IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 16 August 2022, accepted 23 August 2022, date of publication 29 August 2022, date of current version 6 September 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3202532

RESEARCH ARTICLE

CFFS: A Persistent Memory File System for Contiguous File Allocation With Fine-Grained Metadata

JEN-KUANG LIU AND SHENG-DE WANG, (Life Member, IEEE)
Department of Electrical Engineering, National Taiwan University, Taipei 106319, Taiwan

Corresponding author: Sheng-De Wang (sdwang@ntu.edu.tw)

This work was supported in part by the Taiwan Ministry of Science and Technology under Grant 109-2221-E-002-145-MY2.

ABSTRACT Extensive research on **persistent memory (PM)**-aware file systems has led to the development of numerous methods for improving read/write throughput. In particular, accessing or modifying file contents in a similar manner to the memory operations through *mmap* is a common approach. We designed a file system, CFFS (**C**ontiguous File Allocation with **F**ine-Grained Metadata **F**ile **S**ystem), to rapidly allocate PM pages to upper layer applications for *mmap* and to alleviate page fault overheads due to *mmap*. We optimized the physical contiguity of files in PM to reduce file fragmentation and increase fragment alignment with the goal of reducing software overhead. To achieve this goal, we implemented greedy-based buddy systems and implicit preallocation with a not-most-recently-used (NMRU) policy based on our overall page allocation strategy of considering not only the spatial but also the temporal locality of file access patterns. Furthermore, for efficient and atomic metadata operations, we fully leveraged the byte-addressable property of PM to design fine-grained metadata. CFFS adopts persistent doubly linked lists for directory operations to identify and recover from inconsistencies caused by system failures, doing so without using traditional log mechanisms. In experiments, CFFS showed superior page allocation performance to EXT4-DAX and NOVA did under different PM fragmentation levels. Our allocation algorithm also reduced the cost of page faults for frequently appended files. Finally, CFFS's lightweight directory operations performed excellently in creating and deleting files of various quantities. In summary, the main contribution of the paper is proposing an efficient page allocation algorithm to improve the performance of subsequent *mmap*s, based on the strategy of considering not only the spatial but also the temporal locality of file access patterns in PM file systems. Another contribution was the fine-grained and log-free method for atomic directory operations.

INDEX TERMS Non-volatile memory (NVM), persistent memory, operating system, file system, memory management.

I. INTRODUCTION

Non-volatile main memory (NVMM) technologies such as 3D XPoint [1] are widely applied in rapid persistent storage. For example, Intel has launched its Optane™ DC Persistent Memory [2] based on 3D XPoint (In Q2 2022, Intel announced the ceased development of Optane products [55].). Following this trend, researchers have aimed to use persistent memory (PM) appropriately and efficiently at the software level, particularly for file systems and databases.

Data in PM can be directly accessed by CPU *store* and *load* instructions through the memory bus, different from hard disk drives (HDDs) and solid state drives (SSDs) depending on page caches in dynamic random access memory (DRAM) for software access. This feature of PM is called direct access (DAX). Additionally, byte-addressability is a significant feature of PM, enabling storage to be accessed with single-byte granularity. That is, software treats PM identically to the main memory. However, the performance of PM

The associate editor coordinating the review of this manuscript and approving it for publication was Norbert Herencsar.

remains inadequate despite outperforming HDDs and SSDs. Compared with DRAM, PM has 2–3.7× higher latency and one-third bandwidth in *load* performance and similar latency but one-sixth bandwidth in *store* performance [3]. Therefore, determining which persistent data should be cached in DRAM is a key to improving file system performance.

Typically, file systems [4], [5], [6], [7] initially designed to make greater use of PM fully incorporate PM characteristics such as byte-addressability and DAX. Apart from those specifically designed for PM, some file systems originally designed for HDDs added the DAX feature to compatibly support PM as storage [8]; DAX allows the reading or writing of file data in PM to bypass the DRAM page cache. However, these file systems still remain metadata operations and data structures designed for HDDs and thus lack fine-grained mechanisms.

To enable database applications of PM, new databases with atomic PM key-value store [9], [10] have been proposed, and existing databases have been converted from HDDs or SSDs to PM [11]. By studying these cases, we observed that most databases implement their own atomic data modification mechanisms instead of using the low-layer file system's atomic read/write mechanism [12], in order to avoid kernel traps and kernel buffer copies. Database applications often map PM to the application address space through *mmap* and then access PM through memory operations. Even though some databases abandon atomic read/write mechanisms provided by file systems, file systems are still necessary to allocate PM space as files for management and security.

In addition to databases, some PM libraries are designed to atomically modify data in files [13], [14]. These libraries have their own implementations of *mmap* and *msync* based on low-layer file systems. The *mmap* implementation enables a user to directly load or store data in PM; however, written data does not act on target files until *msync* is called. The *msync* implementation atomically synchronizes written data to a corresponding position in the target file. For example, in SplitFS [13], when *mmap* is called it maps to a staging file, suffering the written data for the target file, and then *msync* modifies some metadata in the target file such that the written data in staging files is atomically linked to the target file. Consequently, SplitFS can write data to files atomically with low copy overhead.

Based on the fact that PM databases or libraries implement their own efficient API for data operations to guarantee atomicity, traditional POSIX *read*/*write* system calls executed by the PM file systems aren't the integral parts for databases or libraries. Nevertheless, the implementation of their data operations makes heavy use of the *mmap* system call provided by file systems. Therefore, the *mmap* performance is a critical issue in file systems for PM.

In this study, we observe that page faults caused by *mmap* [53], degrading the entire system's performance, are related to the page allocation for *mmap*ed files in PM. To solve this problem, we designed a PM file system named CFFS (**C**ontiguous File Allocation with **F**ine-Grained Metadata **F**ile **S**ystem), whose page allocation policy is to reduce the overheads of page faults due to *mmap*. In addition to page allocation, CFFS also pays attention to metadata operations such as file creation and deletion.

In regard to contiguity, traditional HDD file systems attempt to preserve data contiguity because HDDs access data sequentially; initially, contiguity may seem irrelevant to PM because PM supports random data access like DRAM. Consequently, most previous PM file systems focused on how to lower the amount of data written into PM. For example, NOVA is a log-structured file system treating write operations as log entries with an append-only method [7], getting rid of doublewrite overhead caused by redo/undo logging.

Another issue is related to metadata operations. One piece of metadata is a tiny object in file systems, so operations for metadata should be fine-grained. Unfortunately, traditional HDD file systems only have coarse-grained operations due to hardware properties. Previous PM file systems designed fine-grained metadata operations, but they required redo/undo logging, causing doublewrite overhead. We discuss these problems of contiguity and metadata solved by CFFS in the following paragraphs.

First, because PM has lower access latency than HDDs or SDDs, software overhead accounts for a significant proportion of total latency. Moreover, the use of *mmap* is critical in PM applications, and the cost of page faults after mapping PM to a file is correlated with the number of file fragments and the fragment alignment. The number of fragments and the degree of alignment are dependent on the algorithms for allocating PM pages to a file. Intuitively, a file with physically contiguous PM pages has fewer file fragments and better alignment, increasing the performance of page allocation and the page fault handler. We analyze how page fault overhead is affected by these two factors in Section VI. Therefore, CFFS optimizes the physical contiguity of files when allocating PM pages to files.

Second, CFFS not only focuses on the physical contiguity of files, but also provides a high-speed page allocation mechanism [54]. To fit spatial and temporal locality in file access patterns, our allocation strategy involves overall storage distribution in PM and appropriate page candidates for different allocation requirements. We thus designed implicit preallocation and two types of greedy-based buddy allocators, with a not-most-recently-used (NMRU) policy.

Third, although many applications access and modify data by their own functions, they still use file systems to manage PM space in the form of files to achieve access restrictions, user isolation, and attribute management. To provide these functions for applications, CFFS is designed with fine-grained data structures to leverage PM's byte-addressable feature for atomic updates of metadata without a coarse-grained journal mechanism.

Finally, directory operations are keys in data-intensive applications [15]; thus, CFFS used the properties of PM to improve the performance of these operations. We managed directory entries as doubly linked lists, enabling operations

TABLE 1. Page allocation in EXT4: fallocate(software layer) v.s. fsync(storage layer) overhead.

	fallocate	fsync
SSD ratio	0.11	0.89
PM ratio	0.53	0.47

to guarantee data consistency without a log mechanism even after crashing. Moreover, we could reuse the fragments in directory pages with a straightforward method.

II. BACKGROUND AND MOTIVATION

In this section, we present an analysis of software overhead in file systems running on PM and discuss some EXT4-DAX designs that are unsuitable for PM in this section. In Section IV, we propose a file system called CFFS based on this analysis.

A. SIGNIFICANT SOFTWARE OVERHEAD

Previously, file system latency bottlenecks were primarily physical access to storage; thus, the contribution of software latency to total access time was negligible. However, the latency and bandwidth of PM are similar to those of DRAM, and access to PM is faster than access to HDDs or SSDs. Therefore, software overhead accounts for a substantial portion of overall latency.

Table 1 presents the proportions of software and storage layer overhead for page allocation in EXT4 on PM and on an SSD. The system call *fallocate* provided by file systems is used to obtain more pages from storage, and *fsync* is used to synchronize page caches in DRAM to storage. In EXT4, *fallocate* first organizes and updates metadata for page allocation; we set it to not synchronize metadata to storage. Thus, the modification buffered in DRAM is purely at the software layer. Then, *fsync* is called to persist the metadata updated by the preceding *fallocate*; almost all of the time required was used for flushing the modification from buffer to storage. The time required by each operation was used to estimate the ratio of software and the storage layer overheads. On SSD, the software layer overhead accounts for 10% of total latency (on HDD, the software contribution is much smaller). On PM (emulated with DRAM), the software layer accounts for 53% of total latency.

B. FILE FRAGMENTATION OVERHEAD

For HDDs, file systems prefer to keep a file physically contiguous to reduce seek time and rotational latency. Technologies such as preallocation and defragmentation [16] have been developed to achieve physical contiguity. PM has good random-access properties; thus, contiguity seems unnecessary. However, contiguity tends to reduce software overhead.

A file fragment is a region of physically and logically contiguous data blocks of a file; a file may comprise multiple fragments. File systems must manage fragments for each file. For example, EXT4 uses ''extent''s to record information about a fragment, including the logical block offset in a file,

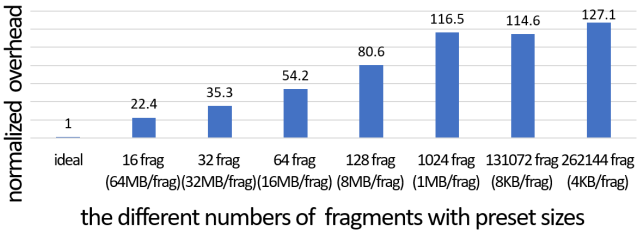


FIGURE 1. EXT4-DAX overhead of prefaulted mmap for 1-GB files with different numbers of fragments. "Ideal" indicates that no other files are in storage, enabling EXT4-DAX to use the fewest possible extents to represent the file. "*x* frag (*y* MB/frag)" indicates that the 1-GB file has *x* file fragments of approximately *y* MB each. All time results are normalized to the ideal case.

the physical block address of the fragment, and the length of the fragment. EXT4 stores those extents as an ''extent tree'' and records the tree information in each file's metadata. (We use the word ''extent'' to describe metadata usage in our paper.)

Data blocks with less contiguity have more file fragments and less alignment. Thus, software overhead increases because file systems must spend additional time searching for the corresponding fragment information in trees with nodes representing a fragment. To add insult to injury, the file system cannot adopt the best page table entry or translation lookaside buffer (TLB) strategy. We describe a case in which *mmap* performance is affected by contiguity.

If a user calls EXT4-DAX's *mmap* to map a file into the application address space and accesses the corresponding address for the first time, page faults occur and trigger EXT4-DAX to translate the logical offset of the accessed file into corresponding PM pages. This process necessitates a search of an extent tree for the correspondence between logical and physical pages; It must traverse the entire tree whose size is correlated with the number of file fragments if the entire file is mapped and all page faults are triggered.

Fragment alignment can cause the page fault handling mechanism to adopt large pages for aligned large fragments to reduce the number of constructed page table entries. Furthermore, a file stored contiguously in PM may have increased TLB performance because multiple TLB entries with virtual and physical address contiguity can be coalesced into a single entry [17], [18].

Fig. 1 presents the measured overhead of EXT4-DAX's *mmap* obtained by mapping then prefaulting a 1GB file with different numbers of extents. Generally, files with higher fragmentation have higher overhead caused by the *mmap*.

C. EXT4-DAX OVERHEAD

To support PM, EXT4 incorporated the DAX feature to become EXT4-DAX. Although EXT4-DAX bypasses page cache in DRAM when a user reads or writes a file and supports DAX *mmap*, its metadata management proceeds with no awareness of PM features. Despite the lack of any PM-specific operation on metadata, EXT4-DAX is worth

exploring because EXT4 has a large user base and has had complete functionality for a long time. Therefore, we analyze EXT4-DAX's software overhead and check whether its operations are suitable for PM.

1) METADATA PERSISTENCE

EXT4 tends to accumulate a certain amount of written data in page cache before flushing a batch of modifications to storage simultaneously to reduce write frequency because frequently writing to a high-latency HDD reduces system performance. Metadata is relatively small, and flushing it on each modification is too expensive. Therefore, when metadata is modified, EXT4 calls JBD2 to perform a transaction rather than immediately making the modification persistent. Until users actively call *fsync* or a periodic system timer passively wakes to trigger JBD2, data in the page cache, including the log entries in journals, is not written back to storage [19].

