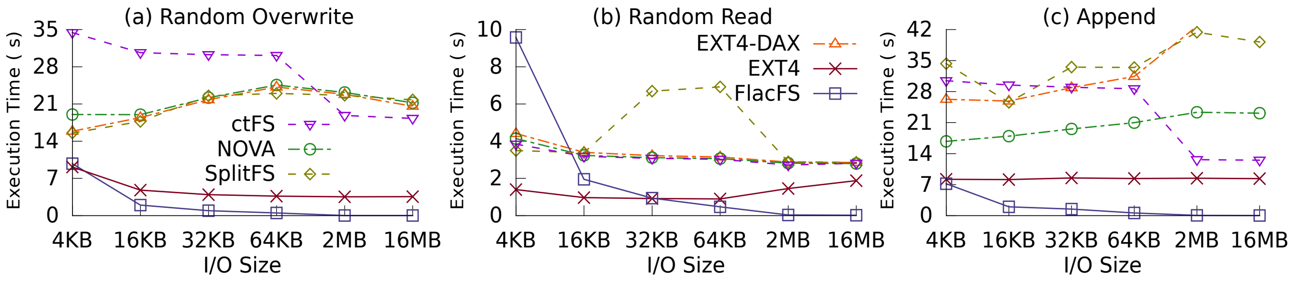


Figure 10: Impact of I/O Size.

time, the concurrency of FlacFS is better than other tested systems. The DAX-based systems are limited by the hardware disadvantages of PM in these experiments. The other cache-based file system, EXT4, is also better than the DAX-based systems because of the locality of the workload, especially in the Webserver case. However, EXT4's performance is still significantly lower than FlacFS because of the inefficiency of the VFS page cache framework.

6.2 Design Analysis

6.2.1 Impact of DRAM Cache Size

Cache size affects the overall performance through two aspects: overhead of cache miss and page eviction.

Cache Miss Overhead. The hit ratio is determined by the cache policy and the workload behavior. As this work mainly focuses on the cache framework design, we just show the performance of the upper (100% hit) and lower (100% miss) bounds. We clear the DRAM cache before each run to evaluate the system performance under cache miss. In particular, write operations in FlacFS do not encounter cache misses in these experiments because the new pages are always attached from the application's DRAM buffer to the FLAC space. By comparing the "EXT4-MISS" and "FlacFS-MISS" in Figure 7 (e) and (f), we found that FlacFS outperforms EXT4 by more than 320 times, which benefits from the asynchronous cache miss handling mechanism. In FLAC, the heterogeneous memory addressing allows pages to be accessed directly whether it is in DRAM or PM, so uncached pages can be attached to the application's read buffer and loaded to DRAM in the background. Therefore, the latency penalty of cache miss is hidden for front-end data I/Os. This design also allows FlacFS to perform better than DAX-based file systems in the cache miss scenario because they need to synchronously copy data from PM to the application buffers.

Eviction Overhead. We append 16GB of data to the files with different eviction thresholds. This experiment is used to measure pure eviction overhead because appending does not have data locality. A smaller threshold means a smaller effective cache capacity and causes more data to be evicted and higher eviction frequency. For example, with 16G threshold, no data is evicted, while half of the data (8G) is evicted at once when the threshold is 8G. The major overhead in eviction comes from copying pages to PM and its performance is bounded by the PM bandwidth. The eviction involves the extra overhead of updating page table entries and invalidating TLB. Figure 9 shows the eviction performance. As expected, a smaller threshold introduces more penalties. For instance, eviction cost under 1G threshold is 2.3 times of 8G threshold, as 1G threshold has nearly twice the amount of eviction data as 8G threshold (15G *vs.* 8G). Additionally, 1G threshold causes more TLB invalidation overhead due to more frequent eviction than 8G threshold.

To sum up the above experiments, we believe that with efficient cache algorithms (out of the scope of this work), FLAC can run efficiently in cache-starved scenarios.

