ctFS: Converting File Index Traversals to Hardware Memory Translation through Contiguous File Allocation for Persistent Memory

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

ctFS is a new file system that aims to exploit the characteristics of byte-addressable persistent memory (PM) so as to reduce file access overheads. It achieves these gains by representing each file as a contiguous region of virtual memory and leveraging memory management abstractions and hardware to efficiently navigate the file. In particular, translating an offset to a file address becomes a simple arithmetic offset operation followed by a mapping of the target virtual address to a physical address which can be performed efficiently by the hardware MMU. This translation incurs a fraction of the overhead of traditional (e.g., extent-tree) index lookups in software.

Byte-addressable persistent memory (PM) fundamentally blurs the boundary between memory and persistent storage. For example, 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) [7]. Many new file systems for PM have emerged in recent years. 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 and directly accesses PM using memory operations (`memcpy()`). Other designs bypass the kernel by mapping different file system data structures into user space to reduce the overhead of switching into the kernel [3,4,8,10,15]. SplitFS, a state-of-the-art PM file system, aggressively uses memory-mapped I/O [8] for significantly improved performance.

With PM closing the performance gap between DRAM and persistent storage, file access bottlenecks have shifted from I/O to file indexing overhead. As shown in Figure 1, the file indexing overhead can be as high as 45% of the total runtime on ext4-DAX for write workloads with many append operations. Yet all existing file systems still rely on traditional tree-based file indexing, first proposed by Unix [13] in the 70s, when the speed of memory and disk differed by several orders of magnitude. While memory-mapped I/O (`mmap()`) can reduce indexing overheads [6], it does not remove them, but only shifts their timing to page fault handling or `mmap()` operations (when pre-fault is used). Even with SplitFS, file indexing contributes to 62% of its runtime in append-heavy workloads. ctFS, on the other hand, nearly eliminates file indexing overhead, achieving 7.7x speedup against ext4-DAX and 3.1x against SplitFS on the append workload (Figure 1).

An alternative to using index-structured files is to use contiguous file allocation. While simple contiguous allocation designs like fixed-size or variable-size partitions are known [14], they face three major design challenges: (1) internal fragmentation for fixed-size partitions, (2) external fragmentation for variable-size partitions, and (3) file resizing. The only use of contiguous file allocation in practice is in CD-ROM, where files are read-only [14].

ctFS addresses the challenges associated with contiguous file allocation and exploits hardware for memory translation. Its source code will be available at `https://github.com/robinlee09201/ctFS`. ctFS was designed from the ground up with the following design elements:

- Each file (and directory) is contiguously allocated in the 64-bit *virtual* memory space. Hence we can leverage existing MMU hardware to efficiently perform virtual-to-physical translation, even as the file needs not be contiguous in the physical address space. 64-bit virtual address spaces are so large that contiguous allocation is practical for any existing system. Furthermore, the virtual address space is carefully managed by using a hierarchical layout of memory paritions, similar to that of the buddy memory allocator [9], in which each partition is subdivided into 8 equal-size sub-partitions. This design speeds up allocation, avoids external fragmentation, and minimizes internal fragmentation.

- A file's virtual-to-physical mapping is managed using *persistent page tables* (PPTs). PPTs have a structure similar to that of regular, volatile page tables for virtual addresses in DRAM, but they are stored persistently on PM. Upon a page fault on an address that is within ctFS's region, the OS looks up the PPTs to create the mappings in the DRAM-based page tables.
- 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()`, a new OS system call we proposed that *atomically* swaps the virtual-to-physical mappings.

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 its operations.

A limitation of ctFS is that we implement it as a user-space, library file system that trades protection for performance. While this maximizes performance by aggressively bypassing the kernel, it sacrifices protection in that it only protects against unintentional bugs instead of intentional attacks. At the same time, we see no reason why ctFS could not be implemented in the OS kernel.

Analysis of File Indexing Overhead

We analyzed the performance overhead of block address translation in Linux's ext4-DAX, the port of the ext4 extent-based file system to PM and in SplitFS [8] using the six microbenchmarks listed in Table 1. The experiments were performed on a Linux server with 256GB Intel Optane DC persistent memory.

