ctFS: Replacing File Indexing with Hardware Memory Translation through Contiguous File Allocation for Persistent Memory

Ruibin Li, Xiang Ren, Xu Zhao, Siwei He, Michael Stumm, and Ding Yuan, *University of Toronto*

https://www.usenix.org/conference/fast22/presentation/li

This paper is included in the Proceedings of the 20th USENIX Conference on File and Storage Technologies.

February 22–24, 2022 • Santa Clara, CA, USA

978-1-939133-26-7

Open access to the Proceedings of the 20th USENIX Conference on File and Storage Technologies is sponsored by USENIX.

ctFS: Replacing File Indexing with Hardware Memory Translation through Contiguous File Allocation for Persistent Memory

Ruibin Li, Xiang Ren, Xu Zhao, Siwei He, Michael Stumm, Ding Yuan
University of Toronto

Abstract

Persistent byte-addressable memory (PM) is poised to become prevalent in future computer systems. PMs are significantly faster than disk storage, and accesses to PMs are governed by the Memory Management Unit (MMU) just as accesses with volatile RAM. These unique characteristics shift the bottleneck from I/O to operations such as block address lookup – for example, in write workloads, up to 45% of the overhead in ext4-DAX is due to building and searching extent trees to translate file offsets to addresses on persistent memory.

We propose a novel *contiguous* file system, ctFS, that eliminates most of the overhead associated with indexing structures such as extent trees in the file system. ctFS represents each file as a contiguous region of virtual memory, hence a lookup from the file offset to the address is simply an offset operation, which can be efficiently performed by the hardware MMU at a fraction of the cost of software maintained indexes. Evaluating ctFS on real-world workloads such as LevelDB shows it outperforms ext4-DAX and SplitFS by 3.6x and 1.8x, respectively.

1 Introduction

The emergence of byte-addressable persistent memory (PM) fundamentally blurs the boundary between memory and persistent storage. Intel's Optane DC persistent memory is byte-addressable and can be integrated as a memory module. Its performance is orders of magnitude faster than traditional storage devices: the sequential read, random read, and write latencies of Intel Optane DC are 169ns, 305ns, and 94ns, respectively, which are the same order of magnitude as DRAM (86ns) [19].

A number of file system designs have been introduced with the aim of exploiting the characteristics of PM. For example, Linux introduced Direct Access support (DAX) for some of its file systems (ext4, xfs, and ext2) that eliminates the use of the page cache. Other designs bypass the kernel by mapping different file system data structures into user space to reduce the overhead of switching into the kernel [7, 8, 21, 25, 37]. SplitFS, a state-of-the-art PM file system, aggressively uses memory-mapped I/O [21] for significantly improved performance.

All of these systems use conventional tree-based index structures for translating the file offset to the device address. This index structure was first proposed by Unix in the 70s [34] when the speed of memory and persistent storage differed by several orders of magnitude. However, with the emergence of PM, this speed difference has shrunk significantly to the point of being almost negligible. This in turn has shifted the bottleneck from I/O to file indexing overheads.

Indeed, we show in §2 that this indexing overhead can be as high as 45% of the total runtime for write workloads on ext4-DAX (e.g., for Append). While memory-mapped I/O (`mmap()`) can mitigate some of the indexing overhead [11], it does not remove indexing overhead but only shifts its timing to page fault handling or `mmap()` (when pre-fault is used). For example, §2 shows that with SplitFS, file indexing overhead can be as high as 63% of the Append workload runtime. This is 18% higher than that of ext4-DAX, even though the runtime of Append is lower on SplitFS; this is because SplitFS's improved performance further shifts the bottleneck and exacerbates indexing overhead.

An alternative to using file indexing is to use contiguous file allocation. While simple contiguous allocation designs with fix-size or variable-size partitions are known [36], they face two major design challenges: (1) internal fragmentation for fix-size partitions, (2) external fragmentation for variable-size partitions, and (3) file resizing, specifically for expansion which often requires

costly data movement. Therefore, the only use of contiguous file allocation in practice today is on CD-ROMs, where files are read-only [36]. SCMFS [39] proposed the high-level idea of allocating files contiguously in virtual memory. However, it does not address the challenges of contiguous file allocation, namely how files are allocated and how resizing is managed. (Its implementation and evaluation are also only based on simulations).

We propose ctFS, a contiguous file system designed from the ground up for PM. ctFS has the following key designs elements:

- Each file (and directory) is contiguously allocated in the 64-bit *virtual* memory space. We demonstrate the practicality of this idea, given that the 64-bit address space is enormous. Furthermore, the virtual address space is carefully managed by a hierarchical layout, similar to the buddy memory allocation [23], in which each partition is subdivided into 8 equal-size subpartitions. This design speeds up allocation, avoids external fragmentation, and minimizes internal fragmentation (§3.2).
- A file's virtual-to-physical mapping is managed using *persistent page tables* (PPT). PPTs have a similar structure as the regular, volatile page tables in DRAM, except that PPTs are stored persistently on PM. Upon a page fault on an address that is within a ctFS's memory region, the OS looks up the PPT and creates the same mappings in the DRAM-based page tables. Therefore, subsequence accesses are served by hardware MMU from DRAM-based page tables, avoiding the indexing cost.
- Initially, a file is allocated within a partition whose size is just large enough for the file. When a file outgrows its partition, it is moved to a larger partition in virtual memory without copying any physical persistent memory. ctFS does this by remapping the file's physical pages to the new partition using *atomic swap*, or `pswap` (§3.3), a new OS system call that *atomically* swaps the virtual-to-physical mappings. Atomic swap also enables efficient crash consistency on multi-block writes without needing to double-write the data. An atomic write in ctFS simply writes the data to a new space, and then `pswap`s it with the old data (§3.4).

In ctFS, the translation from file offset to the physical address now needs to go through the virtual-to-physical memory mapping, which is no less complex than the conventional file-to-block indexes. The key difference is that page translation can be sped up by existing hardware support. Translations that are cached by TLB will be handled transparently from the software and completed in one cycle. In contrast, a file system's file-to-block translation can only be cached by software. Additionally, ctFS can adopt various optimizations for memory mapping, such as using huge pages, to further speed up a variety of operations.

Our evaluation on Intel Optane DC reveals that ctFS can eliminate most indexing overheads, which results in up to a 7.7x and 3.1x speedup over ext4-DAX and SplitFS [21] on the Append workload. ctFS further improves the throughput of LevelDB running YCSB by up to 3.62x, 1.82x, 3.21x, and 2.45x when compared to ext4-DAX, SplitFS, Nova [40], and PMFS [8], respectively. Finally, ctFS improves RocksDB [35] performance by up to 1.6x when compared to ext4-DAX. The source code of ctFS is available at `https://github.com/robinlee09201/ctFS`.

A limitation of ctFS is that we implement it as a user-space library file system that trades protection for performance. While this squeezes the most performance out by aggressively bypassing the kernel, it sacrifice protection in that it only protect against unintentional bugs instead of intentional attacks. We envision that this is an acceptable, or even desirable, trade-off for data center environments. We discuss other limitations in §5.

2 Understanding File Indexing Overhead

We analyzed the performance overhead of block address translation in Linux's ext4-DAX and in SplitFS [21]. Ext4-DAX is the port of the ext4 extent-based file system to PM. It eliminates the page cache, and directly accesses PM using memory operations (`memcpy()`).

Background on SplitFS. We briefly describe SplitFS for a better understanding. SplitFS splits the file system logic into a user-space library (U-Split) and a kernel space component (K-Split), where K-Split uses ext4-DAX. A file is split into multiple 2MB regions by U-Split, where each region is mapped to one ext4-DAX file. Both U-Split and K-Split participate in indexing: U-Split maps a logical file offset to the corresponding ext4-DAX file, and the ext4-DAX in K-Split further searches its extent index to obtain the actual physical address.