6.2.2 Impact of I/O Size

We evaluate the duration of performing random read/overwrite and append in the I/O sizes ranging from 4KB to 16MB with 64 concurrent threads (no contention). Figure 10 shows that FlacFS has significant advantages compared to other systems when the I/O size is greater than 64KB, because the data copy and migration are the major overheads in these scenarios and this meets the optimization point of FlacFS. For I/Os smaller than 64KB, the advantage of FlacFS decreases as the

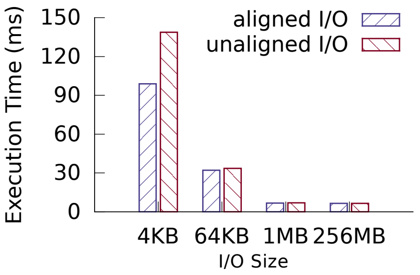


Figure 11: Impact of Unaligned Page.

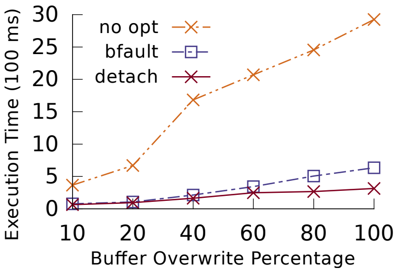


Figure 12: Impact of COW Page Fault.

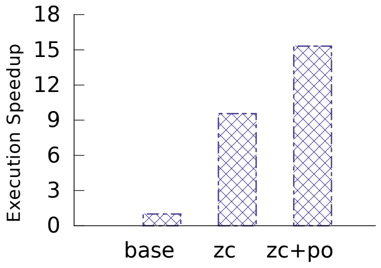


Figure 13: Performance Breakdown.

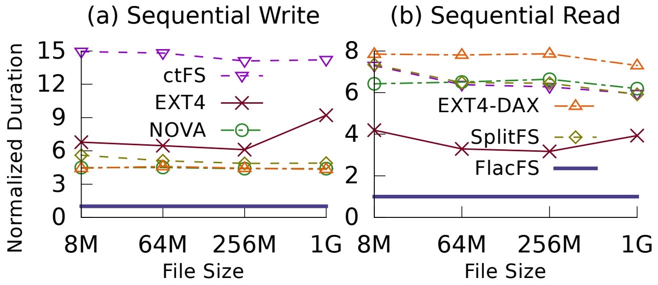


Figure 14: Impact of File Size. For each file size, the duration of other file systems are normalized by FlacFS.

I/O size decreases, as the additional overhead introduced by FlacFS (*e.g.*, TLB flush) becomes apparent in these scenarios. As discussed in § 5, we believe the FlacFS-friendly scenarios can cover a lot of practical workloads. In scenarios where the I/O size is smaller than a page (4KB), they are generally not file system friendly because file systems manage data at a page granularity. Therefore, many real-world applications try to avoid triggering file I/Os smaller than 4KB.

6.2.3 Impact of Page Alignment

FLAC can serve file I/O at any offset and size. The automatic alignment and sliding window buffer are used to solve the page unaligned problem. We evaluate the impact of page unaligned on performance by randomly overwriting 1GB of data in the file under different I/O sizes. Figure 11 shows that unaligned I/Os have a performance degradation of about 20% compared to aligned I/Os when the I/O size is 4KB. However, unaligned accesses have little impact on performance as the I/O size increases, because the amount of data copied by the sliding window buffer does not exceed 4KB, so the proportion of this overhead decreases with the increase in I/O size.