For example, suppose an application calls *fallocate* to obtain more space for a file. If this application wants to guarantee that the new space is permanent before writing data there, it should call *fsync* after calling *fallocate*. Otherwise, system failure may cause the loss of the allocated space and the data written to that space. Therefore, the application must make an additional system call to ensure data integrity.

This behavior occurs because EXT4-DAX does not leverage PM's ''direct-accessed'' feature. If file systems have a proper atomic update mechanism, metadata can be modified directly without using the page cache or batched writes. That is, it is unnecessary to call *fsync* to ensure that the modified metadata is permanent.

Moreover, EXT4-DAX guarantees the atomicity of metadata through journaling [47]. However, the coarse log mechanism is unsuitable for PM. PM's ''byte-addressable'' feature can prevent the overwriting of entire sectors and can enable precise updates to the modified section. In many cases, only a tiny part of metadata is modified; however, EXT4-DAX overwrites all metadata, such as the inode.

2) PAGE ALLOCATION

When *fallocate* is called, EXT4-DAX attempts to allocate the specified number of PM pages. We present the steps of page allocation and analyze the bottlenecks.

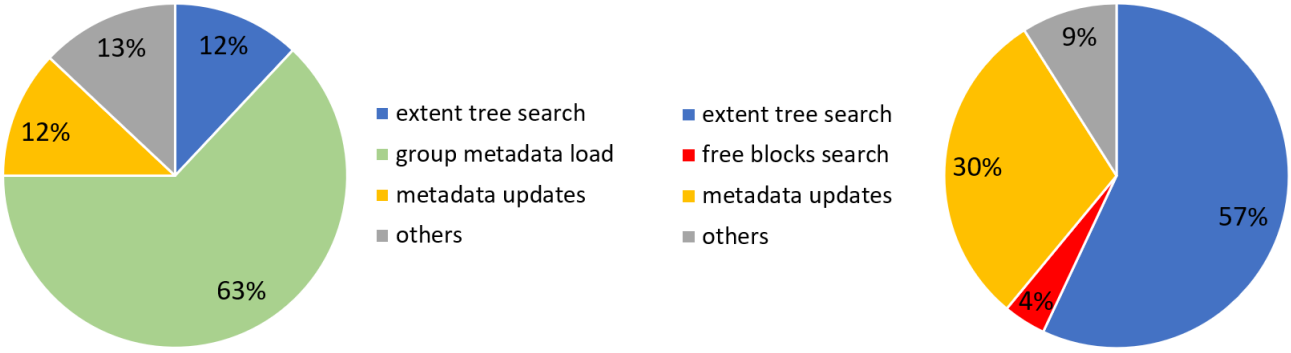
- `Step 1`: EXT4-DAX's *fallocate* must check whether the indicated offset of the file is within the range represented by existing extents. The most recently accessed extent is checked first. If a mismatch is identified, searching the extent tree is necessary; a large tree increases search overhead. However, applications often append new pages to the current file tail contiguously (For example, LMDB's B+ tree is an append-only tree [20]). In this case, the indicated logical offset has not yet been mapped to any physical fragments, and the extent search returns no results.
- `Step 2`: EXT4-DAX's *fallocate* must find a free physical space for allocation; the algorithm is strongly dependent on spatial locality, which is only beneficial for HDDs. EXT4-DAX divides the whole space into numerous groups, and each group uses a bitmap and buddy system to record free pages in the group.
  In Step 1, if the search finds no corresponding extents, the fragment with a logical file offset closest to the indicated offset is found instead. The group in which the physical fragment is located is the first choice for a new allocation because EXT4 attempts to store a file's logically adjacent fragments physically close to each other.
  EXT4-DAX checks whether sufficient pages are preallocated in this group [41] first. If not, the group's bitmap and buddy system are read from storage and written into the page cache. Then, the cached bitmap or buddy system is searched for free pages. If the group does not own enough free pages, the next group is searched in a recursive manner. Thus, the search overhead is variable; in the worst case, it is proportional to the total number of groups.
- `step 3`: After finding available space for a new fragment, EXT4-DAX constructs an extent to record the mapping between the logical file offset and the allocated physical space. The extent must be inserted into the extent tree of the file. If the new fragment is logically and physically contiguous with an existing fragment, the two extents recording their respective information should be merged.
- `step 4`: The aforementioned steps involve the modification of the file and group metadata. This modification only acts on DRAM page cache and is sent to JBD2 for a transaction that provides an atomic update at the checkpoint (*fsync* or periodic system timer) to flush the buffer into storage.

We present the overhead of EXT4-DAX's *fallocate* in Fig. 2. As illustrated in Fig. 2(a), EXT4-DAX allocates a 4-KB page in unused PM to an empty file without any fragments. It first chooses a group and searches for free pages, but the group's bitmap and buddy system have not yet been buffered in page cache and must be loaded from storage. This step makes up over 60% of the total time. As indicated in Fig. 2(b), EXT4-DAX allocates a 4-KB page for a file comprising approximately 20,000 fragments. Searching the file's extent tree requires over half of the total time due to the large number of fragments. Metadata modification is the second-largest contributor to total latency; it requires approximately 20% of the time even though data is not flushed to PM and is only applied to JBD2 for transaction.

D. CFFS CONTRIBUTION

Inspired by the analysis of EXT4-DAX overhead, we designed CFFS with the following techniques.

- CFFS's *fallocate* doesn't need extra *fsync* to persist the allocation, reducing the system call frequency and enhancing the promptness of effective allocation.
- We don't adopt any coarse log mechanism for the atomicity of metadata operations such as *fallocate*,



(a) allocate a page for a new file in unused PM space

(b) allocate a page for a file consisting 20000 fragments

FIGURE 2. Analysis of EXT4-DAX's *fallocate* overhead under different conditions. EXT4-DAX's *fallocate* modifies the metadata for the file, the group, and even the entire file system. It has different execution paths under different conditions. We measured time spent on each subfunction for two cases. In (a), loading group metadata from storage required the most time; searching the extent tree required the most time in case (b).

create and *unlink*. Instead, the tailor-made data structures are designed for metadata management. Specifically, CFFS provides log-free directory operations without compromising atomicity and consistency achieved by tailor-made dentry lists.

- When calling *fallocate*, applications can give CFFS a hint about whether the current page allocation would be intensively appended with logically-contiguous allocation later. Based on the hint, CFFS adopts a suitable allocation algorithm, either growing-size or fixed-size greedy allocator, to optimize file contiguity.
- CFFS maintains the buddy structure with a global view of free space in PM rather than local bitmaps managing a portion of free space. Thus, CFFS's allocation algorithms could select appropriate free pages immediately.

The above analysis demonstrates that fragmentation affects the performance of page faults after *mmap* in EXT4-DAX. In our evaluation, we'll show how the number of file fragments and the degree of fragment alignment affect the performance, and explain why CFFS's contiguity-optimized allocation algorithm results in great performance of page faults.

In this work, we have implemented CFFS running in Linux kernel space. We referenced the codes of some NVM-specific file systems, such as PMFS [6] and NOVA [7], and built up CFFS from scratch. Our design emphasized page allocation and metadata operations, while the *write* operation for files is non-atomic in a similar way to EXT4-DAX. We plan to design a user-space file library to implement the atomic data access for PM files in the future.

III. RELATED WORK

A. FILE CONTIGUITY

Many methods to improve file contiguity have been developed for HDD or SSD file systems, and we surveyed some works for designing suitable methods for PM file systems. First, defragmentation is the most common method. Park *et al.* proposed a file defragmentation scheme on log-structured file systems [50], addressing the severe fragmentation caused by log-structured methods to enhance sequential read performance on SSD. The scheme determines the fragmentation degree of every segments, and migrates the data in the highly fragmented segment to a new segment. Second, writing data to storage in batches is also utilized extensively in file systems. BTRFS is a file system developed by Oracle [51], accumulating data in page cache until the next checkpoint and then writing data to as few fragments as possible on disk. Third, great allocation policy would determine the contiguity of files. Kesavan *et al.* proposed the write allocator in the WAFL file system [52], considering information of allocation area when allocating blocks to a file.

These works enhancing file contiguity were for fitting the hardware characteristic beneficial to sequential access. That is, absolute contiguity is unnecessary. Thus, the contiguity here means that the data in a file is stored around the nearby location. In contrast, CFFS's contiguity improvement is for reduction in the software overhead, because sequential access has a relatively slight impact on PM. CFFS focuses on strict contiguity for great fragment alignment and the low number of fragments, so we have to figure out a method different from previous works for HDDs and SSDs.

B. PM DIRECTORY OPERATION

Directory operations—*create*, *mkdir*, *unlink*, *rename*, and others—may modify multiple inodes. For example, *create* allocates a new inode and inserts a directory entry (containing at least the name of the created file and its inode number) into a directory page at the same time. Hence, atomic directory operations are necessary. Suppose we successfully add a directory entry but fail to include the correct value in the metadata of the new inode due to a system failure. In that case, a directory entry pointing to an inode is nonexistent or incomplete. Conversely, an intact inode with no directory entry pointing to it could exist. Both cases cause inconsistency [23].

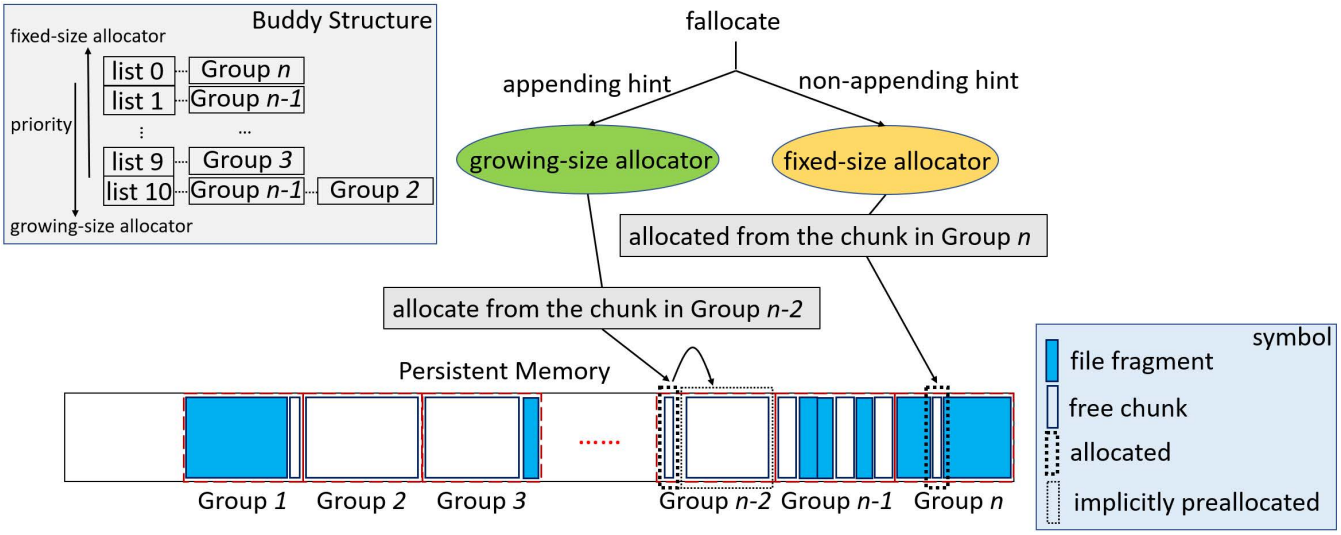


FIGURE 3. Allocation overview.

To guarantee atomicity, ext4-DAX uses a coarse journal mechanism to log modifications of inodes [24]. NOVA used the log-structured property to implement directory operations [26]. However, an additional logging mechanism is still required to ensure that it can both modify multiple inodes and recover from inconsistency caused by a failure.

Logging causes double write overhead. Moreover, NOVA's directory operations have some performance bottlenecks. For example, NOVA records the tail directory entry in the directory inode, contributing to modification not only on the directory page but also on the directory inode's metadata in PM. NOVA's log structure also generates a great number of log entries, resulting in frequent garbage collection and PM page allocation.

Other important issues are out-of-order execution and consistency between cachelines and PM [49]. The first problem is that instructions may be executed in an order optimized by CPU rather than the original order of the program. It is possible to reorder *load*/*store* instructions for PM access, causing inconsistency. The instructions to execute metadata operations should follow their fixed sequences, so memory barriers (*sfence* in x86) [27] are essential for metadata operations. The second problem is that the CPU accesses PM through cachelines because there is no difference between PM and DRAM from the perspectives of CPU. The data kept in cachelines without persisting in PM also causes inconsistency. To solve this problem, developers should use the instructions to flush the data out of cacheline for persistence guarantee (*clflush* in x86).

IV. CFFS PAGE ALLOCATION

Our goal is to allocate pages quickly and increase the contiguity of growing files. We design two types of greedy-based buddy allocation systems and an implicit preallocation mechanism to reduce the potential software overhead. To allocate pages, CFFS first attempts to use preallocated pages to prevent the creation of new fragments. If preallocated pages are insufficient, the buddy allocation system is used for allocation. CFFS supports 4-KB or 2-MB page sizes. The allocation algorithms are similar, so we only describe for the 2-MB case in the following section for brevity.

We illustrate an overall architecture for the CFFS allocation algorithm in Fig. 3. PM is divided into multiple groups, and Buddy Structure is a set of free lists to manage these groups. Each list is in charge of some groups according to the size of free chunks in the group. Once the system call *fallocate* is invoked by applications to request for page allocation, CFFS identifies this allocation as one with the appending or non-appending hint.