SplitFS also proposed a novel operation called `relink` to improve the performance of file expansion and provide crash consistency on file writes without double-writing data. Under its sync mode, file appends are first made to a staging file, and then relinked to the target file either when `fsync()` gets called or the staging file reaches its size limit; file overwrites are applied in-place. Under its strict mode, every file write, whether it's overwriting or appending data, is applied to a staging file and gets

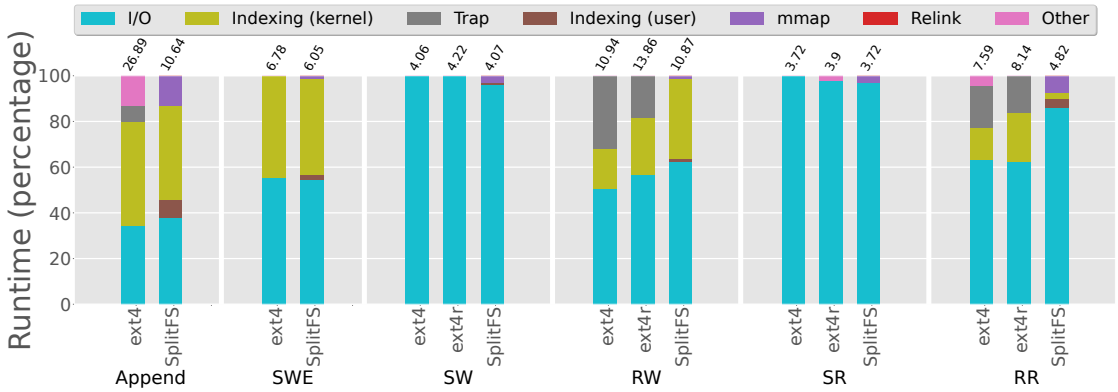

Figure 1: **Performance breakdown** (in percentage) of ext4-DAX and SplitFS on persistent memory. The number above each bar is the total run time in seconds.

relinked at the end of every write. Hence, the indexing time of SplitFS consists of `relink`, mmap, and indexing in both kernel and user components.

Experimental Methodology. Our experiments were conducted on a server with two 128GB Intel Optane DC persistent memory (PM) modules, an 8-core Intel Xeon 4215 CPU running at 2.5 GHz, and 96 GB of DRAM. We used Linux version v5.7.0-rc7+.

We ran a total of 6 tests. The results are shown in Figure 1. Each test either reads or writes a 10GB file. The first test, Append, repeatedly *appends* 4KB of data to a file which is initially empty. The second test, SWE, sequentially writes a total of 10GB of data to an *empty* file with 10 `pwrite()` calls to write 1GB at a time. RR reads 4KB of data from a random (4KB-aligned) offset in a 10GB file, and RW overwrites an existing 10GB file with 4KB of data at a random (4KB-aligned) offset, and they do this 2,621,440 times. Finally, SR/SW we sequentially reads/writes 10GB data, 1GB at a time.[1]

For the SW, RW, RR, and SR tests, we ran the ext4-DAX tests with two types of files: those that were *sequentially* allocated (ext4) and those that were *randomly* allocated (ext4r). Sequentially allocated files were created by SWE, which maximizes ext4-DAX's grouping of blocks into an extent. Randomly allocated files were created by writing to them similarly to the way RW does, except that the file is initially empty (Linux file systems support sparse files); these randomly allocated files limit ext4-DAX's ability to group blocks into extents. The "ext4r" bars in RW, RR, and SR represent tests that operated on such randomly allocated files. Note that ext4-DAX creates 12 extents for a sequentially allocated 10GB file, but creates 256 extents for a randomly allocated file. For SplitFS, all files are sequentially allocated.

Indexing overhead in ext4-DAX. Figure 1 shows the breakdown of the completion time of each test. For ext4-DAX, we observe that indexing overhead is significant in Append and SWE, spending at least 45% of the total runtime on indexing.[2]

For the random access workloads, RR and RW, the proportion of time spent on indexing is lower, but still considerable: 25% and 21% of the total runtime when randomly writing and reading to/from a randomly allocated file (ext4r), and 18% and 15% when the file was sequentially allocated.

Indexing overhead in SplitFS. Figure 1 also shows the breakdown of the completion time of SplitFS's sync mode. [3] Compared to ext4-DAX, SplitFS spends an even higher proportion of the total runtime on indexing in the Append (63%), SWE (45%), and RW workloads (38%), while it spends 14% of the runtime on indexing in RR.

To understand SplitFS's indexing overhead in more detail, consider the Append workload where SplitFS spends a total of 6.62s on indexing. Three components make up this file indexing time: (1) the kernel indexing time as part of page fault handling (4.37s), (2) U-split's file indexing time (0.84s) spent on mapping file offsets to the correct ext4-DAX file, and (3) U-Split's `mmap()` time (1.39s).

3 Design & Implementation of ctFS

This section starts with an overview of ctFS. Then we describe the file system layout (§3.2), and how ctFS interacts with the kernel's memory management system (§3.3). We then explain ctFS's primitive for atomic operations — `pswap()`, and how ctFS handles file updates and en-

[1] We found that the version of SplitFS we tested does not support append operations that write over 128MB under its sync mode. Therefore, in SWE, we write 128MB at a time in SplitFS, instead of 1GB as in ext4-DAX and other the file systems we discuss in §4.

[2] In both cases, the index time includes the time to build the index.

[3] We only show its sync mode result as its semantics is comparable to that of ext4-DAX. SplitFS's strict mode is further evaluated in §4.

Mode	Atomicity		Similar to
	data	metadata	
sync	✗	✓	NOVA-relaxed, PMFS, SplitFS-sync
strict	✓	✓	NOVA-strict, Strata, SplitFS-strict

Table 1: The two modes provided by ctFS.

sures crash consistency (§3.4). Finally we discuss some optimizations (§3.5) and the protection model (§3.6).

3.1 Design Overview

ctFS is a high-performance PM file system that directly accesses and manages both file data and metadata in user space. Each file is stored contiguously in virtual memory, and ctFS offloads traditional file systems' offset to block number indexing to the memory management subsystem. ctFS achieves the following design goals:

- **POSIX compliance:** ctFS currently supports over 30 commonly used functions from the POSIX-compatible file system API.
- **Synchronous writes:** Write operations on ctFS are always synchronous, i.e., writes are persisted on PM before the operation completes. Hence there is no need for `fsync` (which does nothing in ctFS).
- **Crash consistency:** ctFS supports both file data consistency (by using `pswap`) and metadata consistency (by using conventional redo logs).
- **Concurrent operations:** ctFS supports concurrent operations on different files or concurrent reads on the same file; a reader-writer lock is used for each file to synchronize concurrent accesses.

Similar to prior systems, such as NOVA [40] and SplitFS [21], ctFS offers two modes, sync and strict, as shown in Table 1. Both modes ensure atomic metadata operations that include directory operations. Strict mode further ensures file data writes are atomic (by using `pswap`).

ctFS's architecture, shown in Fig. 2, consists of two components: (1) the user space file system library, ctU, that provides the file system abstraction, and (2) the kernel subsystem, ctK, that provides the virtual memory abstraction. ctU implements the file system structure and maps it into the *virtual* memory space. ctK maps virtual addresses to PM's physical addresses using a *persistent page table (PPT)*, which is stored in PM. Any page fault on a virtual address inside ctU's address range is handled

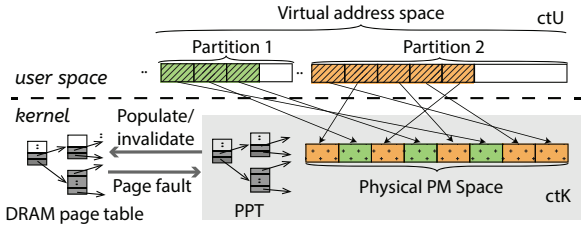

Figure 2: **Architecture of ctFS.** Each box represents a page. Two partitions are shown. The file allocated in partition 1 uses 3 pages (green), and the file in partition 2 uses 5 pages. ctK maintains virtual-to-physical page mappings in the PPT.

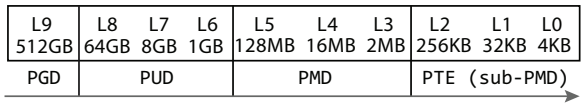

Figure 3: **Size of partitions at levels L0 to L9.** PGD, PUD, PMD, and PTE refer to the four levels of page tables in Linux (from highest to lowest). An L9 partition aligns with PGD, i.e., its starting address has zero in all of the lower level page tables (PUD, PMD, PTE); Similarly, L6-L8 partitions align with PUD, whereas L3-L5 partitions align with PMD.

by ctK. If the PPT does not contain a mapping for the fault address, ctK will allocate a PM page, establish the mapping in the PPT, and then copy the mapping from the PPT to the kernel's DRAM page table, allowing virtual to PM address translations to be carried out by the MMU. When any mapping in the PPT becomes obsolete, ctK will remove the corresponding mapping from the DRAM page table and shoot down the mapping in the TLBs.