6.2.4 Impact of COW Page Fault

Pages in the application buffer are set to read-only when they are attached to/from FLAC for security and isolation. As a result, COW page faults are triggered when the application updates the data in the buffer for the first time after the FlacFS read/write. FLAC proposes two APIs for batch faulting (`bfault`) and detaching (`detach`) for applications to reduce or eliminate the negative effect of COW page faults. We use 16 test threads to random write on 16 files and rewrite the data in the buffer by using `memset` after each write to evaluate these optimizations (read scenario exhibits a similar performance pattern). Figure 12 shows the results. As the baseline ("no opt"), the test threads simply rewrite the read/write buffers so that normal COW page faults will be triggered. Relatively, the test threads call `bfault` or `detach` for the buffer after each FlacFS read/write to show the benefits of batch faulting and detaching.

The results shows that the total execution time of the baseline grows significantly as the percentage of buffer overwrites increases, because the higher the overwrite percentage, the more COW page faults are triggered, which results in the increased overhead of TLB flushing and data copy. In comparison, batch faulting and detaching can reduce the total execution time by 78.3% and 89.2%, respectively, when the overwrite percentage reaches 100%. Batch faulting reduces the overhead of TLB flushing by aggregating multiple COW page faults. Further, detaching completely avoids COW page faults by remapping new pages to the given addresses.

6.2.5 Performance Breakdown

FLAC includes the key techniques of zero-copy caching and parallel-optimized cache management. As the baseline, we implement a simple FLAC equipped with only a heterogeneous page table and use memory copy to transfer data between the FLAC space and the application buffer. Therefore, both zero-copy caching and parallel optimizations are removed from this simple FLAC.

We use 2 concurrent threads to perform 2MB random write I/Os in this experiment. Figure 13 shows the performance breakdown. In the "zc" case, we add the zero-copy caching design into the baseline but use the coarse-grained lock instead of the 2-Phase flushing (*i.e.*, the front-end I/Os are blocked during the data synchronization). The results show that the performance can be improved by around 10 times by adding the zero-copy caching. In the "zc+po" case, both zero-copy caching and parallel optimizations are applied, and the performance is improved by about 15 times compared to the baseline. For the parallel optimizations, this experiment focus on the contribution of 2-Phase flushing, while the benefits of asynchronous cache miss handling are reflected in §6.2.1.

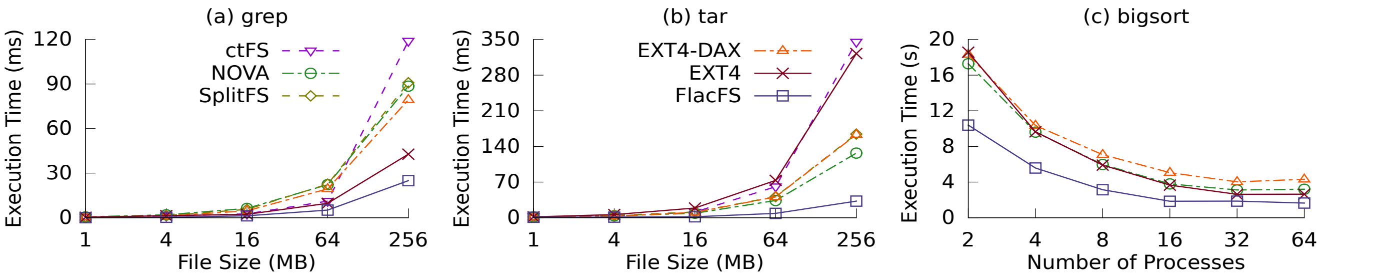


Figure 15: Performance on Real-World Applications.

This experiment shows that FLAC addresses the important bottlenecks of the heterogeneous memory cache framework.

6.2.6 Impact of File Size

We perform 64KB I/Os on different sizes of files (8MB to 1GB) with a single thread to show the impact of file size on the advantages of FLAC. We normalize the duration of other file systems to FlacFS to reflect the fluctuation of FlacFS' performance improvement, *i.e.*, the smoother curve indicates that the file size has less impact on performance improvement. Figure 14 shows that FlacFS has a smooth performance advantage under different file sizes: it has a third of the duration of the second-best system in write and read. The reason is that the performance improvement of FlacFS mainly comes from the zero-copy and parallel optimization, which are not strongly related to the file size.