If the allocation is with an appending hint, the growing-size allocator selects the group managed by the highest order buddy list and allocates pages from the largest free chunk in this group. So far, the new file fragment has been created, and the adjacent free chunk on its right has been implicitly preallocated for the appending-typed allocation next time. If the allocation is with a non-appending hint, the fixed-size allocator selects the group managed by the lowest-order buddy list for allocation and no implicit preallocation.

A. SPACE MANAGEMENT

1) FILE LAYOUT

Each file maintains an "`CFFS_inode`" in PM. Some metadata in the inode representing the mapping between the logical file offset and the physical PM address is essential to enable applications to access data by providing logical file offsets to the file API.

We present the use of this metadata for constructing the mapping through an example in Fig. 4. From the application's perspective, fragment ①②③ constitutes a contiguous space in the logical file. However, this space is not

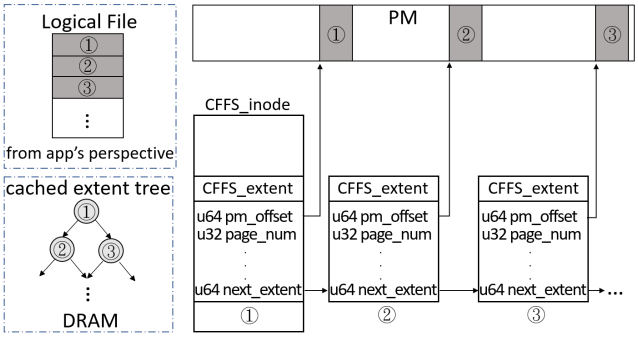


FIGURE 4. **File Layout. Each `CFFS_extent` records a file fragment's physical information. All `CFFS_extents` belonging to the file are connected as a list to represent the file's logical layout, and the first `CFFS_extent` of the file is embedded inside its `CFFS_inode`.**

physically contiguous in PM and comprises three fragments at different PM addresses.

CFFS uses an "`CFFS_extent`" in PM to record file fragment information, including the physical PM address and length, and the three `CFFS_extents` indicated in the figure are linked as a list in the sequence of their logical offsets. CFFS also caches extent information in DRAM and constructs an extent tree of the cached extents for each file to expedite the translation of a logical offset to a PM address.

All `CFFS_extents` of a file are connected as a persistent singly linked list. This linked list guarantees consistency easily due to its structure, but it is unsuitable for searching. Because PM latency is longer than DRAM latency, CFFS constructs cached extent trees in DRAM to expedite searches at the cost of acceptable construction overhead. By contrast, EXT4-DAX maintains extent trees in storage, and the B+-tree structure requires greater overhead to guarantee consistency. Adopting the B+-tree in storage is suitable for HDDs because multiple extents can be stored in the same sector to reduce hardware overhead. PM has better random access performance; thus, the list structure is a feasible option for organizing extents.

2) PHYSICAL LAYOUT

As presented in Fig. 5, each part of the 1-GB-aligned PM space is a group comprising 512 huge pages (2 MB in size). A group is the management unit for selecting proper free space with the NMRU policy; this policy is described in a Section IV-C.

The remaining space that cannot be aligned to the 1 GB boundary is used to store inode tables, extent tables, directory pages, and small file fragments. The unit of this space has a 4-KB page size. When this space is exhausted, some huge pages are extracted from groups to supplement it; we name these pages support pages.

3) GROUP

Fig. 6 presents the group structure management and the free pages in a group. CFFS uses a "`cached_extent`", the DRAM abstraction of a file fragment (including its address

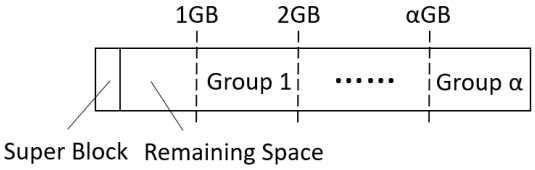


FIGURE 5. **PM Layout. The first 1 GB is the superblock and is not part of a group.**

and length). Using `cached_extents`, an extent list links all file fragments in the group in accordance with the order of the physical fragment offsets. Contiguous unused pages constitute a free space fragment, and we call it a free chunk. CFFS uses "`free_chunk`" as the DRAM abstraction of a free chunk. A free list links all free chunks in the group in size order. Generally, nodes closer to the head of a free list have larger chunk sizes.

One file fragment is always adjacent to one free chunk on its right side with a higher offset in the group (e.g., ① ❶ and ② ❷ in Fig. 6); these fragments always exist in pairs. Thus, we bind `free_chunk` to the `cached_extent` structure. This process is the key idea in implicit preallocation.

4) FREE CHUNK MERGING

If an application deletes all or part of a file, the corresponding file fragments must be released. In the example illustrated in Fig. 6, if fragment ① is released, this space should be merged with free chunk ❶ and ❶ as a larger free chunk; thus, merged `chunk_len` becomes a+b+c. Note that the order of the free list would be adjusted if a+b+c is larger than e.

5) SP FILE

`CFFS_inode` and `CFFS_extent` are both structures in PM. We use inode tables and extent tables to manage these structures. The tables comprise multiple 4-KB pages that are allocated from the space not affiliated to any groups unless this space is exhausted. If the space is exhausted, support pages allocated from groups could resolve deficiencies of pages for inode tables, extent tables, directory pages, or small files. A special file, the support page file (SP file), is used to manage all support pages and for the timely release of unused support pages. Unlike general files, the SP file does not require contiguity; thus, we prefer to allocate small chunks in groups as support pages.

B. IMPLICIT PREALLOCATION

Because new data is appended to files in most data-intensive applications, file systems require efficient mechanisms to allocate pages for growing files. In one method for preallocation, applications request sufficient space at each call to reduce the number of system calls required to allocate pages from file systems; however, applications may not precisely predict space demands. In another method, file systems explicitly preserve pages physically close to existing file fragments; this method is most useful for HDDs.

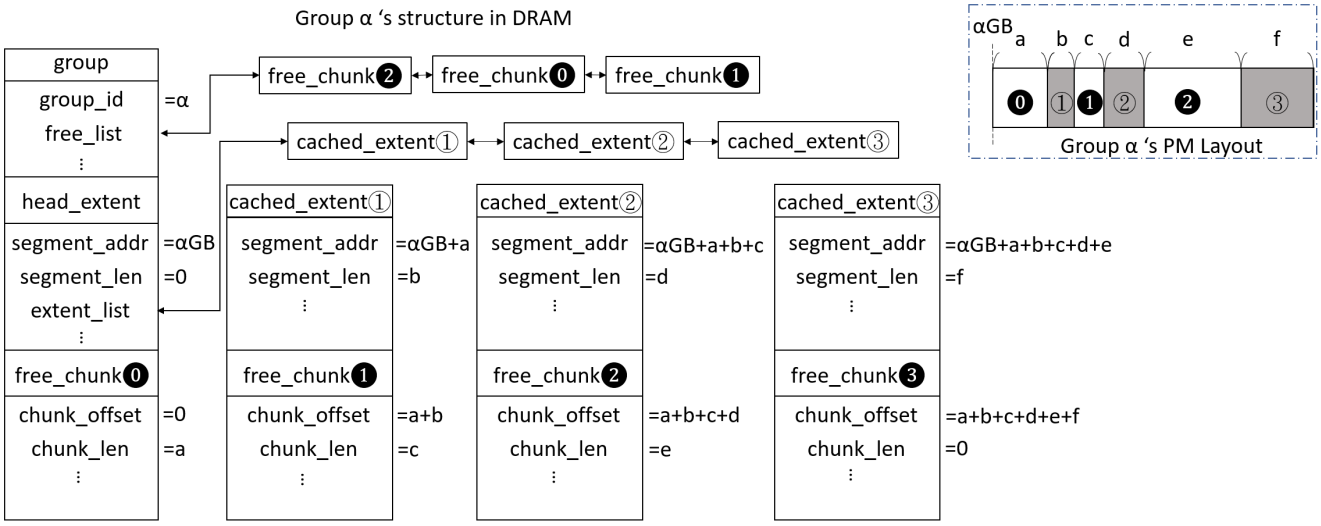


FIGURE 6. **Group Structure. Group α contains free chunks 0 1 2 (their order in the free list is 2 0 1) and file fragments ① ② ③ (their order in the extent list is ① ② ③). ① 1 and ② 2 are two pairs. Yet, 0 at the head position of group α has no partner fragment; thus, it is bound to the `head_extent`, which does not represent any fragment but contains the list head of the extent list. ③ at the tail position has no partner chunk; thus, it is bound to a `free_chunk` with length 0.**

Preallocation in CFFS has two goals. The first is to provide a fast allocation path that requires no search for the global buddy allocation system and only makes minor modifications to metadata in PM. The second is to implicitly reserve pages for file fragments that are likely to extend to reduce the number of fragments and potential software overhead.

1) FRAGMENT EXTENSION

When an application requests for CFFS to allocate new PM pages to an indicated logical file offset, CFFS searches the cached extent tree to identify the fragment with a logical offset smaller than but closest to the required offset and then checks whether the found fragment's offset is logically contiguous to the required offset. If so, CFFS tends to allocate the free PM pages to be physically contiguous to the found fragment, preventing the creation of new file fragments.

File fragments and free chunks exist in pairs, and we can access a `free_chunk` through the `cached_extent` returned from the prior tree search due to the design of the bound structure; thus, we obtain the information of the free chunk on the right side of the identified fragment. Suppose the size of this free chunk is sufficiently large. In that case, page allocation is performed by fragment extension, which is easily completed by modifying a few fields in `cached_extent` and `free_chunk` (in DRAM) and atomically persisting the fragment's length and timestamp by recording in PM. Otherwise, allocation is performed from the buddy system detailed in Section IV-C.

2) EXTENSION TREND

A free chunk is implicitly preallocated to the closest file fragment on its left side. A free chunk preallocated for a growing file fragment should not be selected by the buddy system for allocation to another fragment; therefore, CFFS must learn fragment extension trends.

If a fragment extends its size, jointly, a corresponding free chunk implicitly preallocated for it must decrease in size. CFFS maintains a free list for each group and links all free chunks in the group according to their size. Therefore, if a free chunk decreases its size due to fragment extension, the free list to which this chunk is linked might adjust its sequence. If such an adjustment occurs, the decreased chunk has a lower allocation priority in the group (the buddy system allocates the largest chunk in the selected group). If the decreased chunk is still the largest chunk in the group, this adjustment does not occur. However, the entire group is linked to an extension least-recently-used (LRU) list; the buddy system does not allocate free chunks that belong to groups linked to this list.

C. GREEDY-BASED BUDDY SYSTEM

1) VOLATILITY

CFFS does not maintain scanning-inefficient bitmaps but instead maintains a buddy structure in DRAM. It writes back to PM only during the unmounting procedure for rapid remounting. If the buddy structure is lost due to a crash, it can be recovered by scanning inode tables to identify valid `CFFS_inodes` that record the pages occupied by the files, including the SP file. Thus, the buddy structure does not risk storage inconsistency.

2) BUDDY STRUCTURE

CFFS's buddy structure is used to manage free chunks according to their sizes (the number of 2-MB pages), and the management unit is a group. Fig. 7 illustrates a buddy structure. It has 11 buddy lists, and each list contains chunks in a

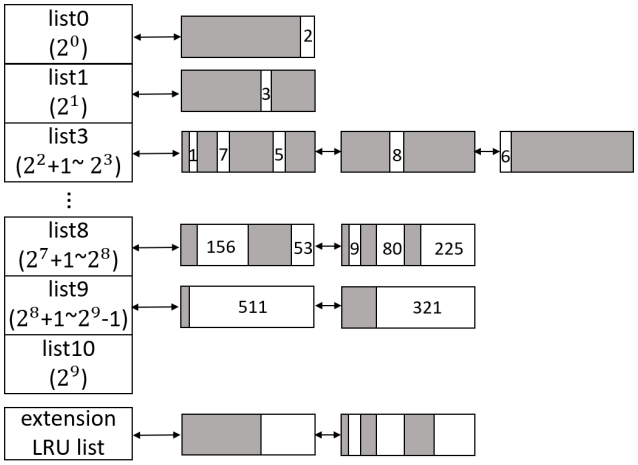


FIGURE 7. **Buddy Structure. The structure has 11 buddy lists and an extension LRU list; these manage all groups in PM. list0 to list10 manage groups ordered by the largest chunk size in each group. For example, the groups linked to list9 own at least one free chunk consisting of 257 to 511 huge pages but fewer than 512 huge pages.**

specific size range. Due to our greedy strategy, determining which buddy list a group belongs to is based on the size of the largest free chunk in the group. Besides, the extension LRU list prevents free chunks reserved for recently extended file fragments from being selected by the buddy allocator for allocation. If the number of groups linked to the LRU list exceeds a threshold, evict the group in which the fragment least-recently extended the size.

3) GROWING-SIZE GREEDY ALLOCATOR

The growing-size greedy allocator is used to allocate new fragments for extension. It is designed to both allocate new fragments and implicitly reserve free pages for the future extension of the newly allocated fragment; thus, it must allocate a position with substantial reserved space. This greedy allocator selects a nonempty buddy list with the current highest order in the buddy structure and rapidly identifies the first group connected to this buddy list. Then, it selects the largest free chunk in the group. This free chunk is larger than chunks in other groups linked to lower-order buddy lists and is thus the target for allocation. To determine which part of this chunk should be allocated, we consider the three situations as follows.