Figure 1 shows the breakdown of the completion time of each benchmark. For ext4-DAX, we observe that indexing overhead is significant in Append and Sequential Write Empty (SWE), spending at least 45% of the total runtime on indexing. In both cases, the index time includes the time to build the index. For the random access workloads, RR and RW, the proportion of time spent on indexing is lower, but still considerable: at 18% and 15% respectively.

Table 1: Six microbenchmarks for evaluating file indexing overheads. Each operates on 10GB files. RR, RW perform the reads/writes 2,621,440 times.

Append	Append 10 GB of data, 4KB at a time to an initially empty file
SWE	Sequentially write 10 GB of data, 1GB at a time, to an initially empty file
RR & RW	Read/write 4KB at a time from/to a random (4KB-aligned) offset of a previously allocated file
SR & SW	Sequentially read/write 10 GB of data, 1GB at a time, from a previously populated file

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%), even though its total runtime is shorter. This is because SplitFS's speedup further shifts the bottleneck and exacerbates the indexing overhead. SplitFS splits the file system logic into a user-space library (U-Split) and a kernel space component (K-Split), where K-Split reuses 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.

To understand its indexing overhead in more detail, consider the Append workload. SplitFS spends a total of 6.62s on indexing overhead. The majority (4.37s) comes from the kernel indexing time during page fault handling. SplitFS converts all read and write operation to memory mapped I/O; hence an operation could trigger a page fault, which in turn triggers kernel indexing. The time spent in `mmap()` itself is smaller (1.39s). The remaining 0.84s comes from the indexing time in its user-space component, U-split, spent on mapping a file offset to the corresponding ext4-DAX file.

In comparison, ctFS successfully eliminates most of the indexing time: in all six benchmarks, the vast majority (at least 97%) of the runtime is spent on I/O, instead of indexing. As a result, it achieves a 7.7x speedup against ext4-DAX and 3.1x against SplitFS on the Append benchmark, whereas its average speedups on the other benchmarks are 2.17x and 1.97x over ext4-DAX and SplitFS, respectively.

Note that both SplitFS and ctFS have two modes, sync and strict. The results in Figure 1 are from their sync

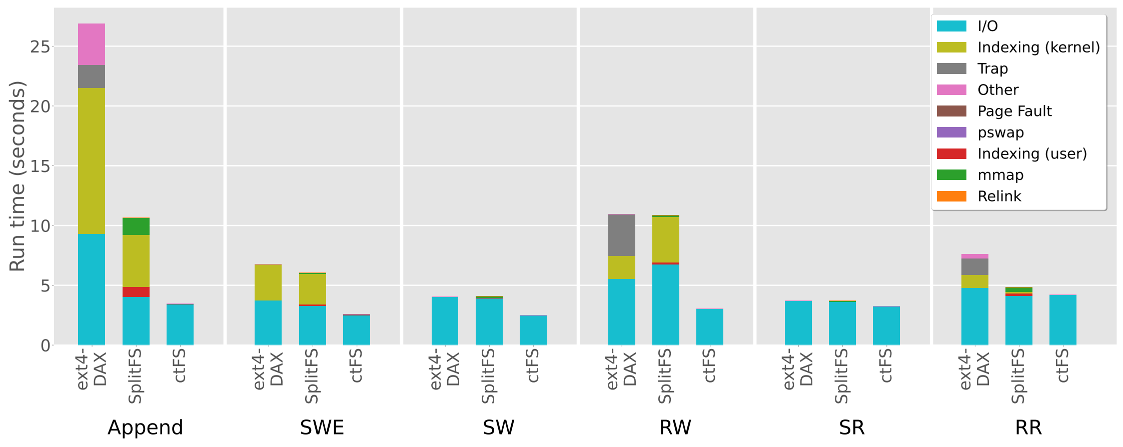

Figure 1: Overhead breakdown for ext4-DAX, SplitFS, and ctFS using persistent memory.