6.3 Real-World Applications

We evaluate FlacFS in some real-world applications to demonstrate its end-to-end performance benefits. For each application, we replace the file system calls and buffer allocation by the FlacFS' interfaces and sliding window buffer mechanism to port it to our system. In addition, we use batch fault or detach (select based on how the buffer is used) for the read/write buffer to optimize the COW page fault overhead before the application reuses the buffer (if have).

6.3.1 Command Line Application

We port two widely used command line utilities to use FlacFS. The first one is `grep` v3.7. We measure the execution time of matching a character within the input file. Figure 15 (a) shows the performance of increasing file size The `grep` only issues read operations and FlacFS runs 6.7 times faster than the best DAX file system (ctFS) and 4.8 times faster than EXT4 at 1MB file size. The second application is `tar` v1.34. The `tar` contains not only read operations but also contains write to generate the output archive. Figure 15 (b) plots the execution time of creating an archive from the input file. FlacFS still achieves the best performance. With 16MB file, FlacFS gets 4.4 times improvement over the best DAX file system (NOVA) and 9.4 times better than EXT4. Additionally, the computation in `tar` is less expensive than `grep`, which process regulator expression matching. Thus, `tar` spends more time on file I/Os than `grep` and the performance gain in `tar` is more than `grep`. For instance, with 256MB file, FlacFS improves over SplitFS by 3.6 and 5.6 times for `grep` and `tar`, respectively.

6.3.2 Big Data Processing

We evaluate FlacFS and other file systems with BigSort [25], a large-scale merge sort application implemented by Lawrence Livermore National Laboratory. Merge sort is an important phase in big data processing (*e.g.*, page ranking). Given a dataset, BigSort partitions it and performs the merge sorted on each partition recursively. There are three phases in each merge sort: Phase 1) reads the unsorted objects from the target file; Phase 2) performs quick sorting on the objects read in the previous phase; Phase 3) stores the intermediate-ordered results in the file system. After the recursive exit, the global ordered results are written to the output file.

We perform merge sorting on a dataset of 134 million integers. Porting BigSort to SplitFS and ctFS causes multiple processes to hang, so we cannot obtain their performance results. Figure 15 (c) shows that FlacFS has up to 2.62 times improvement compared to other file systems when the number of concurrent processes reaches 64. Benefiting from the zero-copy caching design, FlacFS has a significant performance advantage in Phases 1 and 3 because they include intensive large file I/Os (512KB per I/O). Phase 2 is compute-intensive, and it will incur an unnegligible overhead of COW page fault if nothing is done to optimize it. As a result, FlacFS has an obvious performance advantage in this complex application.

7 Conclusion

Heterogeneous memory provides various advantages, but it also poses challenges to the file system architecture. We analyze the shortcomings of existing cache-based and DAX-based storage frameworks in heterogeneous memory, and conclude that DRAM cache still has great potential in fast all-memory architectures. We propose FLAC, a flat cache framework of heterogeneous memory that integrates the page cache into the virtual memory subsystem. FLAC unlocks the potential of cache through zero-copy caching and parallel-optimized cache management. We implement a file system based on FLAC and show that FLAC has significantly better performance than existing cache and DAX solutions.

Acknowledgments

We thank our shepherd Oana Balmau and the anonymous reviewers for their constructive comments and feedback. We also thank our colleagues in the Huawei OS Kernel Lab for their support. Yuxin Ren is the corresponding author.

References

[1] Chloe Alverti, Vasileios Karakostas, Nikhita Kunati, Georgios Goumas, and Michael Swift. DaxVM: Stressing the limits of memory as a file interface. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO'22)*, pages 369–387, 2022.