With this architecture, there is a clear separation of concerns. ctK is *not* aware of any file system semantics, which is entirely implemented by ctU using memory operations. Next, we discuss the designs that are specific to ctFS. We omit the designs that are similar to existing file systems. For example, we use standard transaction logging to provide crash consistency of *metadata*, including directories, inode, and ctFS data structures such as partition headers, bitmaps, etc.

3.2 File System Structure (ctU)

ctFS's user-space library, *ctU*, organizes the file system's *virtual* memory space into hierarchical partitions to facilitate contiguous allocations. The size of each partition at a particular level is identical, and each level's size is 8x the size of the partitions at the next lower level. Figure 3 shows the sizes of the ten levels that ctFS currently supports. The lowest level, L0, has 4KB partitions, whereas the highest level, L9, has 512 GB partitions. ctFS can be easily extended to support more partition levels, *e.g.* L10

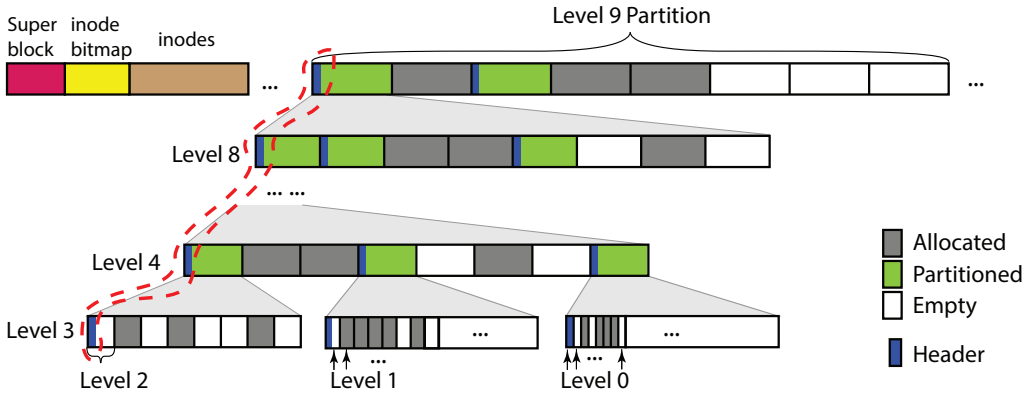

Figure 4: **Layout of ctFS in the *virtual* address space (VAS).** The space of an entire partition is reserved in VAS, whereas the physical PM space is allocated on-demand based on actual usage. Headers circled in the dashed-line reside on the same page.

(4TB), L11 (32TB), *etc*.

A file or directory is always allocated contiguously in one and only one partition, with the size of the partition being the smallest capable of containing the file. For example, a 1KB file is allocated in an L0 partition (4KB); a 2GB file is allocated in an L7 partition (8GB).

We chose each next level to be 8x the size of the previous level because the boundary of the levels should align with the boundary of Linux page table levels (Figure 3). This enables the optimization during `pswap` we describe in §3.3. Therefore, our only options for partition size differences were: 2x (2^1), 8x (2^3), or 512x (2^9). We chose 8x because 2x would be too small and 512x too large.

File System Layout. Figure 4 shows the layout of ctFS. The virtual memory region is partitioned into two L9 partitions. The first L9 partition is a special partition used to store file system metadata: a superblock, a bitmap for inodes, and the inodes themselves. Each inode stores the file's metadata (e.g., owner, group, protection, size, etc.) and *a single field* identifying the virtual memory address of the partition that contains the file's data. The inode bitmap is used to track whether an inode is allocated or not. The second L9 partition is used for data storage. [4]

Each partition can be in one of the three states: Allocated (A), Partitioned (P), or Empty (E). A partition in state A is allocated to a single file; a partition in state P is divided into eight next-level partitions. We call the higher level partition the *parent* of its eight next-level partitions. This parent partition *subsumes* its eight child partitions; i.e., these 8 child partitions are sub-regions within the virtual memory space allocated to the parent. For example, in Figure 4, an L9 partition in state P is divided into 8 L8 partitions. The first L8 partition is also in state P, which means it is divided into 8 L7 partitions, and so on. In this manner, the different levels of partitions form a hierarchy.

This hierarchy of partitions has three properties. (1) For any partition, all of its ancestors must be in state P; and any partition in the A or E state does not have any descendants. (2) Any address in a partition is also an address in the partitions of its ancestors; e.g., any L3 partition in Figure 4 is contained in its ancestor L4-L9 partitions. (3) The starting address of any partition, regardless of its level, is aligned to its partition size; this is the case as long as the top-level L9 partitions are 512 GB aligned.

Partition Headers. ctU needs to maintain book keeping information for each partition, such as its state. To store such metadata, each partition in P-state has a header which contains the state of each of its *child* partitions; ctU stores the header directly on the first page of the partition for fast lookup that does not involve indirections. For example, for each partition in P state at levels L4-L9, the state of its eight children are encoded using 2 bits packed into a `uint16_t`.

To speed up allocation, the header also has an availability level field that identifies the highest level at which a *descendent* partition is available for allocation. For example, the availability level of the left-most L9 partition in Figure 4 is 8 because this L9 partition has at least 1 L8 child partition in E state. With this information, when allocating a level-N partition, if a P partition's availability level is less than N, ctU does not need to drill down further to check its child partitions. This results in constant worst-case time complexity for allocating a parition and is far more efficient than using bitmaps.

Because ctU places the header in the first page of a partition in P state, its first child partition will also contain the same header, and as a result, this first child partition must also be in P state; it cannot be in the Allocated

[4] Note that the 512GB allocated for metadata is virtual memory; The physical pages underneath it are allocated on demand.

state because the first page would need to be used for file content. Therefore, a header page can contain the headers of multiple partitions in the hierarchy. For example, in Figure 4, the headers in the dashed circle are all stored on the same page. This is achieved by partitioning the header page into non-overlapping header spaces for each level from L4-L9.

ctU does not partition L0–L3 further, as the 4KB header space becomes much more wasteful for smaller partition sizes. Instead, each L3 partition (2MB) can only be partitioned as (1) 512 L0 child partitions, (2) 64 L1 child partitions, or (3) 8 L2 child paritions, as shown at the bottom of Figure 4. As a result, there is only one header in each L3 partition that is in state P, and it contains a bitmap to indicate the status of each of its child partitions, which can only be in either state A or E, but not P.

Virtual Memory Allocation. During system initialization, ctU allocates a 1TB, empty (i.e., not backed) virtual memory area (VMA) to accommodate two L9 partitions. It does not restrict the starting address of this VMA, so it can be anywhere in the virtual address space (as long as it is aligned). If the PM size is larger than 512GB, then the next level (L10) would be used, and an 8TB VMA would be allocated. Note that subsequent virtual memory allocations made from the kernel or processes will not clash with ctU's VMA, because the Linux kernel's VMA allocation will only allocate a VMA if it does not conflict with existing VMAs.

TLB usage. ctFS does not use more TLB entries compared to other file systems. In conventional (non-DAX) file systems, the file data will be buffered in memory, either in the file system's buffer cache, or by the process in the case of memory mapped I/O. Such buffering will occupy TLB entries just as ctFS does, and the number of entries used depend on the amount of data a process accesses. Ext4-DAX eliminates the buffer cache by directly accessing the devives using statically mapped virtual kernel addresses. However, this mapping still goes through the page table [14] and hence still occupies TLB entries. Therefore even compared to ext4-DAX, ctFS does not use more TLB entries.

