ZonFS: A Storage Class Memory File System with Memory Zone Partitioning on Linux

Jang Woong Kim, Jae-Hoon Kim, Awais Khan, Youngjae Kim, Sungyong Park
Department of Computer Science and Engineering
Sogang University, Seoul, Republic of Korea
{ski6812, jhkim406, awais, youkim, parksy}@sogang.ac.kr

***Abstract*—Recent developments in storage class memory such as PCM, MRAM, RRAM, and STT-RAM have strengthened their leadership as storage media for memory-based file systems. Traditional Linux memory-based file systems such as *Ramfs* and *Tmpfs* utilize the Linux page cache as a file system. These file systems, when adopted for SCM, have the following problems. First, current implementations of Ramfs and Tmpfs have no mechanism to explicitly allocate pages from specific memory. Second, memory pages allocated from SCM do not follow the Linux kernel's page allocation process exactly, resulting in unnecessary performance overhead. To resolve the aforementioned challenges, we propose the development of a new memory-based file system using Memory Zone Partitioning for SCM called *ZonFS*. *ZonFS* is implemented by extending the Linux Ramfs. In particular, we define a storage zone for SCM, modify the Ramfs to allocate a file system page from SCM. *ZonFS* avoids running unnecessary Linux VM kernel codes such as (i) inserting pages allocated from SCM into the LRU list for VM page replacement and (ii) checking dirty pages for write-back to disk. We also modify the Ramfs to allocate inode cache in SCM and eliminate the risk of inode cache loss in case of power failure. Extensive evaluations indicate that *ZonFS* has up to 9.1% and 14.1% higher I/O throughputs than native Ramfs and Tmpfs.**

I. Introduction

Emergences of non-volatile memory such as STT-RAM [1], PRAM (Phase change RAM) [2], RRAM (Resistive RAM) [3], Intel and Micron 3D X-point [4] gave us the opportunity to use memory as a storage, i.e., Storage Class Memory (SCM). These memories are expected to be directly attached to a processor along with DRAM. These memories fundamentally differ from traditional block devices such as hard disk drives (HDD) and solid-state drives (SSD), which should be accessed through the I/O block layer in OS. On the other hand, SCM can be accessed through memory load and store instructions by CPU. A hybrid memory system combining DRAM and SCM, as shown in Figure 1, was proposed for its high energy efficiency compared to a system with only DRAM [5], [6], [7]. In such a hybrid configuration, DRAM and SCM are both connected to a memory bus and are directly accessed by the CPU.

Memory file systems such as Linux Ramfs and Tmpfs allow us to build memory file systems with host DRAM. These file systems are basically implemented using Linux memory management techniques. When a file is created, memory pages are allocated by the OS. When the file is read, its corresponding pages are referred. This makes it easy to read and write file system data in memory. However, Tmpfs and Ramfs have not been implemented considering such hybrid memory design. It allocates the pages without considering memory type. Hence, it is not possible to know in which area of the memory the page is allocated. Moreover, when system power failure occurs, data in SCM (non-volatile) is preserved, however, the corresponding file control blocks (inode cache) will be lost if they are stored in DRAM (volatile memory). In addition, when the system restarts, file system data of the SCM area may be overwritten. Therefore, current Linux memory file systems can not be directly used as a memory file system for SCM in such a hybrid memory environment.

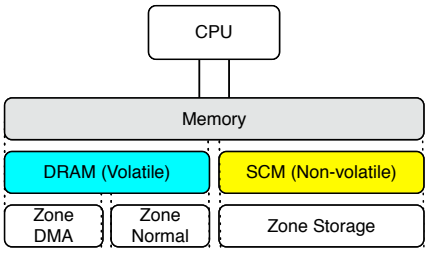


Fig. 1: Illustration of Memory Zone Partitioning for Hybrid Memory combining DRAM and SCM.

In this paper, we propose *ZonFS*, a memory file system for SCM in a hybrid memory using *Linux Memory Zone Partitioning*. Memory pages are allocated according to the memory type and zone. In Linux, memory is divided into three zones: DMA Zone, Normal Zone and HighMem Zone [8]. In particular, the 64-bit kernel does not use the HighMem Zone. To isolate SCM from DRAM, we define Storage Zone for the SCM and specify the entire SCM address space as the Storage Zone. Figure 1 illustrates memory zone partitioning for the hybrid memory with DRAM and SCM. When file systems are built upon SCM, memory pages of file system will be managed in SCM separately from normal pages allocated from DRAM.