[2] Oana Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. TRIAD: Creating synergies between memory, disk and log in log structured key-value stores. In *Proceedings of the USENIX Technical Conference (ATC'17)*, pages 363–375, 2017.

[3] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. SILK: Preventing latency spikes in log-structured merge key-value stores. In *Proceedings of the USENIX Technical Conference (ATC'19)*, pages 753–766, 2019.

[4] Daniel Bittman, Peter Alvaro, Pankaj Mehra, Darrell D. E. Long, and Ethan L. Miller. Twizzler: A data-centric os for non-volatile memory. In *Proceedings of the USENIX Annual Technical Conference (ATC'20)*, pages 1–31, 2020.

[5] Youmin Chen, Youyou Lu, Bohong Zhu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Jiwu Shu. Scalable persistent memory file system with kernel-userspace collaboration. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'21)*, pages 81–95, 2021.

[6] Jungsik Choi, Jaewan Hong, Youngjin Kwon, and Hwansoo Han. Libnvmmio: Reconstructing software IO path with failure-atomic memory-mapped interface. In *Proceedings of the USENIX Annual Technical Conference (ATC'20)*, pages 1–16, 2020.

[7] Johnathan Corbet. Ext4-dax. https://lwn.net/Articles/717953, October 2022.

[8] Mingkai Dong, Heng Bu, Jifei Yi, Benchao Dong, and Haibo Chen. Performance and protection in the ZoFS user-space nvm file system. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'19)*, pages 478–493, 2019.

[9] Peter Druschel and Larry L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'93)*, pages 189–202, 1993.

[10] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System software for persistent memory. In *Proceedings of the European Conference on Computer Systems (EuroSys'14)*, pages 1–15, 2014.

[11] Subramanya R. Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. Data tiering in heterogeneous memory systems. In *Proceedings of the European Conference on Computer Systems (EuroSys'16)*, pages 1–16, 2016.

[12] EleutherAI. GPT3-NEOX. https://github.com/EleutherAI/gpt-neox, October 2023.

[13] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. Hodor: Intra-process isolation for high-throughput data plane libraries. In *Proceedings of the USENIX Annual Technical Conference (ATC'19)*, pages 489–504, 2019.

[14] Intel. 3D XPoint DCPMM. https://www.intel.com/content/www/us/en/products/details/memory-storage/optane-dc-persistent-memory, September 2021.

[15] Intel. Persistent memory development kit. https://pmem.io/pmdk, October 2022.

[16] 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. Basic performance measurements of the Intel Optane DC persistent memory module. *arXiv:1903.05714*, 2019.

[17] Song Jian and Xiaodong Zhan. LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *Proceedings of the ACM Sigmetrics Conference (SIGMETRICS'02)*, 2002.

[18] Myoungsoo Jung. Hello bytes, bye blocks: PCIe storage meets compute express link for memory expansion (CXL-SSD). In *Proceedings of the ACM Workshop on Hot Topics in Storage and File Systems (HotStorage'22)*, pages 45–51, 2022.

[19] Rohan Kadekodi, Saurabh Kadekodi, Soujanya Ponnapalli, Harshad Shirwadkar, Gregory R. Ganger, Aasheesh Kolli, and Vijay Chidambaram. WineFS: A

hugepage-aware file system for persistent memory that ages gracefully. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'21)*, pages 804–818, 2021.

[20] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. SplitFS: Reducing software overhead in file systems for persistent memory. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'19)*, pages 494–508, 2019.

[21] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H Noh, and Young-ri Choi. SLM-DB: single-level key-value store with persistent memory. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'19)*, pages 191–205, 2019.

[22] Juno Kim, Yun Joon Soh, Joseph Izraelevitz, Jishen Zhao, and Steven Swanson. SubZero: Zero-copy IO for persistent main memory file systems. In *Proceedings of the Asia-Pacific Workshop on Systems (APSys'20)*, pages 1–8, 2020.