3.3 Kernel Subsystem Structure (ctK)

ctK manages the PPT. PPT is essentially identical to a regular Linux 4-level DRAM page table, except (1) it is persistent, and (2) it uses relative addresses for both virtual and physical addresses. It uses relative addresses because ctFS's memory region may be mapped to different starting virtual addresses in different processes

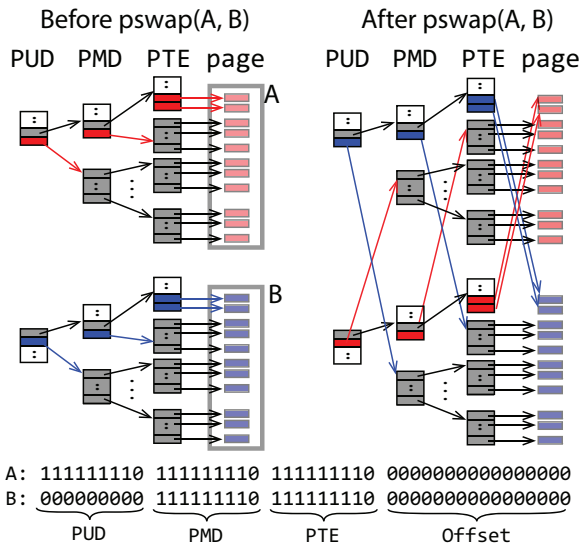

Figure 5: **An example of `pswap`.** The shaded entries in the page tables are the ones used to map the two-page arrays A and B. The red and blue page table entries are the ones that are modified by `pswap`. Before `pswap`, A maps to the red pages and B maps to the blue pages, whereas after `pswap` A maps to blue pages and B maps to red pages. The last 39 bits of A and B's address are shown at the bottom.

due to *Address Space Layout Randomization* [6] [9], and hardware reconfiguration could change PM's starting physical address. We also note that whereas each process has its own DRAM page table, ctK has a single PPT that contains the mapping of all virtual addresses in ctU's memory range (i.e., those inside the partitions). The PPT cannot be accessed by the MMU, so mappings in the PPT are used to populate entries in the DRAM page table on demand as part of page fault handling.

ctK provides a `pswap` system call that atomically swaps the mapping of two same-sized contiguous sequences of virtual pages in the *PPT*. It has the following interface:

```
int pswap(void* A, void* B, unsigned int N,
    int* flag);
```

`A` and `B` are the starting addresses of each page sequence, and `N` is the number of pages in the two sequences. The last parameter `flag` is an output parameter. Regardless of its prior value, `pswap` will set `*flag` to 1 if and only if the mappings are swapped successfully. ctU sets `flag` to point to a variable in the redo log stored on PM and uses it to decide whether it needs to redo the pswap upon crash recovery. `pswap` also invalidates all related DRAM page table mappings (and shoots them down in TLBs), as we found it is more efficient than updating the mappings.

The `pswap()` system call *guarantees crash consis-*

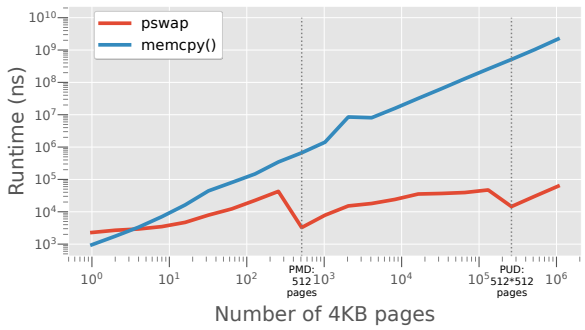

Figure 6: **Comparing the performance of `pswap` and `memcpy`.** Both the X and Y axis are log scale.

tency: it is atomic, and its result is durable as it operates on PPT. Moreover, concurrent `pswap()` operations occur as if they are serialized, which *guarantees isolation* between multiple threads and processes. [5]

`pswap` avoids swapping every target entry in the PTEs (the last level page table) of the PPT whenever possible. Figure 5 shows an example where `pswap` needs to swap two sequences of pages - A and B - each containing 262,658 ($512 \times 512 + 512 + 2$) pages. `pswap` only needs to swap 4 pairs of page table entries or directories (as shown in red and blue colors in Figure 5), as all 262,658 pages are covered by a single PUD entry (covering 512×512 pages), a single PMD entry (covering 512 pages), and two PTE entries (covering 2 pages).

`pswap()` can only perform this optimization if the starting addresses of the two page sequences are *swap-aligned*. We first define the *reach* of a page table *level* to be the size of the memory region that each entry maps — e.g., the reach of PTE, PMD, PUD, and PGD are 4K (bytes), 2M, 1G, and 512G, respectively. Given two contiguous sequences of pages in virtual memory that start at addresses A and B, and given that each sequence spans a memory region of size S, let L be the highest level in the page table such that $reach(L) \leq S$. We then say that the two page sequences A and B are swap-aligned if and only if:

$$A \bmod reach(L) = B \bmod reach(L)$$

In the example of Figure 5, L is PUD, and $reach(L)$ is 1G (2^{30}). $A \bmod reach(L)$ equals $B \bmod reach(L)$ because the last 30 bits of A and B are the same.

Figure 6 shows the performance of `pswap` as a function of the number of pages that are swapped. We compare it with the performance of the same swap implemented with `memcpy` that approximates the use of conventional write ahead or redo logging that requires copying *data* twice. The `pswap` curve shows a wave-like pattern: as the number of pages increases, pswap latency first increases and then drops back as soon as it can swap one entry in a higher-level page table instead of 512 entries in the lower-level table. The two drop points in Figure 6 are when N is 512 (mapped by a single PMD entry) and 262,144 (mapped by a single PUD entry). In comparison, `memcpy`'s latency increases linearly with the number of pages. When N is 1,048,576 (representing 4GB of memory), `memcpy` takes 2.2 seconds, whereas `pswap` takes only 62μs. However, when N is less than 4, `memcpy` is more efficient than `pswap`.

[5] `pswap` uses conventional redo log to ensure crash consistency.

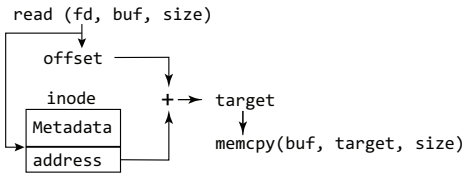

Figure 7: **Implementing `read()` on ctFS.**

Concurrent invocations to `pswap`() will only be serialized if they operate on overlapping memory ranges. We use a binary search tree to store the ranges of concurrent, on-going `pswap`()s.

3.4 File System Operations

Since files are contiguous in virtual memory, read and write operations require special treatment. Other operations that operate on metadata (i.e., directories and metadata in inodes) are similar to those on conventional file systems.

Figure 7 shows how `read()` is implemented in ctFS. Given the file `offset` (from the file descriptor), ctU locates the inode, and further locates the starting address of the file. It adds `offset` to this starting address, which is the virtual address of the data to be read. Then, it uses a single `memcpy`() to copy the data to the user buffer. *All of these operations are performed by the user space ctU.*

ctFS allocates a partition on-demand on the first write to a file. It always allocates the smallest partition that is big enough to store the write. Later when the file size increases beyond the partition size, ctFS will "upgrade" it to the next higher level partition that can accommodate the file. Also recall that ctFS supports two modes, where strict mode offers atomic writes. Consequently there are two special write cases: one that triggers an upgrade and one that requires atomicity. In the normal case where neither applies, write does not differ from read. Both of the special cases use `pswap`, and in both cases ctU *guarantees that the starting addresses are swap-aligned*. Next, we explain each case.

Write with Upgrade. When a write (append) triggers an

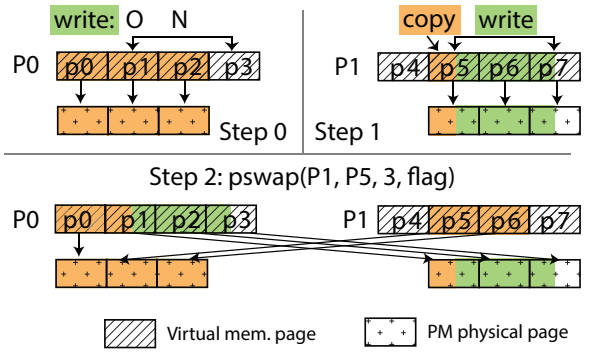

Figure 8: **An example of *atomic write* using `pswap`.** The yellow color indicates the original file content whereas green indicates new data to be written.

upgrade, ctFS will first relocate the file to a new partition before applying the write. It also maintains a redo log to ensure crash consistency of the upgrade. Say a write requires upgrading from a level L partition, P0, to a level M partition, P1 (where $M > L$). ctU first allocates P1 in *virtual memory*. It then calls `pswap (P0, P1, N, flag)`, where N is the number of pages in the level L partition. Note that right after the partition allocation, P1 does not map to any PM pages; therefore, after `pswap()`, P1 points to the PM pages that used to map to P0, and P0 is no longer mapped. Both steps will first be recorded in the redo log, and `flag` is located in the redo log, so if a crash occurs ctU knows whether pswap had completed successfully or not. If a crash happens before `pswap` completes, ctU only needs to free P1. If a crash happens right after `pswap` has completed, then ctU will continue to finish the upgrade by changing the starting address in the file's inode to P1. Partition "downgrades" (e.g., when the file is truncated) are handled in a similar manner.