First, suppose that the requested page number is greater than or equal to the size of the selected chunk. In this case, the whole chunk is allocated. Second, if the selected chunk is at the head of the group, no fragment in the same group exists on its left. In this case, left-aligned pages in the free chunk are allocated, and the remaining free pages are implicitly reserved as in preallocation.

In the third situation, an existing fragment is contiguous to the selected free chunk's left side (the selected chunk and the existing fragment are bound as a pair). In this case, we consider whether this existing fragment is a fixed-sized fragment with no chance of extending its size. If the existing fragment has a fixed size, left-aligned pages in the free chunk are allocated, resulting in no reserved pages for the existing fragment. Otherwise, we implicitly reserve free pages for both the new fragment and the existing fragment by allocating central pages in the free chunk, resulting in the same number of free pages implicitly reserved for both the new fragment and existing fragment. However, if the new fragment does not extend its size substantially, allocating pages from the middle of the free chunk prevents the existing fragment from attaining maximal contiguity. To resolve this problem, CFFS uses application hints and defragmentation.

In all of these situations, new fragments or split free chunks are generated. Thus, the selected group's extent list and free list are adjusted. The buddy system also must be adjusted in accordance with the group's latest largest chunk and the NMRU policy.

An example is presented in Fig. 8(a). In this example, an application requests one huge page as a new fragment. This fragment is expected to extend; thus, CFFS adopts the growing-sized allocator and selects list9's first group whose largest free chunk owns 511 huge pages. The allocator then determines whether the fragment on the chunk's left side is a fixed-sized fragment. If not, centrally aligned allocation is performed on this chunk and the group is migrated to list8's tail. Otherwise, left-aligned allocation is performed and the group is moved to list9's tail.

4) FIXED-SIZE GREEDY ALLOCATOR

By contrast with the grow- ing-size allocator, the fixed-sized greedy allocator reserves no preallocated pages for new fragments; instead, it preserves continuity for other existing fragments. For some allocation cases, continuity is irrelevant because the allocated fragments will not increase in size. For these fragments, applications give hints to *fallocate* indicating that the size of the required fragment is fixed. Moreover, fragmentation does not affect certain types of files. For example, the SP file, comprising support pages and only used by CFFS itself, links all the pages as a list. Most operations on this file, except for linear traversal for crash recovery, take constant time regardless of the number of fragments.

In these cases, CFFS uses the fixed-sized greedy allocator. It selects a nonempty buddy list with the current lowest order and rapidly identifies the first group connected to the list. The low-order list is chosen to preserve large free chunks for the growing-size greedy allocator; moreover, the selected free chunks are relatively small; thus, the existing fragment bound to that chunk has little preallocated pages to gain. That is, if a free chunk owns x pages and an existing fragment uses them as preallocated pages, then the number of file fragments due to extensions can be reduced by at most x; a smaller x results in fewer potential gains. CFFS thus selects a free chunk in the selected group and allocates right-aligned pages in the chunk. No page is reserved for the new fragment; the remaining pages are reserved for the existing fragment.

An example is presented in Fig. 8(b). An application asks for one huge page as a new fragment. Due to a non-appending

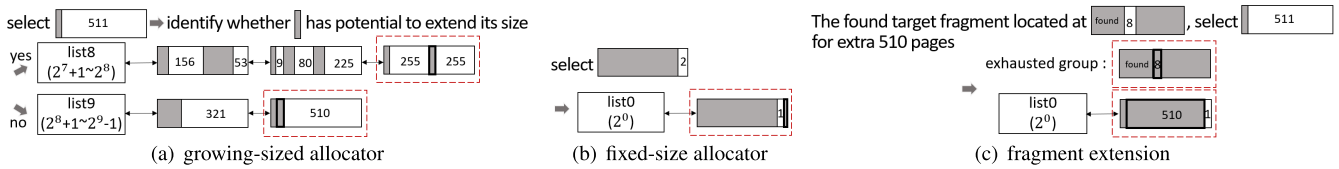


FIGURE 8. **Allocation Example. We present the allocation steps using the growing-size greedy allocator, fixed-size greedy allocator, and fragment extension. The examples use the same buddy structure as in Fig. 7.**

hint, CFFS adopts the fixed-size allocator and selects list1's first group. The allocator executes right-aligned allocation on this chunk and migrates this group to list0's tail.

5) FIXED-SIZED FRAGMENT

Fragment extension may generate fixed-sized fragments under certain conditions other than through the fixed-size allocator. Fig. 8(c) illustrates fragment extension for a single file; 518 huge pages are appended after an existing fragment. First, CFFS searches the cached extent tree to identify the fragment receiving an extension; however, its implicitly pre-allocated eight huge pages is insufficient. Thus, in addition to fragment extension, the growing-sized greedy allocator also allocates an additional 510 huge pages from the first group connected to list9. The free chunk with 511 huge pages in this group is reduced to 1 huge page, and the group then migrates to list0's tail. In this case, one logical-contiguous allocation is split into two physical file fragments, and the existing fragment extended by eight huge pages becomes a fixed-sized fragment.

6) NMRU

Typically, file system access has strong temporal and spatial locality [21], [22]. If a file fragment has recently extended its size or been newly allocated, extension is likely to happen repeatedly in the short run. Therefore, the free chunk implicitly reserved for this fragment should not be allocated for new fragments by the buddy system. To achieve this goal, three levels of NMRU are included in CFFS's buddy system.

The first level is inside the buddy structure. Note that the buddy list with the current highest order is used by the growing-size greedy allocator to pick a group. After centrally aligned allocation, the original largest chunk is cut in halves, and the selected group would be migrated to a buddy list with a lower order according to the size of its new largest chunk. Consequently, if a group is picked by the growing-sized greedy allocator, migration to a lower buddy list reduces its precedence for growing-sized greedy allocation. Moreover, if fragment extension causes a group to be isolated in the extension LRU list, the group will not be selected by the allocator until it is removed from the LRU list. Thus, fragment extension also satisfies the first level of NMRU.

The second level is inside the buddy list and the extension LRU list. In the first level, we see that a group migrates to the extension LRU list or a lower buddy list, and a migrated group is always added to the target list's tail. If a group does not need to migrate to another list, it is still moved to the tail of the original list. For buddy lists, the allocator always picks the first group connected to the selected buddy list. For the extension LRU list, the first group is returned to the buddy list when the extension LRU list is full. That is, the group at the tail position is the last choice for the allocator in the same buddy list or extension LRU list.

The third level is inside a group. If a group is selected by the buddy system for allocation, the largest free chunk in the group is selected. That is, allocating a new fragment or extending an existing fragment decreases the size of a free chunk and thus decreases its allocation priority within the group.

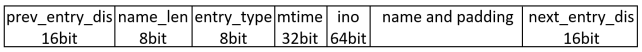


FIGURE 9. **Dentry format.**

V. CFFS DIRECTORY

CFFS leverages PM's byte-addressable property and considers PM's access granularity [48] to design suitable directory structures. Moreover, we implemented some simple operations, such as *create*, *mkdir*, and *unlink*, with a log-free method with memory barriers (*sfence* in x86) to ensure operation sequences and the 64-bit memory atomic *store* [27] (guaranteed by x86) to avoid logging. Even after a crash, CFFS can recover from inconsistencies without additional recovery processes. Finally, CFFS reuses fragments in directory pages generated by *unlink* or *rmdir*.

A. CFFS DIRECTORY STRUCTURE IN PM

1) DIRECTORY PAGE

A directory contains multiple pages linked as a list, and the address of the first page (4 KB per page) is recorded in its `CFFS_inode`. (`CFFS_inodes` for directories and files may have discrepancies because directory pages do not require continuity). The beginning of each directory page includes a page header that records the address of the next directory page, the offset of the first directory entry in this page, and other information. The remaining space is used for directory entries; we allow a fragment to exist between two consecutive entries.

2) DIRECTORY ENTRY

Fig. 9 presents the information stored in a directory entry (dentry). Fields storing an inode number and name are required; moreover, a field is used to record an entry type (e.g., file, directory, or symbolic link) and some bits are

reserved as flags for atomic *unlink*. Unlike other methods, our method also records the distance from an entry to its previous and next entry (i.e., the entry on its left or right, respectively) in the `prev_entry_dis` and `next_entry_dis` fields; it is meaningful to place these two fields on the far left and far right, respectively, of an entry.

To reduce the length variations, each dentry is padded to 8-bytes aligned to facilitate fragment management and increase directory compactness. The minimum length is 24 bytes, and the maximum is 280 bytes, primarily depending on the length of the file name.

3) DENTRY LIST

Because a dentry records the distance to its previous and next entry, all entries in one directory page form a dentry list whose head is the first dentry in the page. Moreover, one directory may own multiple directory pages, forming multiple dentry lists. Therefore, we must have a method of connecting dentry lists established in different directory pages to traverse all entries.

To achieve this connection, the last entry in a directory page is marked as `CROSS PAGE` or `END`. The last entry does not have a next entry in the same page; thus, we use its `next_entry_dis` field to mark the final page with `END`. Otherwise, the page is marked with `CROSS PAGE`, indicating that the directory traversal procedure should continue to the next page.

A dentry list is similar to a doubly linked list, except for the last entry of the list using the `next_entry_dis` field for a special mark. Doubly linked lists in PM have consistency problems [28], [29]. We use the problems in reverse to determine which entries stay in inconsistent states due to crashes. Suppose that two entries in the same page link to each other; the previous entry's `next_entry_dis` value should be equal to the next entry's `prev_entry_dis` value in a consistent state. If these fields are unequal, an inconsistency exists, and the operations broken by crashes can be recovered via the information in these two entries.

4) FRAGMENTS IN DIRECTORY PAGES

A dentry may be invalidated by operations such as *unlink* and *rmdir*; invalidated entries in directory pages become fragments. If two connected entries have no fragments between them, the previous entry's `next_entry_dis` and the next entry's `prev_entry_dis` value should be 0. Otherwise, these values should be equal to the length of the fragment. Reusing fragments to store new directory entries is an important feature of CFFS that can reduce page consumption and the frequency of directory compaction.

B. CFFS DIRECTORY STRUCTURE IN DRAM

1) DENTRY CACHED TREE

Although a dentry is stored in PM, we cache its name, inode number, and address as a node in DRAM, and then build a dentry cached tree consisting of these nodes for each directory to transfer a file/directory name to an inode number without searching dentry lists in PM.

2) FRAGMENT BUCKET

Knowing that a fragment might exist between two linked dentry, we manage fragment information to reuse them. Because we can extract all information about fragments from dentry lists, information regarding a fragment is only stored in DRAM as a fragment node.

We maintain a set of fragment buckets for each directory, and we put each fragment node into a corresponding bucket according to its length, which is also aligned by 8 bytes because a fragment stems from invalidated dentry. Then, we have to determine the number of buckets in a set. Of course, the use of more buckets allows fragments to be classified in greater detail, but this comes at the cost of more DRAM consumption. Currently, we maintain only four buckets in a set to manage fragments with lengths of 24, 32, 40, and >40 bytes, respectively, because most file/directory names in common workloads are not too long.

3) TAIL ENTRY

If no suitable fragment is present for a new dentry to be filled, we must add it behind the tail entry, whose `next_entry_dis` value is `END`; then, the new dentry becomes the tail entry. Therefore, recording the address of a tail entry is necessary. We record it in DRAM rather than PM, unlike NOVA, because we can locate a tail entry whose `next_entry_dis` value is a recognizable special mark in a dentry list.

C. CFFS DIRECTORY TIMESTAMP PROBLEM

The timestamp plays a vital role in databases and servers [33], [34]. Most file systems persist atime, ctime, and mtime for files or directories in storage as metadata inside inode. Some directory operations would update timestamps. Take *create* as example, file systems not only complete a newly created file's inode and add a new dentry into a directory page but also update the parent directory's mtime. Thus, consistency among these three modified regions should be guaranteed. In CFFS, the timestamp could be released from this consistency by attaching the `mtime` field in the directory entry and timestamp properties, which are discussed as follows.

1) MONOTONICALLY INCREASING

OS usually adopts Network Time Protocol(NTP) for clock synchronization between different machines [35], which might adjust system time backward. It is fatal to servers and databases because transaction timestamps are disorganized and may break the consistency models of client's requests. Consequently, most databases and servers do not allow one to turn back system clocks. Instead, they slew the clock-that is, time is still increased but the rate of increase is slowed down [36]. In addition, timestamps generated by different cores can achieve global ordering by ORDO [44].

If system administrators obey the aforementioned rules, timestamps with file operations are naturally monotonically increasing. We consider this property to design CFFS's metadata updates related to timestamps more efficiently. If administrators do not care about backward steps in time, we must use stricter methods to update timestamps.

2) MOST RECENT TIMESTAMP

As mentioned, because CFFS attaches a timestamp to each dentry for directory operation, we do not have to modify `CFFS_inode`'s `i_mtime` field frequently and simply synchronize directory mtime at sparsely distributed checkpoints. NOVA also uses this technique to maintain directory timestamps. Because the tail entry in each NOVA directory page is most recently appended, it makes sense to use the tail entry to recover directory timestamps after crashing.

However, because CFFS reuses fragments in directory pages, the tail entry does not represent the most recent entry in a directory. Nevertheless, based on monotonically increasing timestamps, the most recent entry could be identified by the largest timestamp. If this supposition is not valid, we can add an `epoch` field in dentry. Among all timestamps attached to entries in a directory, the largest one is the valid mtime. In case of crashing, CFFS traverses all entries in a directory to build a dentry cached tree and drop by to find out the largest timestamp or epoch.