[23] Miryeong Kwon, Sangwon Lee, and Myoungsoo Jung. Cache in hand: Expander-driven CXL prefetcher for next generation CXL-SSDs. In *Proceedings of the ACM Workshop on Hot Topics in Storage and File Systems (HotStorage'23)*, pages 24–30, 2023.

[24] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A cross media file system. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'17)*, pages 460–477, 2017.

[25] Lawrence Livermore National Laboratory. Bigsort. https://gitlab.com/arm-hpc/benchmarks/coral-2/BigSort, May 2023.

[26] Benjamin C. Lee, Ping Zhou, Jun Yang, Youtao Zhang, Bo Zhao, Engin Ipek, Onur Mutlu, and Doug Burger. Phase-change technology and the future of main memory. *IEEE Micro*, 30(1):143–143, 2010.

[27] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. KVell: the design and implementation of a fast persistent key-value store. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'19)*, pages 447–461, 2019.

[28] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. KVell+: Snapshot isolation without snapshots. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*, pages 425–441, 2020.

[29] Baptiste Lepers and Willy Zwaenepoel. Johnny Cache: the end of dram cache conflicts (in tiered main memory systems). In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI'23)*, pages 519–534, 2023.

[30] Bojie Li, Tianyi Cui, Zibo Wang, Wei Bai, and Lintao Zhang. SocksDirect: Datacenter sockets can be fast and compatible. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM'19)*, pages 90–103. 2019.

[31] Ruibin Li, Xiang Ren, Xu Zhao, Siwei He, Michael Stumm, and Ding Yuan. ctFS: Replacing file indexing with hardware memory translation through contiguous file allocation for persistent memory. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'22)*, pages 35–50, 2022.

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

[33] Yubo Liu, Yuxin Ren, Mingrui Liu, Hanjun Guo, Xie Miao, and Xinwei Hu. Cache or direct access? revitalizing cache in heterogeneous memory file system. In *Proceedings of the Workshop on Disruptive Memory Systems (DIMES'23)*, pages 38–44, 2023.

[34] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit O. Kanaujia, and Prakash Chauhan. TPP: transparent page placement for CXL-enabled tiered memory. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'23)*, pages 742–755, 2023.

[35] Nimrod Megiddo and Dharmendra S. Modha. ARC: A self-tuning, low overhead replacement cache. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'03)*, pages 115–130, 2003.

[36] openGauss. openGauss. https://opengauss.org/, December 2023.

[37] Jiaxin Ou, Jiwu Shu, and Youyou Lu. A high performance file system for non-volatile main memory. In *Proceedings of the European Conference on Computer Systems (EuroSys'16)*, pages 1–16, 2016.

[38] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. IO-Lite: A unified I/O buffering and caching system. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI'99)*, 1999.

[39] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. HeMem: Scalable tiered memory management for big data applications and real nvm. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'21)*, pages 392–407, 2021.

[40] Yuxin Ren, Gabriel Parmer, Teo Georgiev, and Gedare Bloom. CBufs: Efficient, system-wide memory management and sharing. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management (ISMM'16)*, pages 68–77, 2016.

[41] Liana V. Rodriguez, Farzana Yusuf, Steven Lyons, Eysler Paz, Raju Rangaswami, Jason Liu, Ming Zhao, and Giri Narasimhan. Learning cache replacement with CACHEUS. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'21)*, pages 341–354, 2021.

[42] Vasily A. Sartakov, Lluís Vilanova, and Peter Pietzuch. 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)*, pages 546–558, 2021.

[43] Timothy Stamler, Deukyeon Hwang, Amanda Raybuck, Wei Zhang, and Simon Peter. zIO: Accelerating IO-Intensive applications with transparent Zero-Copy IO. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI'22)*, pages 431–445, 2022.

[44] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the XFS file system. In *Proceedings of the USENIX Technical Conference (ATC'96)*, pages 363–375, 1996.