Atomic Write. In strict mode, ctFS handles an atomic write using a write-and-swap protocol. Assume a write that writes N bytes to offset O of a file in a level L partition, P0. Figure 8 shows an example, where O is *not* page aligned, and N spans three pages where the last page, p3, has not been accessed and is hence not mapped to PM. ctU carries out the following two steps.

Step 1: ctU first allocates a *staging* partition, P1, that is also at level L. It then writes the new data to the *same offset* O in P1. If O is not page-aligned, as is the case in Figure 8, ctU copies the data fragment between the start of the first page and O in P0 to P1, and similarly, it will copy any fragment data at the end if O+N is not page aligned. Note that ctU does *not* need to copy any pages that are not modified.

Step 2: ctU invokes `pswap()` to atomically swap the page sequence in P0 with its corresponding sequence in P1. In Figure 8, it `pswap`s pages p1–p3 in partition P0 with pages p5–p7 in partition P1.

To handle crash consistency, ctU uses the redo log that records the status of each step, and the flag used in `pswap`() is located on this redo log.

3.5 Other Optimizations

Huge Page. ctK allocates huge pages (2MB pages) whenever possible. Because the address of any partition is aligned with the partition size, all files that reside in level L3 or above benefit from huge pages. However, when ctU performs `pswap` with small N, huge pages may have to be broken into base pages, adding extra overhead to `pswap`(). Note that `pswap` can apply its optimization whenever the page sequences are swap-aligned regardless of whether they are huge pages or not. Huge pages are enabled in our evaluation unless otherwise noted. In §4.1.3, we evaluate and explain the impact of huge pages in details.

Optimized append in strict mode. ctFS performs an optimization on append operations by exploring the invariant between a file's metadata and its data [4, 12]. Instead of using the write-and-swap protocol, it directly appends the new data and then changes the file size in the inode. If a crash occurs before the append completes, the file will be consistent, as the file size still has the old value, presenting a view as if the append did not occurr.

Instruction choices in `memcpy()`. We experimented with different memory copy strategies (e.g. repeat instructions, non-temporal instructions, cache flush) and found that AVX512 [1] non-temporal 512-bit load and store (VMOVDQU and MOVNTDQ) performed the best, resulting in a 5%–20% performance gain over what SplitFS and ext4-DAX uses.

3.6 Protection

For protection, ctFS's exloits both Intel Memory Protection Keys (MPK) and regular page table protection. We first explain Intel MPK before discussing ctFS's design.

Background on Intel MPK. MPK allows each memory page to be tagged with one of 16 protection keys, K0, K1, ..., K15. Four unused bits in each page table entry are used to store the key for the page. Each key's protection rights can be changed from user space, *using a special instruction*. For example, key K0's right can be set to no access, K1 can be set to read only, and K2 can be set to read/write. The access rights associated with each key are stored in a register called `PKRU`. Hence the access rights are thread-local (as each core has its own `PKRU` register).

A key advantage of using MPK over the conventional

`mprotect()` system call is performance. While assigning a key to a page still requires a system call, setting/changing the access permission associated with each key is a user-space instruction that only consumes around 20 cycles [33].

Protection in ctFS. ctFS tags each page within ctFS's memory region with a single MPK key, which we refer to as NONE. When a ctFS operation is invoked, ctU sets the access right for the NONE key to be read/write, and it resets the access right back to no access before returning. Therefore, any access to ctFS's memory space from outside of ctFS will be prevented. If multiple processes with different access rights access the same file concurrently, ctFS can protect the same page differently for different processes as the access rights for the same key, NONE, can be set differently on different cores.

This protection strategy protects ctFS against *unintentional* bugs. For example, a dangling pointer in an application won't be able to accidentally corrupt the file system, given that changing the rights associated with the key requires a special instruction. However, this design does not protect against *intentional* attacks. For example, a malicious application could intentionally set the rights for the NONE key to be read/write and modify the file system in an arbitrary manner.

4 Evaluation

In this section, we present the results of evaluating ctFS against other PM file systems (FS) using both real-world applications and microbenchmarks. The server and OS settings used in our evaluation are as described in §2.

4.1 Micro-benchmarks

We evaluate the performance of ctFS and compare it with that of SplitFS, ext4-DAX, PMFS, and NOVA, using the same 6 micro-benchmarks as in §2. We repeat each experiment 10 times and report the average. In all experiments, ctFS uses demand paging and does not prepopulate any pages in order to accentuate the maximum page fault handling overhead in ctFS. SplitFS prefaults staging files at its system bootup time so there are no page faults on those files.

4.1.1 Append

Append is particularly relevant as the append operation is the dominant file system operation of many application [21], and it is the operation on which SplitFS shows the most significant speedup. Figure 9 shows the performance of Append.

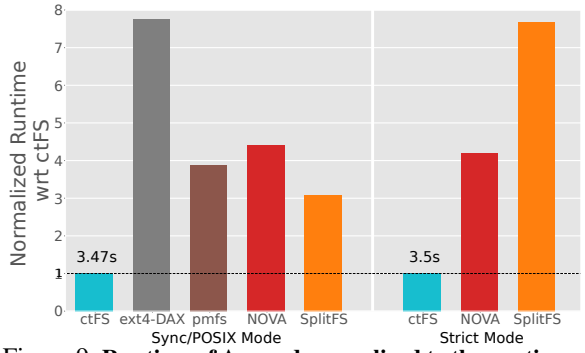

Figure 9: **Runtime of Append normalized to the runtime of ctFS.** The different file system and configuration combinations are grouped by the crash consistency guarantees on file data.

ctFS achieves a 7.7x speedup against ext4-DAX for Append in sync mode. 45% of ext4-DAX's runtime is in building and searching indices as it has to allocate many small extents. Even if we deduct kernel trap overhead (shown in Figure 1) from the runtimes of ext4, ctFS-sync still achieves an 7.0x speedup. This shows the benefit of using contiguous file allocation, regardless of whether it is a user-space or kernel-space implementation.

While SplitFS is able to significantly reduce the indexing time by using memory-mapped I/O, ctFS still achieves 3.1x speedup over SplitFS in sync mode. Specifically, SplitFS takes 7.2s longer than ctFS to run Append, and 92% (6.62s) of that time comes from indexing. The other 8% of the speedup comes from ctFS's improved I/O performance. In contrast, ctFS successfully eliminates most of the overhead of file indexing, primarily by having the MMU perform the indexing in hardware during memory page translation. (See Figure 11 for a breakdown of ctFS's runtime.) It only spends 24ms in page fault handling, compared to SplitFS's 4.4s of page fault handling (§2). Even though ctFS has a similar number of page faults (525,490) as SplitFS (578,260), SplitFS triggers page faults whose handling requires file indexing, whereas all of ctFS's page faults are minor faults.

For the Append workload, whether running in sync or strict mode does not affect ctFS performance because of ctFS's append optimizations (§3.5); ctFS achieves 7.66x speedup over SplitFS in strict mode.

Compared to NOVA's sync mode and pmfs, ctFS-sync achieves 4.4x and 3.87x speedups, respectively.

4.1.2 Other Micro-benchmarks

Figure 10 shows ctFS's performance compared to that of ext4-DAX and SplitFS for the other 5 microbenchmarks. In sync mode, ctFS achieves an average speedup of

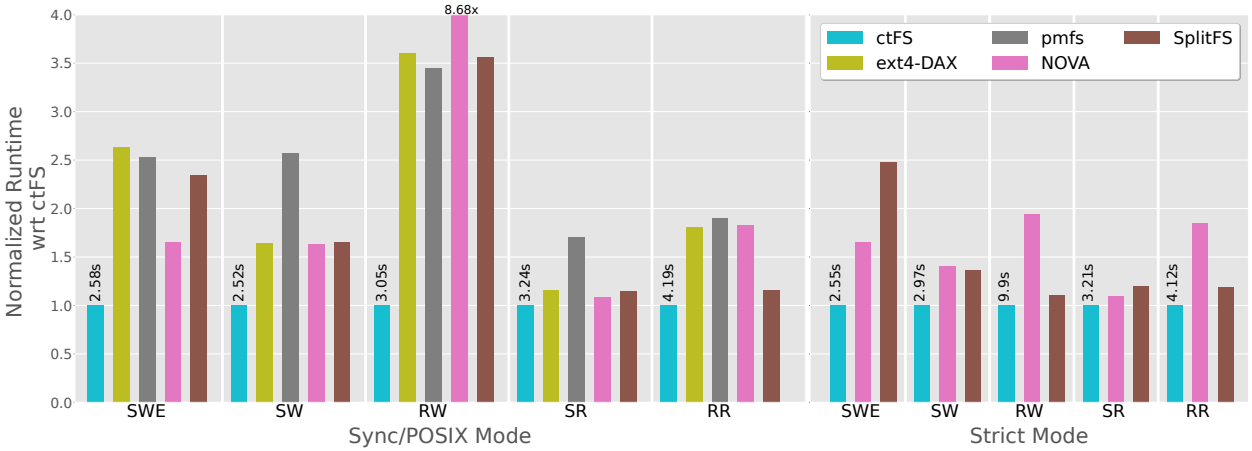

Figure 10: **Performance comparison** of ext4-DAX, Nova, PMFS, SplitFS, and ctFS for five micro-benchmarks.