D. CFFS DIRECTORY OPERATION

1) CREATE/MKDIR

These operations insert a new dentry into a dentry list in a directory page and allocate a new `CFFS_inode` filled with intact metadata. We combine these two items as one atomic operation. If the operation fails, CFFS can recover from an incomplete state. As the aforementioned explanation for dentry list, each directory page contains a dentry list composed of every page-affiliated dentry. The `prev/next_entry_dis` fields of a dentry record the distance to its previous and next entry inside the same directory page. Such 16-bits field is analogous to the pointer pointing to its previous or next entry so that a dentry serves for a node in a list.

As we all know, inserting a new node into a list necessitates updating some nodes already existing. We leverage this fact ingeniously to manipulate some fields of associated entries in the well-designed specific order whereby these existing entries carry the information of a new entry. Thus, even if the insertion fails incurred by system-crashes, the recovery procedure can detect the inconsistency, and refine the necessary information from existing entries to undo all modification caused by failed insertion, without inefficient logging mechanisms.

We divide one atomic *create* or *mkdir* operation into three main parts as follows: (i) finding space and filling a new dentry, (ii) allocating a new `CFFS_inode` and initializing it, and (iii) inserting a new entry into the dentry list. Because the detailed steps and execution order inside one operation subtly differ according to the new entry's position in the directory pages, looking for space in the directory pages for a new entry is the first step.

CFFS prefers to reuse fragments as new entries; thus we first look up fragment buckets to identify a fragment whose length is equal to or greater than what the new entry requires. If no suitable fragment is identified, we attempt to use the space following the tail entry in the last directory page. If that space is inadequate, we allocate a new directory page. In summary, the three new entry positions are type a, in a fragment; type b, in a new page; and type c, behind the tail entry in the last page. Fig. 10 presents an example to illustrate the different *create* procedures for these three situations.

For type a, a new entry needs inserting between two entries in a dentry list. These two entries in the same directory page are on the left and right of the selected fragment, called the previous and next entry, respectively. Step1 initializes a new entry with metadata (inode number, file name, etc.), and its `prev/next_entry_dis` fields unilaterally point to its previous and next entry. Then, Step2 makes the next entry's `prev_entry_dis` field point to the new entry via the 64-bit atomic *store*, starting the dentry list insertion. The insertion completes once Step4 is finished, making the previous entry's `next_entry_dis` field atomically point to the new entry. Before Step4, Step3 should enable a new `CFFS_inode` with intact metadata of the new file for the current *create*.

We elaborate on the inconsistency induced by system-crashes happening in different steps for type a. Step2 and Step4 are both one atomic instruction, so we use them to demarcate the boundaries (before Step2, after Step2 but before Step4, and after Step4).

First, if a failure happens before Step2, inconsistency does not exist, because Step1 only constructs a new entry without modifying other entries and metadata. Inside this boundary, the previous entry's `next_entry_dis` and next entry's `prev_entry_dis` values still point towards each other. Second, if happening after Step2 but before Step4, the dentry list is inconsistent, which means the next entry's `prev_entry_dis` value points to the new entry but the previous entry's `next_entry_dis` value points to the next entry. Inside this boundary, the new dentry is still invisible to file system, whereas Step3 is possible to begin (or not) enabling a new `CFFS_inode`, incurring the inconsistency between inode and dentry. Last, if Step4 succeeds, the new entry is inserted, restoring all consistency and committing one *create* or *mkdir* operation.

According to above consistency analysis, we design the recovery procedure for type a. It traverses each dentry list to identify the partially-inserted entry (inside the second boundary). Such partial insertion entails the inconsistency of dentry list and inconsistency between inode and dentry. In order to dismiss inconsistency, the recovery procedure retrieves the inode number from the `ino` field of the partially-inserted entry to disable the inconsistent inode, and relies on the

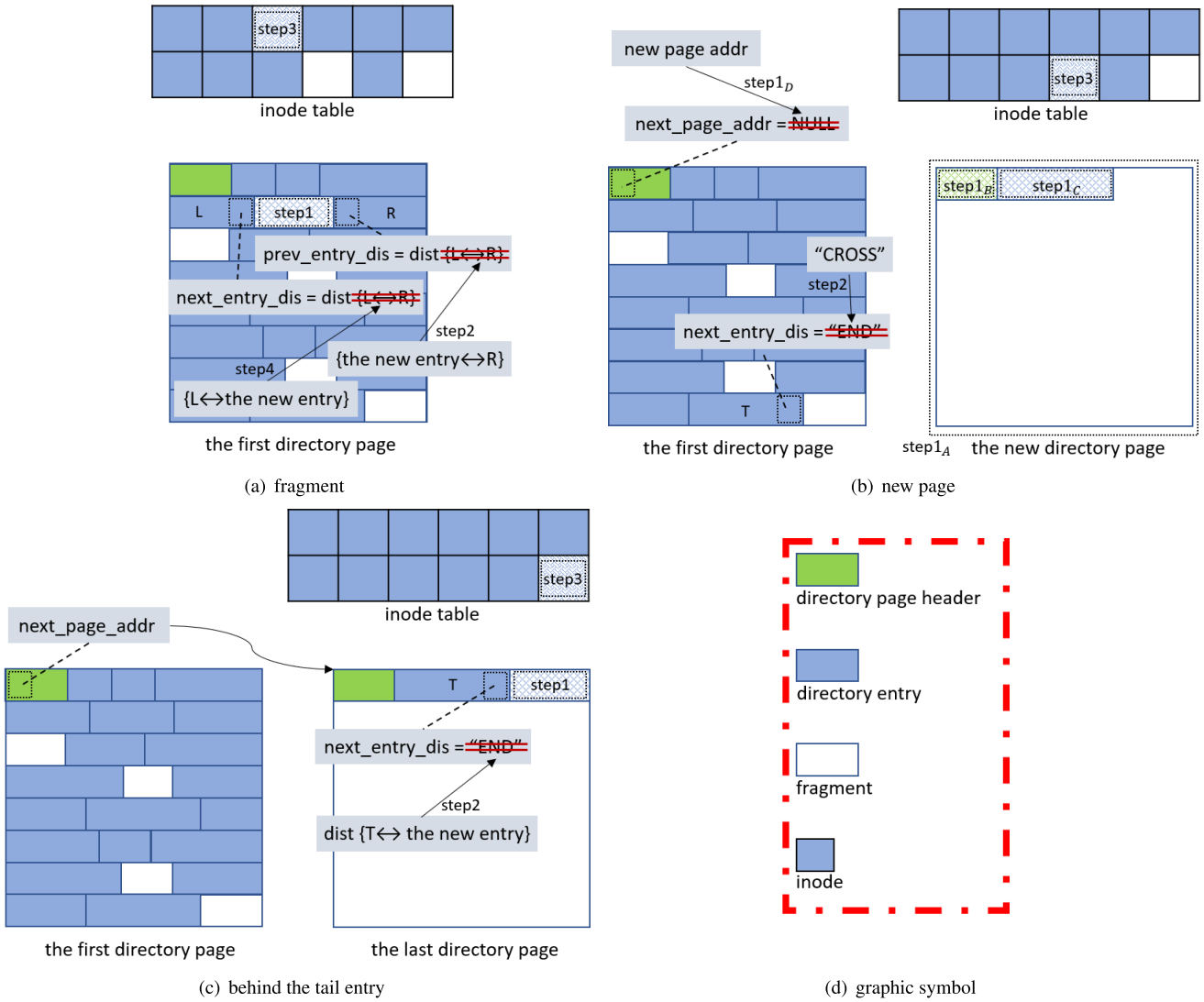


FIGURE 10. **Three situations for *Create*. There are three types of positions for new entries, and each type is associated with different procedures for *create*. The sequence among the steps is ensured via the memory barrier (x86's *sfence*).**
(a) fragment
Step1: select a fragment and construct a new entry in this fragment.
Step2: update entry R's `prev_entry_dis` **field from dist{entry L ↔ entry R} to dist{the new entry ↔ entry R}.**
Step3: enable `CFFS_inode` **with correct metadata.**
Step4: update entry L's `next_entry_dis` **field from dist{entry L ↔ entry R} to dist{entry L ↔ the new entry}.**
(b) new page (it is not necessary to force the execution order among Step1$_A$, Step1$_B$, Step1$_C$, and Step1$_D$)
Step1$_A$: allocate a new page.
Step1$_B$: initiate the new page header.
Step1$_C$: append a new entry immediately after the new header.
Step1$_D$: record the new page address in the header of the previous page.
Step2: update entry T's `next_entry_dis` **field, from** `END` **to** `CROSS PAGE`**.**
Step3: enable `CFFS_inode` **with correct metadata.**
(c) behind the tail entry
Step1: append a new entry after entry T.
Step2: update entry T's `next_entry_dis` **field, from** `END` **to dist{entry T ↔ the new entry}.**
Step3: enable `CFFS_inode` **with correct metadata.**

`prev/next_entry_dis` field of the partially-inserted entry recording the location of its previous and next entry to restore the dentry list to the state before incomplete insertion.

For type b, a new directory page is allocated, establishing a new dentry list. This page is the last page of the directory, and the former last page where the former tail dentry is located records its address. Besides, the new dentry inserted into the new dentry list acts as the tail entry of the directory, so the state of the former tail dentry should be updated. Step1 initializes a new entry whose `next_entry_dis` value is `END` in a new directory page. The `END` value marks this new entry as the tail entry. In Step 2, the former tail

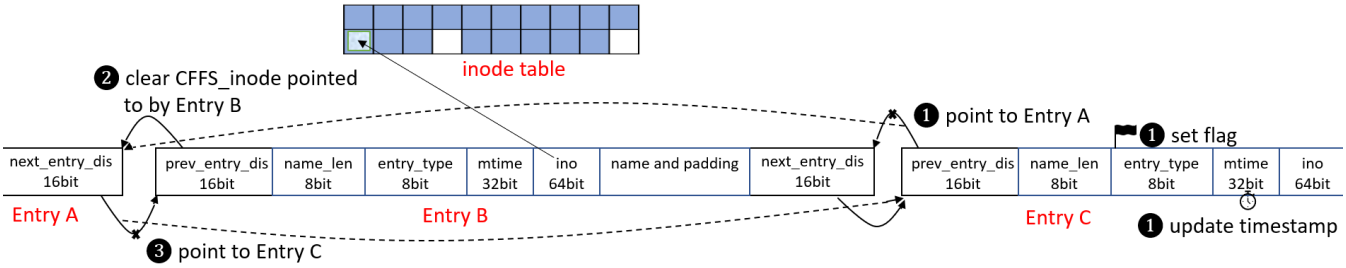


FIGURE 11. **Unlink. Removal of entry A from a dentry list and the clearing of its mapped inode. Even after a crash, the operation can be undone if all steps of 1 have been completed.**

entry's `next_entry_dis` value is updated from `END` to `CROSS PAGE`, which means this entry is the last entry of its dentry list but not the tail entry of the directory. If a traversal procedure for a directory reads the `CROSS PAGE` value from an entry's `next_entry_dis` fields, it switches to the next dentry list for continuing traversal. Thus, Step2 guarantees the consistency between dentry lists. However, until Step3 finishes, the dentry and inode are inconsistent.

For type c, a new entry is the tail dentry appended immediately behind the former tail entry. That is, the new entry is inserted after the former tail entry inside the same dentry list. Its procedure is similar to but simpler than type b. The same inconsistency problem as that of type b occurs. Fortunately, the recovery procedure can tackle both types with the same approach. When the recovery procedure finds a dentry whose `next_entry_dis` value is `END`, check whether the `CFFS_inode` corresponding to this dentry is intact and enabled. If not enabled, undo this failed operation.

The preceding explanation reveals that the `next/prev_entry_dis` fields of some existing entries are updated for inserting a new entry into a dentry list or are marked with `CROSS PAGE`. For types a and c, if the new entry and the associated updates are close, such locality is beneficial to both the cacheline size and the PM access granularity. This is why the `prev/next_entry_dis` fields are included on the leftmost and rightmost sides.

The entire procedure is unnecessary for synchronizing the `CFFS_inode` of the parent directory in PM with the latest metadata because directory size can be calculated from the number of directory pages and because modified time can be extracted from timestamps recorded in entries. Thus, `vfs_inode` (defined by Linux virtual file system) [32] is only updated in DRAM and `CFFS_inode` is not updated in PM, which has higher overhead. After the procedure is completed, the newly created file or directory information can be added to a dentry cached tree for quick lookup.

As stated above, the recovery procedure for a directory needs to traverse all entries in the directory. Fortunately, upon restarting CFFS from system-crashes, rebuilding the dentry cached tree is executed for efficient lookup, and its procedure also traverses directory entries, dropping by to do the inconsistency recovery.

2) UNLINK/RMDIR

These operations are inverse operations for *create* and *mkdir*, used to remove a dentry from a dentry list in a directory page. However, the operations are distinct from *create/mkdir*. First, *unlink* might only remove a hard link without clearing the `CFFS_inode` mapped by the link. Second, unlike *create* or *mkdir*, which record a timestamp in a new entry, *unlink* and *rmdir* generate no new entry; the timestamp should be recorded elsewhere. An example illustrated in Fig. 11 reveals how these problems can be solved.