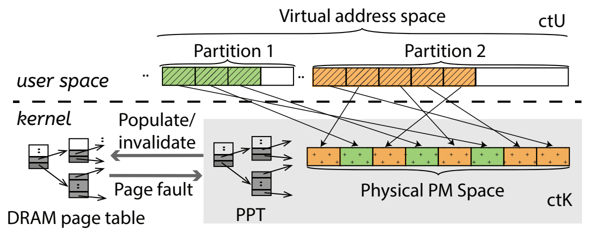

Figure 2: Architecture of ctFS. Each box represents a page. Two partitions are shown. The file allocated in partition 1 uses 3 pages (green), and the file in partition 2 uses 5 pages. ctK maintains virtual-to-physical page mappings in the persistent page table (PPT).

L9 512GB	L8 64GB	L7 8GB	L6 1GB	L5 128MB	L4 16MB	L3 2MB	L2 256KB	L1 32KB	L0 4KB
PGD	PUD			PMD			PTE (sub-PMD)		

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.

mode, because it offers comparable crash consistency guarantees to ext4-DAX. The results on strict mode, which offers stronger crash consistency, as well as comparison with other research file systems like NOVA [16] and pmfs [4], can be found in our FAST'22 paper [12].

Overview of ctFS

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 translations to the memory management subsystem. In addition, ctFS provides an efficient atomic primitive called `pswap` to ensure data consistency while minimizing double-writing. Write operations on ctFS are always synchronous, i.e., writes are persisted on PM before the operation completes; in fact, similar to Linux's DAX PM file systems, all writes are directly applied on PM without being cached in DRAM.

ctFS's architecture, shown in Figure 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 manages 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 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, then copy the mapping from the PPT to the kernel's regular DRAM page table, allowing virtual to PM address translation to be carried out by the MMU hardware. When any mapping in the PPT becomes obsolete, ctK will remove the corresponding mapping from the DRAM kernel 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.

File System Structure (ctU)

ctFS's user-space library, *ctU*, organizes the file system's *virtual* memory space into hierarchical partitions to facil-

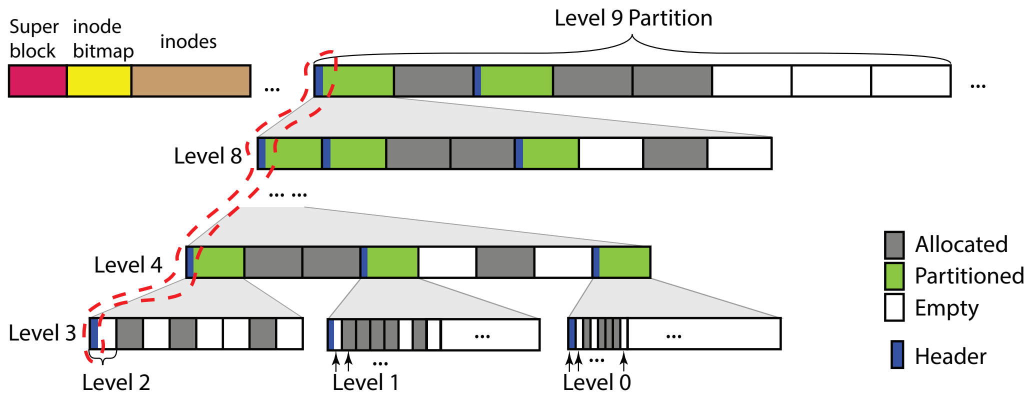

Figure 4: Layout of ctFS in *virtual* address space. The space of an entire partition is reserved in the virtual memory space, whereas the physical PM space is allocated on-demand based on the actual usage. Headers circled in the dashed-line reside on the same page.

itate contiguous allocations. The size of each partition at a particular level is identical; and the size of a partition at a particular level 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 (4TB), L11 (32TB), *etc*.

A file or directory is always allocated contiguously in one and only one partition, and such that the partition is of the smallest size 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 later. Therefore, our only options for partition size differences are: 2x (2^1), 8x (2^3), or 512x (2^9). We chose 8x because 2x would be too small and 512x too large.