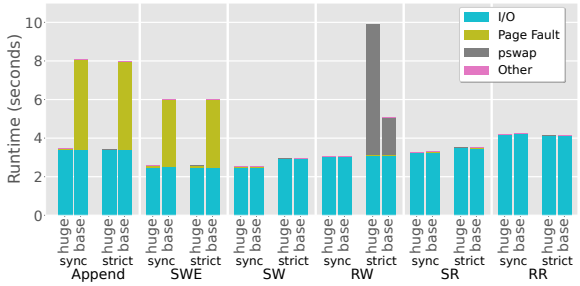

Figure 11: **ctFS overhead breakdown** under four configuration combinations: with huge pages enabled and disabled, while running in sync and strict mode.

2.17x, 1.97x, 2.43X, 2.97X, against ext4-DAX, SplitFS-sync, pmfs, and NOVA, respectively. In strict mode, ctFS achieves an average speedup of 1.46x and 1.59X against SplitFS-strict and NOVA-strict.

4.1.3 ctFS Runtime Breakdowns

Figure 11 shows the breakdown of ctFS's runtime on each test while running in sync and strict mode, and with huge pages enabled and disabled. We first consider the difference between ctFS's sync and strict modes. Recall that ctFS invokes `pswap` at the end of file overwrite operations under strict mode. This affects both RW and SW. In RW, 68% of the run time of ctFS-strict is spent on `pswap`. This test represents the worst-case scenario for ctFS-strict, as each write triggers a `pswap` at the smallest granularity (4KB page): `pswap` cannot perform any optimizations when swapping the entries in the last-level page table, and it is foreced to break up the huge pages into base pages.[6] In comparison, while ctFS also needs to invoke `pswap` in SW when running in strict mode, because `pswap` is only invoked once at the end, it incurs negligible overhead (5.7ms).

The figure also shows the difference between having huge pages enabled and disabled. With huge pages enabled, ctFS indeed eliminates much of the indexing overhead, as all workloads are bottlenecked by I/O, except for the RW workload when ctFS runs in strict mode. With huge pages disabled, both the persistent page table (PPT) and the DRAM page table have 512x more entries, and each TLB entry now only maps 4KB instead of 2MB. For SW, RW, SR, and RR, the overhead after disabling huge pages is negligible in both sync and strict modes (at most 3.4% in SR-strict). This indicates that the overhead of additional TLB misses is negligible. In RR, for example, there are 512x more TLB misses with huge pages disabled, yet this still results in negligible overhead. Note that the number of page faults remains the same even when huge pages are disabled, because ctK copies 512 page table mappings (or the mappings for a 2MB region) from the PPT to the DRAM page tables on each page fault. In comparison, the large overheads in Append and SWE come from allocating physical PM page frames and building the persistent page tables (PPT), because with only base pages, the PPT contains 512x more entries.

Interestingly, in RW, disabling huge pages results in a 2x speedup for ctFS-strict. This is because with huge pages enabled, every write, which is at the granularity of a base page (4KB), will trigger a `pswap` that breaks the huge page and causes TLB shootdowns. In comparison, when huge pages are disabled, there is no need to break up huge pages.

4.2 Real-world Applications

We evaluated ctFS using LevelDB [28] and RocksDB [35], both of which are persistent key-

[6] Even then, ctFS outperforms SplitFS and NOVA in strict mode as shown in Figure 10. SplitFS also uses huge pages, so that it also needs to break up huge pages, which makes up 37.6% of its runtime.

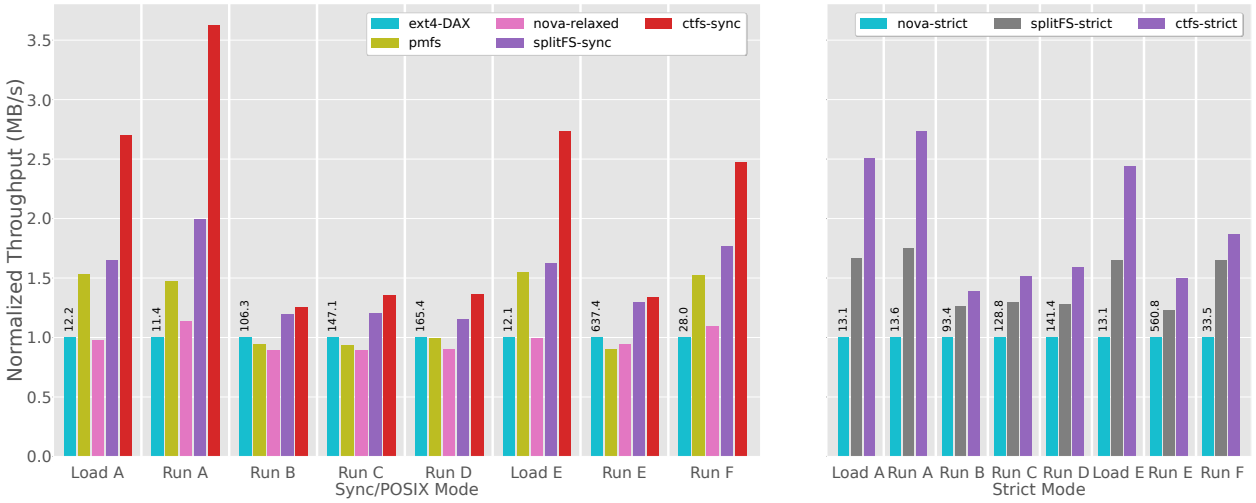

Figure 12: **YCSB on LevelDB.** Results are measured in throughput that is normalized to ext4-DAX in the sync/POSIX group and NOVA-strict in the strict group. The number on top shows the absolute throughput.

A	Update heavy: 50/50 read/write mix
B	Read mostly: 95/5 read/write mix
C	Read only: 100% read
D	Read latest: new records are inserted, and the most recently inserted records are read the most
E	Short ranges: short ranges of records are queried, instead of individual records
F	Read-modify-write: read a record, modify it, and write back the changes

Table 2: YCSB runs and their characteristics.

value stores. We drove LevelDB with the Yahoo! Cloud Serving Benchmark (YCSB) [5]. YCSB includes six different key-value workloads that are described in Table 2. We drove RocksDB using RocksDB's built-in benchmark `db_bench` with three workloads: *random fill*, which creates and adds key-value pairs; *random read*, which returns the values of given keys; and *random update*, which updates the values of given keys. Each of these tests carries out 5 million operations. Both LevelDB and RocksDB use `pwrite` and `pread` instead of memory-mapped I/O.

The LevelDB workloads demonstrate ctFS's performance advantage achieved by removing the indexing overheads in a real world application. The RocksDB workloads show that it is feasible and beneficial to replace write-ahead logs (WALs) with ctFS's atomic writes.

LevelDB. Figure 12 shows the performance of different PM file systems on LevelDB using the YCSB workloads. ctFS outperforms all the other file systems in each of the workloads when run at comparable consistency levels.

ctFS achieves the most significant improvement in throughput under write-heavy workloads: Load A and E and Run A, B, F. [7] Among these write-heavy workloads, ctFS-sync's throughput is 1.64x the throughput of SplitFS-sync on average, with 1.82x the throughput in the best-case (under Load E). In strict mode, ctFS's throughput is 1.30x higher than that of SplitFS on average, with 1.50x higher in the best-case (under Load A). Compared with ext4-DAX, ctFS-sync has 2.88x higher throughput on average and 3.62x higher throughput in the best case (under Run A).

On read-heavy workloads, ctFS's thoughtput is still higher than that of the other file systems, but by a smaller amount. It achieves an average of 1.25x - 1.36x higher throughput over ext4-DAX. As for SplitFS, recall from our microbenchmarks that it spends more time on indexing in random reads than sequential reads. This is why ctFS achieves better throughput than SplitFS in Run B, C, and D, which are dominated by random reads; e.g., ctFS's throughput is 1.18x and 1.25x higher than that of SplitFS under Run D when run in either sync or strict mode. For Run E, which is dominated by sequential reads, cfFS has 1.02x and 1.22x higher throughput.