Entry B, between two entries in the same page, is the removal target. First, the dentry list is made inconsistent by first changing the `pev_entry_dis` field of Entry C; the field initially points to Entry B and points to Entry A after Step 1. This inconsistency exists until Step 3 is committed. Because the `prev_entry_dis`, `entry_type`, and `mtime` fields have an 8-byte alignment, they can be updated simultaneously with a single 64-bit atomic memory *store*. Thus, in a single atomic instruction, Step 1 causes Entry C's `prev_entry_dis` field to point to Entry A, the `mtime` field is updated to the current time, and a flag for failure recovery is inserted in the `entry_type` field to indicate whether the inode pointed to by Entry B should be cleared (because *unlink* does not necessarily delete an inode). If Step 1 is completed, even if the inode has not been cleared by Step 2 or the `next_entry_dis` field of Entry A has not been pointed to Entry C by Step 3, inconsistencies in the dentry list can be detected and *unlink* or *rmdir* can be redone using the information in Entry C.

3) RENAME

rename may involve two parent directories and two inodes; thus, some information must be logged to guarantee an atomic update. Each directory page header has some space reserved for logging to avoid competing for access to system-level log space and unnecessary locking overhead.

VI. EVALUATION

A. EXPERIMENTAL ENVIRONMENT

We evaluated CFFS on a system with Linux (kernel 5.4) and that was powered by a 4-core Intel Core i7-7700 CPU at 3.60 GHz. PM was emulated using 32 GB of DRAM [37].

TABLE 2. File Creation Throughput.

	Throughput (files/second)	Gain (compared with CFFS)
CFFS	436156	1
NOVA	361668	0.829
EXT4-DAX	128188	0.294

CFFS supports multi-grained page management [42]. A huge page's size is 2 MB, and huge pages can be allocated using the SP file, which manages its space with 4-KB units. If a file requires a fragment smaller than 2-MB, the pages are allocated from the SP file. Thus, pages for small allocation are managed with a small group of 512 4-KB pages (i.e., 2 MB); the pages for huge allocation is managed with a group whose size is 512 2-MB pages (i.e., 1 GB). The algorithm for small allocation is similar to the algorithm for huge allocation but based on different management units of the buddy structure (2 MB and 1 GB, respectively).

CFFS could dynamically allocate or reclaim huge pages for or from the SP file based on the SP file's space consumption. Administrators could also statically assign huge pages for the SP file during mounting. In our experiment, the SP file was located in a 1-GB space statically split from the PM for small fragment allocation; some leftover space that could not be designated as a 1-GB-aligned group was also subordinated to the SP file (see Fig. 5), preferentially for metadata or directory page allocation.

B. EVALUATION OVERVIEW

For CFFS and other PM file systems, we quantify the capability of directory operations, the performance of page allocation, and the influence of contiguity on page faults resulting from *mmap*.

In terms of directory operations, we measure the file creation throughput. Besides, we design workloads mixing file creation and deletion, and alter the magnitude of the workload size to observe whether the performance degradation exists when the workload size is large.

The performance of page allocation would vary with the degree of free space fragmentation. Therefore, before measuring the latency of *fallocate*s, we tune the fragmentation level in the PM volume. We design workloads for small size allocation (4-KB) and large size allocation (2-MB) respectively, and observe the variation of *fallocate* latency with the severer fragmentation.

We want to verify that CFFS could maintain great file contiguity in disadvantaged situations, and the contiguity would reflect in the performance of page faults caused by *mmap*. Therefore, our workloads would not generate spatial-occupied files by allocating sufficient pages at once for each file. Instead, each allocation only asks for few pages to present the behavior of file growth, and if a file undergoes consecutive allocation, another file would preempt the right to next allocation, simulating the condition that multiple applications interleavedly request allocation from one PM storage for their own files.

C. DIRECTORY OPERATIONS

The results regarding the throughput for *create* are presented in Table 2. CFFS created 436,156 files/second; this figure for NOVA was 82.9% smaller at 361,668 files/second. CFFS's advantage was due to its spatial locality enabling the full use of PM's access granularity and cacheline size. For each new added dentry, NOVA must persist the offset of the tail entry as metadata in an inode. This inode is not physically near the new dentry located in a directory page; thus, different cachelines are separately occupied, and more cacheline flushing is necessary. In addition, NOVA uses journaling to guarantee the consistency of directory operations between a new inode and its parent inode; this method not only writes more data to PM but also aggravates the aforementioned bottleneck.

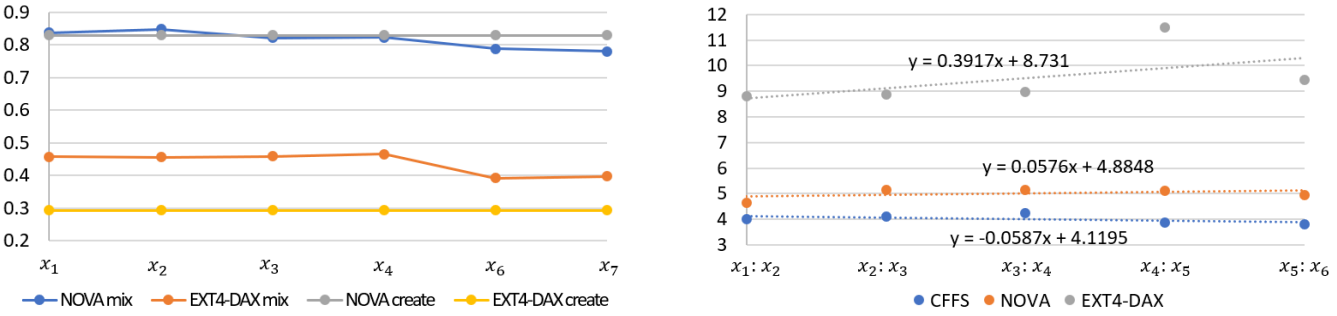
By contrast, CFFS's dentry insertion is an operation that adds a new entry and modifies the previous entry's `next_entry_dis`' field or the next entry's `prev_entry_dis` field, which is typically physically close to the new entry. Due to CFFS's dentry format, the leftmost `prev_entry_dis` and rightmost `next_entry_dis` fields, filling a new entry and modifying the corresponding fields of its neighbors can be included in the same cacheline. That is, these operations can be flushed together to leverage the access granularity of PM (the granularity of Optane DC Persistent Memory is 256 bytes) [48].

We then measured the latency of workloads with mixed *create* and *unlink* tasks. First, x files were created and, x files were then deleted. The order of deletion was random, so NOVA's garbage collection was easily triggered. We used Table 2's throughput information to inversely estimate the latency and compare it with measurements (Table 3). The actual latencies of these file systems were lower than their estimates, indicating that *unlink* is a lighter operation than *create* because *unlink* requires lower I/O amount.

CFFS's latency is lower than that of other file systems for all x values (Table 3). When x is multiplied by two, CFFS's latency doubles. The latency gradient between x_i and x_{i+1} was calculated as $(latency_{x_{i+1}} - latency_{x_i})/(x_{i+1} - x_i)$; the results are presented in Fig. 12(b). For CFFS and NOVA, the slope of the gradient was small, indicating that the latency increased approximately linearly with x; by contrast, the gradient for EXT4-DAX's increased gradually with intervals of x. Thus, creation/deletion scalability affects the performance of EXT4-DAX, explaining the decline of the curve for EXT4-DAX in Fig. 12(a). Fig. 12(a) presents the performance results with CFFS as the benchmark; the performance metrics were calculated as $latency_{CFFS}/latency$ or $throughput/throughput_{CFFS}$. The relative performance of EXT4-DAX for file creation was 0.294 (Table 2) for 128,188 files/second. We compared the EXT4-DAX-mix and EXT4-DAX-create curves and observed that EXT4-DAX's performance was close to that of CFFS for mixed workloads (the gain is about 0.45) but that EXT4-DAX decreased at x_6 and x_7 (the gain is about 0.4).

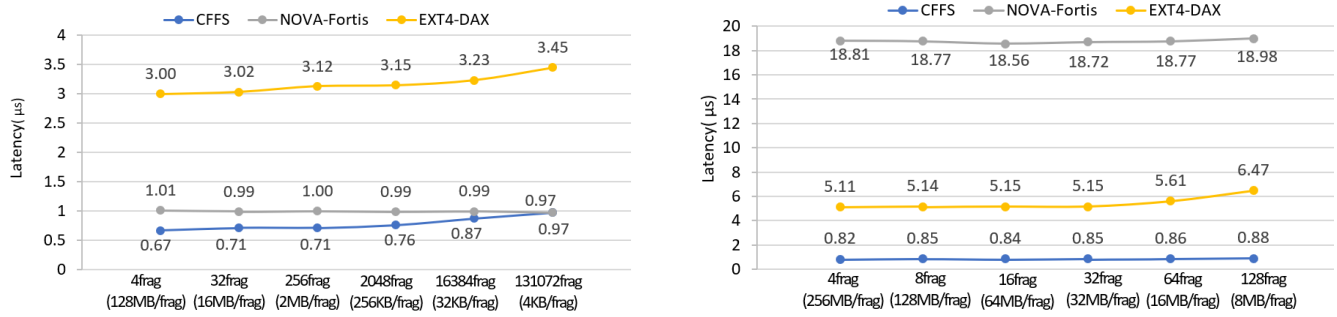
TABLE 3. Latency of mixed directory workloads (create and unlink).

Latency (create x files, then unlink x files)						
	$x_1 = 512$	$x_2 = 1024$	$x_3 = 2048$	$x_4 = 4096$	$x_5 = 8192$	$x_6 = 16384$
CFFS	2111μs	4155μs	8346μs	17055μs	32846μs	64117μs
CFFS(estimated)	2348μs	4696μs	9392μs	18784μs	37568μs	75136μs
NOVA	2521μs	4899μs	10160μs	20717μs	41705μs	82170μs
NOVA(estimated)	2831μs	5662μs	11324μs	22648μs	45296μs	90592μs
EXT4-DAX	4607μs	9108μs	18190μs	36594μs	83628μs	160981μs
EXT4-DAX(estimated)	7988μs	15976μs	31952μs	63904μs	127808μs	255616μs



(a) **Mix and create gain compared with CFFS.** (b) **Latency gradient for various file number intervals.**

FIGURE 12. Performance analysis for mixed directory workloads.



(a) a file is allocated with 512 MB in total (each *fallocate* returns a 4-KB page) (b) a file is allocated with 1 GB in total (each *fallocate* returns a 2-MB page)

FIGURE 13. **Average latency of one fallocate. Before measurement, we determined the free space fragmentation level of PM. The sum of all free chunks (free space fragments) is 512 MB for (a) and 1 GB for (b). "*x* frag (*y* MB/frag)" indicates the fragmentation level that *x* free space fragments of *y* MB each exist in the storage volume. Our workloads allocated file fragments from free space fragments, and we compared the performance under different free space fragmentation levels.**

In our workloads, NOVA directory operations did not have the scalability problem in Fig. 12(b). However, according to the NOVA-mix curve in Fig. 12(a) for x_6 and x_7, the gain of NOVA-mix is lower than the gain of NOVA-create (0.829), identified in Table 2. That is, the performance gap was greater for workloads involving *unlink*; this difference was mainly caused by NOVA's garbage collection, which rearranges directory pages containing excessive invalidated entries generated by *unlink*. Furthermore, NOVA invalidates entries by appending a new entry representing deletion; this method increases space consumption, resulting in more frequent allocation for directory pages. By contrast, CFFS reuses fragments in directory pages that reduce the frequency of immediate directory compaction.

D. PAGE ALLOCATION

We substituted NOVA with NOVA-Fortis [38] because NOVA does not support *fallocate*. Code that clears the content of the allocated pages was commented out because we aimed to analyze pure software overhead.

We measured sequential allocations for various free space fragmentation levels. Each *fallocate* allocates a file fragment logically contiguous to the previous file fragment. Fig. 13(a) presents workloads that allocate 512 MB of data from 512 MB of total free chunk by calling *fallocate* 131,072 (131,072 = 512 MB/4 KB) times. Fig. 13(b) presents workloads that allocate 1 GB of data from 1 GB of total free chunk by calling *fallocate* 512 (512 = 1 GB/2 MB) times. Both in (a) and (b), we set up various free space fragmentation levels to generate different numbers of free chunks.

1) CFFS LATENCY

CFFS had the lowest latency of all file systems for all fragmentation levels; this high performance was due to its global view of free space, which facilitated the quick acquisition of suitable pages. However, its latency increased with fragmentation and gradually converged to the latency of NOVA-Fortis in Fig. 13(a). In these workloads, the number of file fragments generated by CFFS was approximately equal to the number of free chunks; thus, workloads at high fragmentation levels result in overly numerous file fragments. CFFS caches file fragment information in cached extent trees, and it adopts red-black trees to avoid the excessive index sorting of B+ trees' internal nodes due to random allocation; however, the height of red-black trees increases with the number of file fragments.

2) NOVA-FORTIS LATENCY

NOVA-Fortis's average *fallocate* latency in Fig. 13(b) is over ten times greater than its latency in Fig. 13(a). This result is primarily due to its incomplete implementation of multi-grained page management. NOVA-Fortis achieves fast translation from logical to physical pages by using each file's radix tree in DRAM. Each leaf node represents a logical 4-KB page, and a fragment consisting of multiple 4-KB pages requires representation by the same number of leaf nodes. Therefore, multiple leaf nodes could be mapped to one file fragment; this differs from CFFS and EXT4-DAX's cached extent tree in which each node represents an entire file fragment.

For the workloads in Fig. 13(b), each 2-MB allocation comprises 512 contiguous small pages; thus, NOVA-Fortis fills up 512 leaf nodes of a file radix tree. The software overhead was even higher than that of EXT4-DAX, which uses an extent to represent contiguous pages. Adopting methods [39], [46] for leaf node compaction might reduce the overhead of its file radix trees.