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 i-nodes themselves. Each inode stores the file's metadata (e.g., owner, group, protection, size, etc.), and it contains *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. Note that the 512GB allocated for an L9 partition is in virtual memory; The physical pages underneath it are allocated on demand.

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.

ctU needs to maintain book keeping information for each partition, such as its state, as well as information that facilitates fast allocations. 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 in the first page of the partition for fast lookup that does not involve indirections.

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 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 partition within an L9 partition 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 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 allow partitions in levels L0–L3 to be further partitioned, 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 partitions, 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.

Kernel Subsystem Structure (ctK)

ctK manages the PPT and implements `pswap()`. The structure of the PPT is identical to Linux's 4-level page table with two key differences: (1) It resides on PM and is thus persistent; (2) It uses relative addresses for both virtual and physical addresses, because ctFS's memory region may be mapped to different starting virtual addresses in different processes due to *Address Space Layout Randomization* [2] [5], and hardware reconfiguration could change starting physical address. 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

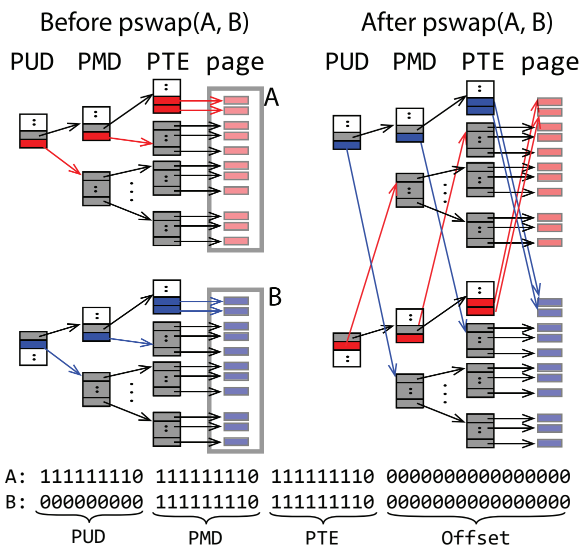
Before pswap(A, B)
After pswap(A, B)
PUD PMD PTE page
PUD PMD PTE page
A
B
A: 111111110 111111110 111111110 0000000000000000
B: 000000000 111111110 111111110 0000000000000000
PUD
PMD
PTE
Offset

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.

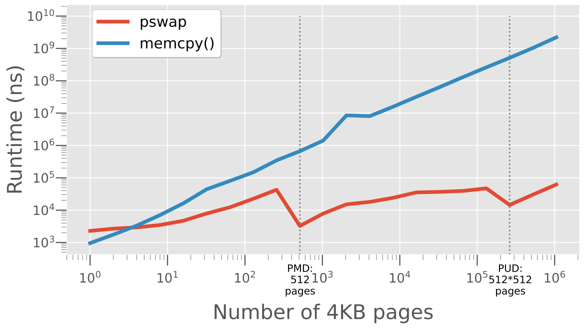

Figure 6: Comparing the performance of `pswap` and `memcpy`. Both the X and Y axis are log scale.

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.

The `pswap()` system call *guarantees crash consistency*: 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.

To optimize performance, `pswap()` avoids swapping every target entry in the PTEs (the last level page table) of the PPT whenever possible. Figure 5 shows an example