[45] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A flexible framework for file system benchmarking. *USENIX ;login:*., 41(1):6–12, 2016.

[46] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight persistent memory. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'11)*, pages 91–104, 2011.

[47] Jing Wang, Youyou Lu, Qing Wang, Minhui Xie, Keji Huang, and Jiwu Shu. Pacman: An efficient compaction approach for log-structured key-value store on persistent memory. In *Proceedings of the USENIX Technical Conference (ATC'22)*, pages 773–788, 2022.

[48] Yongfeng Wang, Yinjin Fu, Yubo Liu, Zhiguang Chen, and Nong Xiao. Characterizing and optimizing hybrid DRAM-PM main memory system with application awareness. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE'22)*, pages 879–884, 2022.

[49] Hobin Woo, Daegyu Han, Seungjoon Ha, Sam H. Noh, and Beomseok Nam. On stacking a persistent memory file system on legacy file systems. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'23)*, pages 281–296, 2023.

[50] Xiaojian Wu and A. L. Narasimha Reddy. SCMFS: A file system for storage class memory. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'11)*, pages 1–11, 2011.

[51] Jian Xu and Steven Swanson. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'16)*, pages 323–338, 2016.

[52] Jian Yang, Juno Kim, Morteexe Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'20)*, pages 169–182, 2020.

[53] Juncheng Yang, Ziming Mao, Yao Yue, and K. V. Rashmi. GL-Cache: Group-level learning for efficient and high-performance caching. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'23)*, pages 115–133, 2023.

[54] Juncheng Yang, Yazhuo Zhang, Ziyue Qiu, Yao Yue, and K. V. Rashmi. FIFO queues are all you need for cache eviction. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'23)*, pages 130–149, 2023.

[55] Qirui Yang, Runyu Jin, Bridget Davis, Devasena Inupakutika, and Ming Zhao. Performance evaluation on CXL-enabled hybrid memory pool. In *Proceedings of the International Conference on Networking, Architecture and Storage (NAS'22)*, pages 1–5, 2022.

[56] Shao-Peng Yang, Minjae Kim, Sanghyun Nam, Juhyung Park, Jin yong Choi, Eyee Hyun Nam, Eunji Lee, Sungjin Lee, and Bryan S. Kim. Overcoming the memory wall with CXL-enabled SSDs. In *Proceedings of the USENIX Technical Conference (ATC'23)*, pages 601–617, 2023.

[57] Jifei Yi, Mingkai Dong, Fangnuo Wu, and Haibo Chen. HTMFS: Strong consistency comes for free with hardware transactional memory in persistent memory file systems. In *Proceedings of the USENIX Conference on*

File and Storage Technologies (FAST'22), pages 17–34, 2022.

[58] Baoquan Zhang and David HC Du. NVLSM: A persistent memory key-value store using log-structured merge tree with accumulative compaction. *ACM Transactions on Storage*, 17(3):1–26, 2021.

[59] Wenhui Zhang, Xingsheng Zhao, Song Jiang, and Hong Jiang. ChameleonDB: a key-value store for Optane persistent memory. In *Proceedings of the European Conference on Computer Systems (EuroSys'21)*, pages 194–209, 2021.

[60] Shengan Zheng, Morteza Hoseinzadeh, and Steven Swanson. Ziggurat: A tiered file system for non-volatile main memories and disks. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'20)*, pages 207–219, 2020.

[61] Diyu Zhou, Vojtech Aschenbrenner, Tao Lyu, Jian Zhang, Sudarsun Kannan, and Sanidhya Kashyap. Enabling high-performance and secure userspace nvm file systems with the trio architecture. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'23)*, pages 150–165, 2023.

[62] Diyu Zhou, Yuchen Qian, Vishal Gupta, Zhifei Yang, Changwoo Min, and Sanidhya Kashyap. ODINFS: Scaling PM performance with opportunistic delegation. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'22)*, pages 179–193, 2022.