3) EXT4-DAX LATENCY

We also observed the growth of EXT4-DAX latency as fragmentation increased (Fig. 13). The average EXT4-DAX latency in Fig. 13(b) was slightly higher than that of Fig. 13(a). This phenomenon is similar to the aforementioned phenomenon for NOVA-Fortis. We did not adjust the EXT4-DAX block size to 2 MB (we only used ''-stride'' to enable 2 MB alignment); thus, a 2-MB allocation modifies 512 bits in block allocation bitmaps.

E. MMAP

Applications can directly access files through *mmap*, and file systems must handle page faults generated by the first reference to mapped pages. Some systems [43] prefaults with *mmap* to construct page table entries for all mapped pages in advance, preventing the page fault handler with high overhead from interrupting application execution. We observed that the latency of page faults was affected by the physical contiguity of files. Thus, we measured the latency of page faults under different workloads to determine which factors affect performance.

1) WORKLOAD

We designed workloads to simulate the behavior of multiple applications requesting for new pages to gradually grow files and measured the latency of *mmap* on those files. Typically, allocation is requested for each file multiple times, and the size of files could grow in the long term. Allocation requests for other files may occur during a sequence of allocation requests for a specific file. Our workloads simulated this behavior by allocating for other files if the consecutive number of allocation requests for a specific file exceeded the set number, reflecting the temporal locality of allocation patterns.

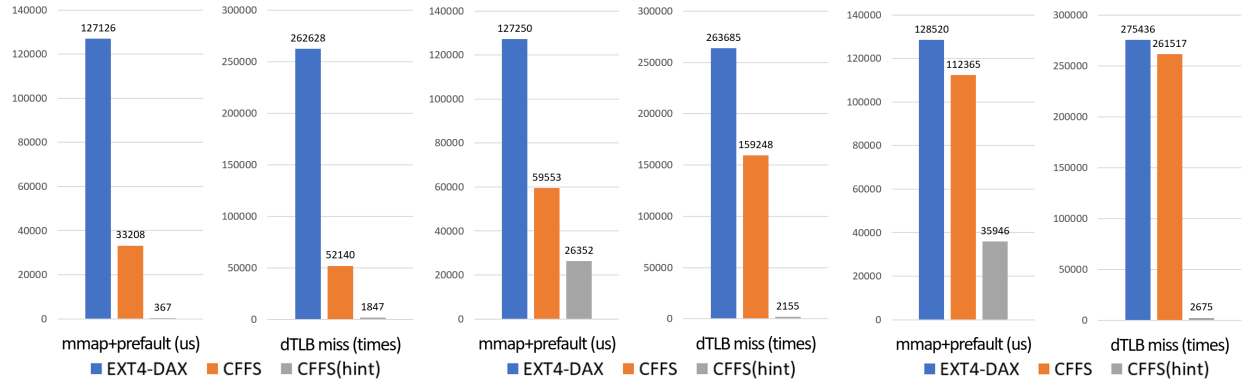
After the workloads in in Fig. 14 were run, files with different sizes were generated. These were broadly divided into two types: huge files and small files. Small files were files smaller than 2 MB; huge files were those greater than or equal to 2 MB. The total files generated by the workloads occupied 1 GB, and different workloads had different ratios of small files to huge files in terms of total size. Then, the number of huge and small files were separately determined, and their total sizes were randomly distributed to each file.

Each file required a 4-KB page on each request, and allocation was rotated to another file after two consecutive requests for the original target. Under CFFS, small file allocations first use the static 1-GB space for the SP file set up during mounting. No such setting was used for EXT4-DAX.

2) MEASUREMENT

Fig. 14(a) reveals that CFFS's latency for prefaulted *mmap* and number of dTLB misses for accessing the mapped PM pages was approximately one-quarter of that of EXT4-DAX. Because CFFS's page allocation assumes that a fragment is likely to grow successively and attempts to maintain spatial locality, huge files could maintain better physical continuity, as reflected in the performance of *mmap*. The performance can be dramatically enhanced if we include application hints when calling *fallocate* for the small files in our workload. By using these hints, applications can inform CFFS that the allocated fragments will remain small. In this case, CFFS uses the fixed-sized greedy allocator instead of the growing-size greedy allocator.

We presented the results for a greater number of small files in Fig. 14(b). CFFS still has approximately half of the *mmap* latency and dTLB misses as compared to EXT4-DAX, but CFFS's advantage of performance decreased with respect to the case in Fig. 14(a). In Fig. 14(c), the capacity and number of small files increased further. If CFFS's allocation for small files was restricted to using the 1-GB capacity in the SP file we set during mounting, its performance was only better than EXT4-DAX's performance slightly. However, if hints for small files were applied, CFFS always had a noticeable performance advantage.



(a) 1020 MB for 64 huge files
2 MB for 128 small files

(b) 1020 MB for 64 huge files
4 MB for 256 small files

(c) 1008 MB for 64 huge files
16 MB for 1024 small files

FIGURE 14. **mmap(prefault) performance.** ***mmap*s are done for all huge files, not for any small files. We measure the latency of all *mmap*s with prefaults and the sum of data-TLB misses triggered by accessing the mapped PM region.**

TABLE 4. **Results for single-size workloads. Each column presents the result of a workload generating a given number of identically sized files with a total volume of 1 GB. Each file space allocation was a 4-KB page. After four consecutive allocations for a file, a different file was chosen for allocation with the round-robin method. After performing one of the workloads, we measure the average number of file fragments in each file, the *mmap* latency for all allocated pages, and data-TLB misses for accessing all mapped pages.**

		4x 256-MB files	16x 64-MB files	64x 16-MB files	512x 2-MB files	1024x 1-MB files
EXT4-DAX	Fragments per file	16434	4108	1027	128	64
	mmap prefault latency	31906485μs	8073755μs	2009496μs	251982μs	148679μs
	data-TLB miss	367836	268425	267733	269508	267233
CFFS	Fragments per file	128	32	8	1	7
	mmap prefault latency	80654μs	24326μs	10008μs	2691μs	133154μs
	Data-TLB miss	2046	2046	2333	2394	272692

We observed the degradation of CFFS's performance as the small file ratio increased, especially in trials without hints for small files. This result is unsurprising because the growing-size greedy allocator is primarily beneficial in situations with files that tend to increase in size. For numerous small files, application hints substantially improve the performance.

3) ANALYSIS

We used various experiments to analyze the preceding results and demonstrate how the height of the extent trees and the construction of page table entries affects page faults.

First, we ran the workloads described in Table 4. Each column of the table denotes the workload generating a certain number of single-size files totally occupying 1 GB. From (4x 256-MB files) to (1024x 1-MB files) column, the *mmap*-prefault latency gradually decreased for EXT4-DAX. We attributed this phenomenon to the decreased number of fragments per file. Compared to EXT4-DAX, CFFS had remarkably small *mmap*-prefault latency from (4x 256-MB files) to (512x 2-MB files) column. However, the overhead exploded in (1024x 1-MB files) column whose latency was similar to that of EXT4-DAX. This performance degradation was induced by the lack of 2-MB alignment. We explained how these two factors, the number of file fragments and fragment alignment, were related to the results in Table 4 below.

For EXT4-DAX, Table 4 reveals that the number of fragments per file and the latency of *mmap* with prefaults decreased with the number of files, but the total number of fragments and dTLB misses did not change substantially. Although the total number of fragments was almost constant under different workloads, they were distributed across a different number of files. Files with fewer fragments have smaller extent trees. We thus infer that the decrease in prefaulted *mmap* latency was primarily due to the reduction of the size of the extent trees. To handle a page fault on a file page, EXT4-DAX searches the file's extent tree for that page. If the tree size is small, the path sum for searching is also small. The DAX page fault handler in Linux uses a radix tree to record each file's mapped offset, amplifying the effects of reduced tree size. We excluded the construction of page table entries because little difference was observed in dTLB misses across trials, indicating that the number of page table entries and construction overhead were similar in all trials.

For CFFS, Table 4 reveals that prefaulted *mmap* also had decreased latency due to the reduction in the height of each extent tree. The last column presents the result of a workload of generating 1024 files with a size of 1 MB; the *mmap* latency was substantially higher than those in other columns, even if there are no such significant differences

TABLE 5. Results for workloads generating huge files accompanied by numerous small dummy files. Each column presents the results of a workload generating a given number of identically sized huge files with an total volume of 512 MB. Along with huge files, each workload generated small dummy files with total volume of 512 MB. Each allocation for a file was a 4-KB page. After two consecutive allocations for a huge file, two new 4-KB small dummy files were created, and then a different huge file was chosen for allocation with the round-robin method. The measurement items are similar to the items in Table 4, but we only measure them for huge files, excluding the measurement for small files.

		1x 512-MB files	4x 128-MB files	16x 32-MB files	64x 8-MB files	256x 2-MB files
EXT4-DAX	Fragments per file	54126	16433	4109	1028	257
	mmap prefault latency	61841541μs	16974189μs	4495431μs	1056541μs	304557μs
	dTLB miss	129935	135412	135216	134861	136391
CFFS	Fragments per file	69334	17333	4323	1034	1
	mmap prefault latency	63693059μs	15994895μs	4583744μs	760335μs	2468μs
	dTLB miss	134321	131053	130442	99471	835
CFFS(hint)	Fragments per file	256	64	16	4	1
	mmap prefault latency	226172μs	65792μs	49745μs	5124μs	2584μs
	dTLB miss	663	698	682	1007	849

between the number of file segments in each column. Other columns present the results of workloads generating files whose size were multiples of 2 MB; running these workloads with CFFS's allocation algorithm could generate intact 2-MB fragments, and the construction of page table entries could adopt the page middle directory (PMD) with huge page entries. However, PMD could not be used for 1-MB files, and page fault thus constructed more entries for normal pages, resulting in higher overhead. The sharp increase of data-TLB misses in the last column indirectly verifies this conclusion. Thus, we attribute that skyrocketing latency in the last column for CFFS to the lack of highly aligned file fragments.

Second, we ran the workloads described in Table 5. Each column of the table denotes the workload generating 128,000 small dummy files with 4-KB size and a certain number of single-size huge files totally occupying 512 MB. In fact, if a file extends its size to 2 MB, CFFS no longer treats it as a small file, and its subsequent allocations are performed using the 2-MB-unit buddy system instead of the SP file. For meaningful analysis, note that we disabled this policy so that huge files always use the 4-KB-unit buddy system and all allocation gets free pages only from the 1-GB space statically split for small fragment allocation.

In Table 5, from (1x 512-MB files) to (64x 8-MB files) column, EXT4-DAX and CFFS had similar *mmap*-prefault latency. Excluding the above-mentioned factor of disabling the policy for huge file allocation from our discussion, the reason why CFFS performance in those workloads wasn't superior was that we treated every small 4-KB dummy file as if a new fragment would be further appended to extend its size. Thus, the growing-size greedy allocator was adopted for the small fixed-size files so that implicit preallocation applied to them hindered the potential contiguity for the huge files in those workloads. Those huge files could only own 2-MB alignment in the first or second 2 MB inside each file, incurring the severe latency for page faults. However, in (256x 2-MB files) column, the overhead for CFFS was prominently improved, and its *mmap*-prefault latency was substantially lower than that of EXT4-DAX, because the size of each huge file in this case was exactly 2 MB, satisfying 2-MB alignment for the whole file.

The results of CFFS(hint) in Table 5 showed outstanding performance, whose *mmap*-prefault latency was less than one tenth of EXT4-DAX's latency from (1x 512-MB files) to (256x 2-MB files) column. Such an astonishing advantage was achieved by the realization of 2-MB alignment for all file fragments in the huge files. Recall that applications could give CFFS's *fallocate* a hint to indicate whether the current allocation would be fixed-size or growing-size. For the workloads of CFFS(hint), the pages of 4-KB small dummy files were allocated with a fixed-size hint, and the pages of huge files were allocated with a growing-size hint. Therefore, the allocation for those small dummy files would avoid destroying the contiguity of intact free chunks, and the allocation for those growing-size huge files would implicitly preallocate sufficient pages for the extension of file fragments. As a result, the huge files in CFFS(hint) could maintain fine contiguity, resulting in a high degree of fragment alignment and few fragments per file, and the fine contiguity reflected in the great performance of page faults.

VII. CONCLUSION

Physically contiguous allocation was used to reduce software overhead for *mmap*-based operations on PM file systems. File with high contiguity have few fragments, and each fragment is highly aligned. Reducing the number of fragments alleviates the cost of managing PM pages for files. Fragment alignment affects the page size adopted for constructing page table entries for the pages inside the fragment. We tested the scheme on numerous workloads to reveal how these two factors affect the performance of page faults on the performance of *mmap* for a file in PM.

CFFS's allocation strategy optimizes the two aforementioned factors by considering both spatial and temporal locality of file access patterns. An application frequently requests contiguous pages in a short period for growing files. We thus designed greedy-based buddy systems and implemented implicit preallocation with an NMRU policy, increasing file contiguity by avoiding blockage due to allocation of space for other files. Besides, the allocation provides sufficient pages for applications with a high speed algorithm.

We also propose a fine-grained scheme for managing and persisting various metadata. By fully using the cacheline size and access granularity of PM, the fine-grained metadata updates resulted in improved performance. CFFS's directory operations had better performance for PM than other common logging mechanisms.

Our next project plans to design a user-space library for file operations, cooperating with kernel-space CFFS. The library will leverage *fallocate* and *mmap* provided by CFFS to implement APIs. The library can obtain file pages by calling *fallocate* with an appending or non-appending hint determined by the calling history and the current size of a file. Then, the pages can be mapped to user-space address by calling *mmap* so that applications can access PM without system call overhead. Thus, atomic write operations for files can be designed in user space. We believe that such memory-style file operation can facilitate flexibility and encompasses performant APIs, compared to traditional system calls of kernel-space file systems.