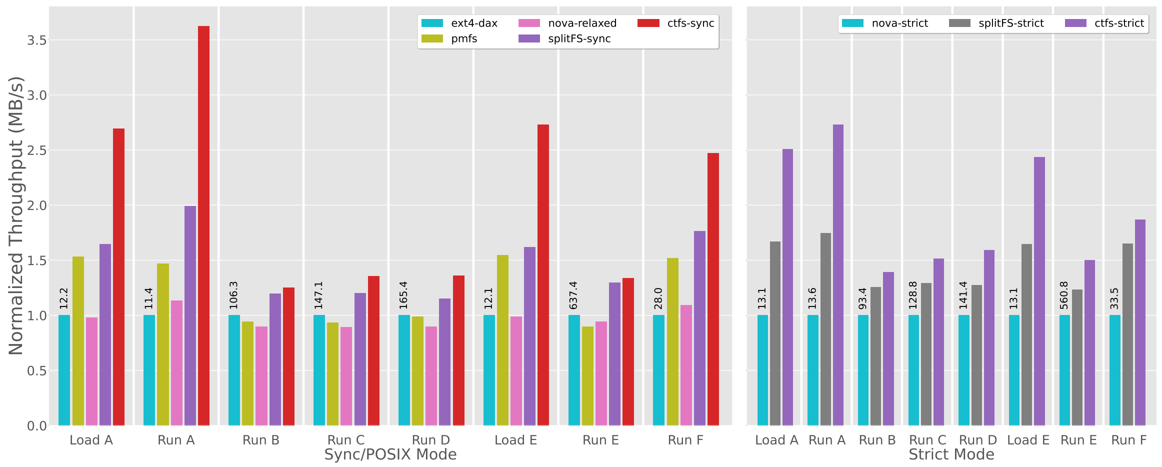

Figure 7: YCSB on LevelDB. Results are measured in throughput that is normalized to ext4-DAX in the sync/POSIX group and NOVA-strict in the strict group. The number on top shows the absolute throughput.

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 in the PPT (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).

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 curve of `pswap` performance shows a wave-like pattern: as the number of pages increases, the 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` only takes 62μs. However, when N is less than 4, `memcpy` is more efficient than `pswap`.

There are two use cases of `pswap`. First, when a write (append) triggers an upgrade to a larger partition or a truncate triggers a downgrade, instead of copying the file data from the old partition to the new partition, ctU uses `pswap` to atomically change the mapping. Second, `pswap` can be used to support atomic write on any amount of data. To do so, ctU first writes the data to a staging partition, and when the write is complete, it swaps the newly written data to the memory region belong to the target file.

Performance on Real-world Application

We evaluated ctFS on LevelDB [11], using the YCSB [1] benchmark. (Microbenchmark results were shown in Figure 1.) YCSB includes six different key-value access workloads, including update heavy (A), read mostly (B), read only (C), read records that were recently inserted (D), range query (E), and read-modify-write (F), as well as two load workloads (A and E).

Figure 7 shows the performance of different PM file systems on LevelDB using YCSB workload. ctFS outperforms all other file systems of comparable consistency levels in every workload. ctFS achieves the most significant speedup in write-heavy workloads, Load A and E and Run A, B, F. Among these write-heavy workloads, ctFS-sync's throughput is 1.64x of the throughput of SplitFS-sync on average, with 1.82x in the best-case (in Load E). Compared with ext4-DAX, ctFS-sync's throughput is 2.88x on average with 3.62x in the best case (in Run A). In strict mode, ctFS's throughput is 1.30x of SplitFS on average, with 1.50x in the best-case (in Load A). In comparison, ctFS's speedups on read-heavy workloads are smaller. But it still achieves an average of 1.25x - 1.36x speedup over ext4-DAX.

References

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

[2] Corbet. Address space randomization in 2.6. https://lwn.net/Articles/121845/, 2005.

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

[4] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System software for persistent memory. In *Proceedings of the Ninth ACM European Conference on Computer Systems*, EuroSys '14, 2014.

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

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

[7] 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.

[8] 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 *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 494–508, 2019.

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

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

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

[12] R. Li, X. Ren, X. Zhao, S. He, M. Stumm, and D. Yuan. ctFS: Replacing file indexing with hardware memory translation through contiguous file allocation for persistent memory. In *Proceedings of the 20th Usenix Conference on File and Storage Technologies*, FAST'22, 2022.

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

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

[15] H. Volos, S. Nalli, S. Panneerselvam, V. Varadarajan, P. Saxena, and M. M. Swift. Aerie: Flexible file-system interfaces to storage-class memory. In *Proceedings of the Ninth ACM European Conference on Computer Systems*, EuroSys '14, pages 14:1–14:14, 2014.

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