By studying the breakdowns of ext4-DAX's time consumption, we observe that indexing time takes up 19.6%, 25%, and 24.5% of the total runtime of LoadA, RunA, and LoadE, respectively. Meanwhile, ctFS only spends a maximum of 0.2% on indexing overhead (in handling page faults) across all workloads. Hence, indexing accounts for 39.3%, 49.9%, and 46.4% of ctFS's speedup over ext4-DAX on these three workloads. Another 22.5%, 36.4%, and 33% of ctFS's speedups arise from a more efficient I/O data path over ext4-DAX.

[7] Load A and Load E create the respective key value stores that are used by the six YCSB runs.

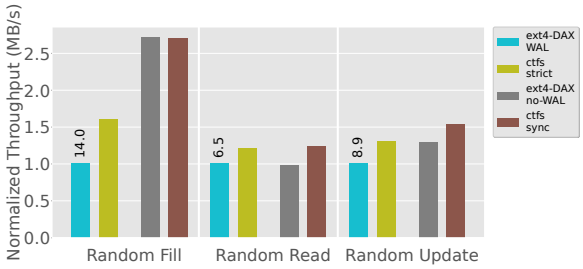

Figure 13: **RocksDB performance.**

	SplitFS		Ext4-DAX	ctFS
	sync	strict		
Bootstrap (µs)	1.4×10^6	1.1×10^6	0	5
open (create) (µs)	40	40	15	2
open (existing) (µs)	4	10	4	2
unlink (µs)	32	43	31	1.6
DRAM usage (MB)	198	572	N/A	0.52
Space available after format (GB)	230.7		230.7	248.1
Space used after YCSB LoadA (MB)	5486		5337	5378

Table 3: **Metadata operation and resource overhead.** There is no difference on between sync and strict modes for ctFS.

RocksDB. We ran our RocksDB experiments two configurations: *strict* and *relaxed*. With *strict*, where data consistency is guaranteed, ext4-DAX is run with WAL enabled, and ctFS is run in strict mode (ctFS-strict) but with WAL disabled. With *relaxed*, where crash consistency is not guaranteed, both ext4-DAX and ctFS-sync are run with WAL disabled.

With *strict*, ctFS-strict has 1.60x, 1.22x and 1.3x the throughput of ext4-DAX for the Random Fill test, the Random Read test and the Random Update test, respectively. This demonstrates the feasibility of replacing WALs in applications with atomic writes in ctFS, as doing so not only improves performance but also simplifies application logic.

With *relaxed*, ctFS-sync is on par with ext4-DAX with the Random Fill test, but has 1.25x and 1.19x the throughput for the Random Read and Random Update test.

4.3 Resource Usage & Other Operations

Table 3 shows the cost of several frequently used file system operations, as well as DRAM overhead after filesystem initialization and space efficiency for ctFS, SplitFS and Ext4-DAX. Notably, SplitFS spends over one second to initialize because it needs to build the U-Split mapping table, create and `mmap` all the staging files. Similarly, because of the mapping table, SplitFS uses 3 orders of magnitude more DRAM comparing with ctFS. The DRAM usage does not change significantly for SplitFS

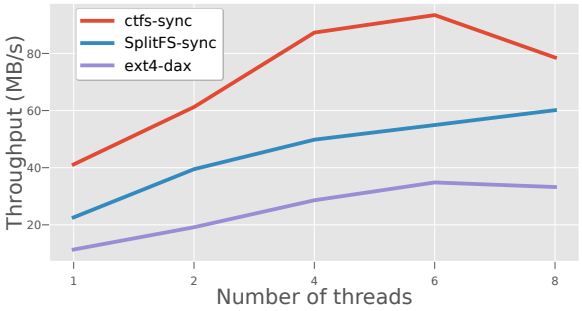

Figure 14: **Scalability of ctFS** versus ext4-DAX and SplitFS on LevelDB running YCSB Run A in terms of throughput.

and ctFS when running workloads as both of them primarily operate on PM.

In terms of space efficiency, ctFS has 7.52% more available space than ext4-DAX and SplitFS when newly formatted. In fact, ctFS only incurs 10MB memory overhead for newly formated 248.06GB space. This is because ctFS allocates inodes and inode bitmaps on demand. After running Load A in the YCSB test on LevelDB, ctFS occupies 0.78% more space than ext4-DAX and 2% less than SplitFS.

4.4 Scalability

The design of the cfFS's concurrency model is the same as that of ext4-DAX. Figure 14 shows ctFS's scalability compared with ext4-DAX, running YCSB Run A on LevelDB. All file systems scale similarly. Performance of ctFS peaks at 6 worker threads in a 8 core machine (as two additional threads are spawned by LevelDB for other purpose).

5 Limitations and Discussion

The design of ctFS presents two unique trade-offs. First, compared with an in-kernel file system, ctFS's user-space file system design trades protection for performance. While ctFS is not suitable as a general purpose file system, it presents a (rather extreme) trade-off point for data center applications to squeeze the most out of the hardware, as in data center environments each machine runs only a small number of applications that often trust each other, and protection against intentional attacks is ensured at the boundary of machines or data centers. Furthermore, ctFS can be used as an application's private file system, i.e., where one or several applications own one instance of ctFS.

Second, ctFS's design is also at the expense of internal fragmentation within each fixed-sized partition in the *virtual* memory address space. We argue this is acceptable

given the size of today's virtual address spaces. Both Intel and Linux now support 57 bits virtual addresses with 5-level paging, enabling a 128PB virtual address space. In comparison, the maximum size of Intel Optane DC that can be supported by a server today is 6TB [27]. Note that ctFS does not waste physical storage space as any unused regions of a partition are not mapped to physical memory. In addition, ctFS's allocation algorithm is similar to the buddy memory allocator, and hence, the internal fragmentation problem is fundamentally inline with that of modern size-segregated memory allocators like jemalloc [10], TCMalloc [13], and Go's runtime [16]; the wide adoption of these allocators further suggests that internal fragmentation is an acceptable trade-off.

6 Related Work

To the best of our knowledge, this paper is the first to propose a complete file system that supports contiguous files with a detailed design and evaluation.

SCMFS. SCMFS [39] proposed the high-level idea of allocating each file contiguously in the virtual address space. However, its design is only at a conceptual level. How files are allocated in the virtual memory space is not clearly described. Specifically, it does not address file resizing and external fragmentation, the two fundamental challenges faced by contiguous files. It is unclear what happens if one file expands into the range of another file. Finally, SCMFS's implementation and evaluation are entirely based on simulation.

File systems for PM. A number of file systems were designed to bypass the kernel. Aerie [37], PMFS [8], Strata [25], SplitFS [21], and ZoFS [7] all allow the user to directly access file data through a user-space component; PMFS, SplitFS, and ZoFS map the metadata and data in application's virtual memory space. In Aerie, metadata updates and locking requests must be sent via IPC to be processed by a trusted system service. Strata logs updates in userspace which are then digested in the kernel. ZoFS strives to provide security by only mapping the metadata to the users who have access permission, and only allows trusted library code to modify the metadata by exploiting MPK memory protection keys. KEVIN [24], a file system for NAND SSD instead of PM, provides an FPGA implementation of the log-structured merge tree, and ports file operations on top.

All of the file systems mentioned above still use a tree-structured index for file indexing. B*e*trFS proposes a B^{ε}-tree that is a write-optimized variant B-tree [20]. HashFS [30] uses a global fixed-sized hash table for indexing. However, it still suffers software indexing overhead, and its performance is no better when compared to SplitFS. KUCO [3] offloads some indexing from the kernel to the userspace through "collaborative indexing", to improve scalability. However, it still uses traditional ext2-style block mapping. In comparison, ctFS uses a contiguous file design that obsoletes file indexing.

Crash consistency on file data. Conventional write-ahead logging/journaling [15, 17, 38] typically requires writing the data *twice*: first to journal before updating the target file. The cost of double-write for data may be large, and several mechanisms that avoid data copying have been proposed [2, 4, 18, 26, 29]. Similar to `pswap`, SplitFS's relink is used to efficiently provide atomic writes without copying the data to the journal. `pswap` differs from `relink` in that the former swaps the virtual-to-physical memory mapping, whereas `relink` changes the mapping within ext4-DAX's extent trees. Failure atomic msync [32] atomically commits changes to a memory mapped file by using ext4's journalling function. SHARE [31] atomically lets pairs of pages share the same physical page in the flash storage. It does not explore the page table hierarchy for optimization.