REFERENCES

[1] J. Handy. (2015). *Understanding the Intel/Micron 3D XPoint Memory*. [Online]. Available: https://www.snia.org/sites/default/files/SDC15_presentations/persistant_mem/JimHandy_ Understanding_the-Intel.pdf

[2] *Intel OptaneT Persistent Memory (PMem) 200 Series*, Intel, Santa Clara, CA, USA, 2020.

[3] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, J. Zhao, and S. Swanson, "Basic performance measurements of the Intel optane DC persistent memory module," 2019, *arXiv:1903.05714*.

[4] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better I/O through byte-addressable, persistent memory," in *Proc. ACM SIGOPS 22nd Symp. Operating Syst. Princ.*, 2009, pp. 133–146.

[5] X. Wu, S. Qiu, and A. L. N. Reddy, "SCMFS: A file system for storage class memory and its extensions," *ACM Trans. Storage*, vol. 9, no. 3, pp. 1–23, Aug. 2013.

[6] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System software for persistent memory," in *Proc. 9th Eur. Conf. Comput. Syst.*, 2014, pp. 1–15.

[7] J. Xu and S. Swanson, "NOVA: A log-structured file system for hybrid volatile/non-volatile main memories," in *Proc. 14th USENIX Conf. File Storage Technol.*, Feb. 2016, pp. 323–338.

[8] (2019). *Direct Access for Files*. [Online]. Available: https://www.kernel.org/doc/Documentation/filesystems/dax.txt

[9] J. Seo, W.-H. Kim, W. Baek, B. Nam, and S. H. Noh, "Failure-atomic slotted paging for persistent memory," in *Proc. 22nd Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Apr. 2017, pp. 91–104.

[10] Y. Chen, Y. Lu, F. Yang, Q. Wang, Y. Wang, and J. Shu, "FlatStore: An efficient log-structured key-value storage engine for persistent memory," in *Proc. 25th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Mar. 2020, pp. 1077–1091.

[11] J. Xu, J. Kim, A. Memaripour, and S. Swanson, "Finding and fixing performance pathologies in persistent memory software stacks," in *Proc. 24th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Apr. 2019, pp. 427–439.

[12] K. Shen, S. Park, and M. Zhu, "Journaling of journal is (almost) free," in *Proc. 12th USENIX Conf. File Storage Technol.*, Feb. 2014, pp. 287–293.

[13] R. Kadekodi, S. K. Lee, S. Kashyap, T. Kim, A. Kolli, and V. Chidambaram, "SplitFS: Reducing software overhead in file systems for persistent memory," in *Proc. 27th ACM Symp. Operating Syst. Princ.*, Oct. 2019, pp. 494–508.

[14] J. Choi, J. Hong, Y. Kwon, and H. Han, "Libnvmmio: Reconstructing software IO path with failure-atomic memory-mapped interface," in *Proc. USENIX Annu. Tech. Conf.*, 2020, pp. 1–16.

[15] S. Patil and G. Gibson, "Scale and concurrency of GIGA+: File system directories with millions of files," in *Proc. 9th USENIX Conf. File Storage Technol.*, Feb. 2011, pp. 177–190.

[16] T. Sato, "EXT4 online defragmentation," in *Proc. Ottawa Linux Symp.*, 2007, pp. 179–186.

[17] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee, "CoLT: Coalesced large-reach TLBs," in *Proc. 45th Annu. IEEE/ACM Int. Symp. Microarchitecture*, Dec. 2012, pp. 258–269.

[18] G. Cox and A. Bhattacharjee, "Efficient address translation for architectures with multiple page sizes," in *Proc. 22nd Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Apr. 2017, pp. 435–448.

[19] S. Son, J. Yoo, and Y. Won, "Guaranteeing the metadata update atomicity in EXT4 file system," in *Proc. 8th Asia–Pacific Workshop Syst.*, Sep. 2017, pp. 1–8.

[20] (2021). *LMDB*. [Online]. Available: https://github.com/LMDB/lmdb

[21] D. Roselli, J. R. Lorch, and T. E. Anderson, "A comparison of file system workloads," in *Proc. USENIX Annu. Tech. Conf.*, 2000, pp. 41–54.

[22] A. Moulton and S. E. Madnick, "A temporal and spatial locality theory for characterizing very large data bases," in *Proc. 22nd Int. Conf. Intell. Transp. Syst. (HICSS)*, vol. 2, Jan. 1989, pp. 612–620.

[23] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, "Crash consistency: FSCK and journaling," in *Operating Systems: Three Easy Pieces*. Scotts Valley, CA, USA: CreateSpace Independent, 2018, ch. 42. [Online]. Available: https://pages.cs.wisc.edu/~remzi/OSTEP/

[24] Y. Son, S. Kim, H. Y. Yeom, and H. Han, "High-performance transaction processing in journaling file systems," in *Proc. 16th USENIX Conf. File Storage Technol.*, Feb. 2018, pp. 227–240.

[25] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Analysis and evolution of journaling file systems," in *Proc. USENIX Annu. Tech. Conf.*, 2005, pp. 105–120.

[26] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," *ACM Trans. Comput. Syst.*, vol. 10, no. 1, pp. 26–52, Feb. 1992.

[27] *Intel 64 and IA-32 Architectures Software Developer's Manual*, Intel, Santa Clara, CA, USA, 2021.

[28] T. C.-H. Hsu, H. Brügner, I. Roy, K. Keeton, and P. Eugster, "NVthreads: Practical persistence for multi-threaded applications," in *Proc. 12th Eur. Conf. Comput. Syst.*, Apr. 2017, pp. 468–482.

[29] T. David, A. Dragojević, R. Guerraoui, and I. Zablotchi, "Log-free concurrent data structures," in *Proc. USENIX Annu. Tech. Conf.*, 2018, pp. 373–386.

[30] J. Gu, Q. Yu, X. Wang, Z. Wang, B. Zang, H. Gua, and H. Chen, "Pisces: A scalable and efficient persistent transactional memory," in *Proc. USENIX Annu. Tech. Conf.*, 2019, pp. 913–928.

[31] K. Wu, J. Ren, I. Peng, and D. Li, "ArchTM: Architecture-aware, high performance transaction for persistent memory," in *Proc. 19th USENIX Conf. File Storage Technol.*, Feb. 2021, pp. 141–153.

[32] *Overview of the Linux Virtual File System*. Accessed: Oct. 5, 2021. [Online]. Available: https://www.kernel.org/doc/html/latest/filesystems/vfs.html

[33] K. Torp, C. S. Jensen, and R. T. Snodgrass, "Effective timestamping in databases," *VLDB J. Int. J. Very Large Data Bases*, vol. 8, nos. 3–4, pp. 267–288, Feb. 2000, doi: 10.1007/s007780050008.

[34] K. Torp, C. S. Jensen, and R. T. Snodgrass, "Stratum approaches to temporal DBMS implementation," in *Proc. Int. Database Eng. Appl. Symp.*, 1998, pp. 4–13, doi: 10.1109/IDEAS.1998.694346.

[35] Wikipedia. *Network Time Protocol*. Accessed: Feb. 24, 2022. [Online]. Available: https://en.wikipedia.org/wiki/Network_Time_Protocol

[36] (2021). *How to Configure NTP in Your Enviornment and Common Issues*. IBM. [Online]. Available: https://www.ibm.com/support/pages/how-configure-ntp-your-enviornment-and-common-issues

[37] (2017). *How to Emulate Persistent Memory Using Dynamic Random-access Memory (DRAM)*. Intel. [Online]. Available: https://software.intel.com/content/www/us/en/develop/articles/how-to-emulate-persistent-memory-on-an-intel-architecture-server.html

[38] J. Xu, L. Zhang, A. Memaripour, A. Gangadharaiah, A. Borase, T. B. Da Silva, S. Swanson, and A. Rudoff, "NOVA-Fortis: A fault-tolerant non-volatile main memory file system," in *Proc. 26th Symp. Operating Syst. Princ.*, Oct. 2017, pp. 478–496.

[39] S. Ma, K. Chen, S. Chen, M. Liu, J. Zhu, H. Kang, and Y. Wu, "ROART: Range-query optimized persistent ART," in *Proc. 19th USENIX Conf. File Storage Technol.*, Feb. 2021, pp. 1–16.

[40] M. Cao, S. Bhattacharya, and T. Ts'o, "Ext4: The next generation of Ext2/3 filesystem," in *Proc. Linux Storage Filesystem Workshop*, 2007.

[41] A. Kumar, M. Cao, J. Santos, and A. Dilger, "Ext4 block and inode allocator improvements," in *Proc. Ottawa Linux Symp.*, 2008, pp. 263–274.
[42] T.-Y. Chen, Y.-H. Chang, M.-C. Yang, Y.-J. Chen, H.-W. Wei, and W.-K. Shih, "Multi-grained block management to enhance the space utilization of file systems on PCM storages," *IEEE Trans. Comput.*, vol. 65, no. 6, pp. 1831–1845, Jun. 2016.
[43] J. Choi, J. Kim, and H. Han, "Efficient memory mapped file I/O for in-memory file systems," in *Proc. 9th USENIX Conf. Hot Topics Storage File Syst.*, 2017.
[44] S. Kashyap, C. Min, K. Kim, and T. Kim, "A scalable ordering primitive for multicore machines," in *Proc. 13th EuroSys Conf.*, Apr. 2018, pp. 1–15.
[45] C. Wang, Q. Wei, L. Wu, S. Wang, C. Chen, X. Xiao, J. Yang, M. Xue, and Y. Yang, "Persisting RB-tree into NVM in a consistency perspective," *ACM Trans. Storage*, vol. 14, no. 1, pp. 1–27, Feb. 2018.
[46] V. Leis, A. Kemper, and T. Neumann, "The adaptive radix tree: ARTful indexing for main-memory databases," in *Proc. IEEE 29th Int. Conf. Data Eng. (ICDE)*, Apr. 2013, pp. 38–49, doi: 10.1109/ICDE.2013.6544812.
[47] C. Chen, J. Yang, Q. Wei, C. Wang, and M. Xue, "Fine-grained metadata journaling on NVM," in *Proc. 32nd Symp. Mass Storage Syst. Technol. (MSST)*, 2016, pp. 1–13, doi: 10.1109/MSST.2016.7897077.
[48] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson, "An empirical guide to the behavior and use of scalable persistent memory," in *Proc. 18th USENIX Conf. File Storage Technol.*, Feb. 2020, pp. 169–182.
[49] C. Wang, Q. Wei, J. Yang, C. Chen, and M. Xue, "How to be consistent with persistent memory? An evaluation approach," in *Proc. IEEE Int. Conf. Netw., Archit. Storage (NAS)*, Aug. 2015, pp. 186–194.
[50] J. Park, D. H. Kang, and Y. I. Eom, "File defragmentation scheme for a log-structured file system," in *Proc. 7th Asia–Pacific Workshop Syst.*, 2016, pp. 1–7.
[51] O. Rodeh, J. Bacik, and C. Mason, "BTRFS: The Linux B-tree filesystem," *ACM Trans. Storage*, vol. 9, no. 3, pp. 1–32, Aug. 2013.
[52] R. Kesavan, M. Curtis-Maury, and M. K. Bhattacharjee, "Efficient search for free blocks in the WAFL file system," in *Proc. 47th Int. Conf. Parallel Process.*, Aug. 2018, pp. 1–10.
[53] R. Kadekodi, S. Kadekodi, S. Ponnapalli, H. Shirwadkar, G. R. Ganger, A. Kolli, and V. Chidambaram, "WineFS: A hugepage-aware file system for persistent memory that ages gracefully," in *Proc. ACM SIGOPS 28th Symp. Operating Syst. Princ. (CD-ROM)*, Oct. 2021, pp. 804–818.
[54] I. Neal, G. Zuo, E. Shiple, T. A. Khan, Y. Kwon, S. Peter, and B. Kasikci, "Rethinking file mapping for persistent memory," in *Proc. 19th USENIX Conf. File Storage Technol.*, Feb. 2021, pp. 97–111.
[55] Intel. *Intel Optane Business Update: What Does This Mean for Warranty and Support*. Accessed: Aug. 2, 2022. [Online]. Available: https://www.intel.ca/content/www/ca/en/support/articles/000091826/memory-and-storage.html

JEN-KUANG LIU received the bachelor's degree from the Department of Electrical Engineering, National Taiwan University, Taipei, Taiwan, in 2019, where he is currently pursuing the master's degree with the Department of Electrical Engineering. His research interests include storage systems and operating systems.

SHENG-DE WANG (Life Member, IEEE) received the B.S. degree from the National Tsing Hua University, Hsinchu, Taiwan, in 1980, and the M.S. and Ph.D. degrees in electrical engineering from the National Taiwan University, Taipei, Taiwan, in 1982 and 1986, respectively. Since 1986, he has been as a Faculty Member of the Department of Electrical Engineering, National Taiwan University, where he is currently a Professor. From 1995 to 2001, he worked as the Director of the Computer and Information Network Center, Computer Operating Group, National Taiwan University. He was a Visiting Scholar with the Department of Electrical Engineering, University of Washington, Seattle, from 1998 to 1999. From 2001 to 2003, he worked as the Department Chair of the Department of Electrical Engineering, National Chi Nan University, Puli, Taiwan. His research interests include embedded systems, internet computing and security, and intelligent systems.