SubZero [22] proposed a `patch()` function that atomically overwrites the destination region of a mmap file with the content of the source region. `pswap` is different in a few ways. First, `pswap` swaps the mapping whereas `patch` discards the content in the source region. In addition, `pswap` leverages the page table hierarchy to achieve significant speedup. Finally, `pswap` is mainly used for fast cross-partition expansion and shrink, whereas `patch` is only used for atomic writes.

7 Concluding Remarks

This paper proposes ctFS, a persistent memory file system which offloads file system indexing to the memory management hardware by keeping files contiguous in virtual memory. Our evaluation shows ctFS can outperform ext4-DAX and SplitFS by up to 7.7x and 3.1x, and improve YCSB throughputs by up to 3.6x and 1.8x.

Acknowledgements

We thank our shepherd Youngjin Kwon and the anonymous reviewers for their insightful comments. SplitFS authors provided valuable help in obtaining the breakdown of SplitFS's runtime. Rishikesh Devsot and Devin Gibson have ported an early version of ctFS to boot into a Linux shell without hitting Linux's file system. This research was supported by the Canada Research Chair fund, an NSERC discovery grant, and a VMware gift.

References

[1] Intel AVX-512 instructions. https://software.intel.com/content/www/us/en/develop/articles/intel-avx-512-instructions.html.

[2] J. Bonwick and B. Moore. ZFS: The last word in file systems. https://wiki.illumos.org/download/attachments/1146951/zfs_last.pdf.

[3] Y. Chen, Y. Lu, B. Zhu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and J. Shu. Scalable persistent memory file system with Kernel-Userspace collaboration. In *Proc. 19th Conf. on File and Storage Technologies*, pages 81–95. USENIX Association, Feb. 2021.

[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. 22nd Symp. on Operating Systems Principles*, SOSP'09, pages 133–146, 2009.

[5] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proc. 1st Symp. on Cloud Computing*, SoCC'10, pages 143–154, 2010.

[6] corbet. Address space randomization in 2.6. url-https://lwn.net/Articles/121845/, 2005.

[7] M. Dong, H. Bu, J. Yi, B. Dong, and H. Chen. Performance and protection in the ZoFS user-space NVM file system. In *Proc. 27th Symp. on Operating Systems Principles*, SOSP'19, pages 478–493, 2019.

[8] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System software for persistent memory. In *Proc. 9th European Conf. on Computer Systems*, EuroSys'14, New York, NY, USA, 2014. Association for Computing Machinery.

[9] J. Edge. Randomizing the kernel. url-https://lwn.net/Articles/546686/, 2013.

[10] J. Evans. A scalable concurrent malloc (3) implementation for freebsd. In *Proc. of the BSDCAN Conf., Ottawa, Canada*, 2006.

[11] A. S. Fedorova. Why mmap is faster than system calls. https://sasha-f.medium.com/why-mmap-is-faster-than-system-calls-24718e75ab37, 2019.

[12] G. R. Ganger, M. K. McKusick, C. A. N. Soules, and Y. N. Patt. Soft updates: A solution to the metadata update problem in file systems. *ACM Trans. Comput. Syst.*, 18(2):127–153, May 2000.

[13] S. Ghemawat and P. Menage. TCMalloc. http://goog-perftools.sourceforge.net/doc/.

[14] M. Gorman. Page table management. https://www.kernel.org/doc/gorman/html/understand/understand006.html.

[15] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1992.

[16] R. Griesemer, R. Pike, and K. Thompson. Golang. https://golang.org/.

[17] R. Hagmann. Reimplementing the Cedar file system using logging and group commit. In *Proc. 11th ACM Symp. on Operating Systems Principles*, SOSP'87, pages 155–162, 1987.

[18] D. Hitz, J. Lau, and M. Malcolm. File system design for an NFS file server appliance. In *Proc. of the USENIX Winter 1994 Technical Conference*. USENIX Association, 1994.

[19] 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. https://arxiv.org/abs/1903.05714v3, 2019.

[20] W. Jannen, J. Yuan, Y. Zhan, A. Akshintala, J. Esmet, Y. Jiao, A. Mittal, P. Pandey, P. Reddy, L. Walsh, M. A. Bender, M. Farach-Colton, R. Johnson, B. C. Kuszmaul, and D. E. Porter. BetrFS: Write-optimization in a kernel file system. *ACM Trans. Storage*, 11(4), Nov. 2015.

[21] 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 Symp. on Operating Systems Principles*, SOSP'19, pages 494–508, 2019.

[22] J. Kim, Y. J. Soh, J. Izraelevitz, J. Zhao, and S. Swanson. Subzero: Zero-copy io for persistent main memory file systems. In *Proc. 11th Asia-Pacific Workshop on Systems*, APSys'20, pages 1–8, New York, NY, USA, 2020. Association for Computing Machinery.

[23] K. C. Knowlton. A fast storage allocator. *Commun. ACM*, 8(10):623–624, October 1965.

[24] J. Koo, J. Im, J. Song, J. Park, E. Lee, B. S. Kim, and S. Lee. Modernizing file system through in-storage indexing. In *Proc. 15th Symp. on Operating Systems Design and Implementation*, OSDI'21, pages 75–92. USENIX Association, July 2021.

[25] Y. Kwon, H. Fingler, T. Hunt, S. Peter, E. Witchel, and T. Anderson. Strata: A cross media file system. In *Proc. 26th Symp. on Operating Systems Principles*, SOSP'17, pages 460–477, 2017.

[26] E. Lee, H. Bahn, and S. H. Noh. Unioning of the buffer cache and journaling layers with non-volatile memory. In *Proc. 11th Conf. on File and Storage Technologies*, FAST'13, pages 73–80, 2013.

[27] Lenovo. Intel Optane Persistent Memory 100 Series Product Guide. https://lenovopress.com/lp1066-intel-optane-persistent-memory-100-series#mixed-mode-requirements.

[28] LevelDB. https://github.com/google/leveldb.

[29] C. Mohan. Repeating history beyond ARIES. In *Proc. 25th Intl. Conf. on Very Large Data Bases*, VLDB'99, pages 1–17. Morgan Kaufmann Publishers Inc., 1999.

[30] 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 Conf. on File and Storage Technologies*, FAST'21, pages 97–111. USENIX Association, Feb. 2021.

[31] G. Oh, C. Seo, R. Mayuram, Y. Kee, and S. Lee. SHARE interface in flash storage for relational and NoSQL databases. In *Proc. 2016 Intl. Conf. on Management of Data*, SIGMOD'16, pages 343–354, 2016.

[32] S. Park, T. Kelly, and K. Shen. Failure-atomic msync(): A simple and efficient mechanism for preserving the integrity of durable data. In *Proc. 8th European Conf. on Computer Systems*, EuroSys'13, pages 225–238, 2013.

[33] S. Park, S. Lee, W. Xu, H. Moon, and T. Kim. Libmpk: Software abstraction for Intel memory protection keys (Intel MPK). In *Proc. 2019 Usenix Annual Technical Conf.*, USENIX-ATC'19, pages 241–254, 2019.

[34] D. M. Ritchie and K. Thompson. The UNIX time-sharing system. *Commun. ACM*, 17(7):365–375, July 1974.

[35] Direct I/O - RocksDB Wiki (accessed 2019-03-05). https://github.com/facebook/rocksdb/wiki/Direct-IO.

[36] A. Tanenbaum and H. T. Boschung. *Modern operating systems*. Pearson, 2018.

[37] H. Volos, S. Nalli, S. Panneerselvam, V. Varadarajan, P. Saxena, and M. M. Swift. Aerie: Flexible file-system interfaces to storage-class memory. In *Proc. 9th European Conf. on Computer Systems*, EuroSys'14, pages 14:1–14:14, 2014.

[38] D. Woodhouse. JFFS : The journalling flash file system. In *Proc. Ottawa Linux Symposium*. RedHat Inc., 2001.

[39] X. Wu and A. L. Narasimha Reddy. SCMFS: A file system for storage class memory. In *Proc. 2011 Intl. Conf. for High Performance Computing, Networking, Storage and Analysis*, SC'11, pages 1–11, 2011.

[40] J. Xu and S. Swanson. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *Proc. 14th Conf. on File and Storage Technologies*, FAST'16, pages 323–338, Santa Clara, CA, 2016.