In this paper, the following contributions are made:

- *Memory Zone Partitioning:* We isolate memory pages of SCM from DRAM memory pages using Memory Zone Partitioning. In Linux, physical memory addresses are allocated sequentially in the memory slot. When the SCM is plugged into the memory slot, the physical memory address can be found in the BIOS. In this paper, we use part of DRAM as SCM. We specify the start and end addresses of the SCM in the Linux kernel code. Hence, the pages in SCM will be

used only when file systems are built for SCM.

- *Avoiding Unnecessary Kernel Code:* Ramfs and Tmpfs use page cache to perform file I/O. Since the file operations such as read() and write() use Linux kernel's generic file I/O functions such as $generic\_perform\_write()$, unnecessary overheads are entailed, if pages are allocated from SCM. These include (i) checking dirty pages and (ii) inserting pages into LRU list. When dirty pages are more than certain threshold of pages, kernel's generic file I/O operations will write them back to disk and make dirty pages under certain number of pages throughout the system. In addition, page cache is managed as an LRU list. In practice, pages in SCM do not have to be replaced, but in current Linux kernel, all pages in SCM are added to the LRU list for page replacement, resulting in unnecessary search operations. Thus, we modify the Linux kernel code to bypass the above-mentioned unnecessary operations on the pages allocated from the SCM.
- *Linux Kernel Development and Evaluation:* We developed a memory file system for SCM by extending Linux Ramfs and modifying the Linux kernel memory management code. To demonstrate the efficacy of *ZonFS* with Memory Zone Partitioning, we compare *ZonFS* with native Ramfs and Tmpfs for I/O throughputs using IOZone benchmark [9]. As a representative example, for write operations, *ZonFS* showed up to 9% and 13% performance improvements over native Ramfs and Tmpfs.

The rest of this paper is organized as follows. Section II describes describes the memory access method and its implementation for *ZonFS*. Section III shows performance comparison results of *ZonFS* with Linux memory file systems, Tmpfs and Ramfs. Section IV introduces other memory-based file systems and their problems when implemented for SCM file systems. We conclude our work in Section V.

II. Design and Implementation

In this section, we describe the design principles for the SCM file system implementation using the Linux page-cache. We implement *ZonFS* with the following design goals – (i) manages the pages in storage zone separately from those in DRAM memory zone and (ii) optimizes the current Linux kernel code for SCM file systems. We achieve these goals by adding a new memory zone in the Linux kernel and thus allowing the kernel to distinguish pages of DRAM and SCM.

Before explaining the details of *ZonFS* implementation, we describe the memory area management in the Linux kernel and track the Linux kernel function calls to write and read in Ramfs to understand the code-level I/O behavior of the page cache based file system. We then describe the kernel modifications made to optimize the file system for SCM.

A. Linux Memory Zone Management

Linux kernel manages memory region by dividing it into three zones: $ZONE\_DMA$, $ZONE\_NORMAL$ and $ZONE\_HIGHMEM$. The DMA zone is a memory area for hardware that requires a specific memory range, and it uses 0 to 16 MB of memory area.

TABLE I: E820 Memory Map with Storage Zone.

E820 Type	**Usage**
E820_RAM	System RAM
E820_RESERVED	Reserved Memory
E820_ACPI	ACPI Tables
E820_NVS	ACPI Non-volatile Storage
E820_UNUSABLE	Unusable Memory
E820_PMEM	Persistent Memory
E820_PRAM	Persistent Memory (legacy)
E820_RESERVED_KERN	System RAM (reserved)
E820_STORAGE	SCM Storage

The Normal zone is used for general memory allocations. The purpose of the Highmem zone is to free the 4 GB virtual memory space limitations of the 32-bit instruction set architecture system. Hence, this area is not used in 64-bit systems. During the booting phase, Linux splits memory space into zones and divides each zone into multiple pages. Each of the zone is managed by a kernel structure $struct\_zone$, which also maintains a list of pages in its zone. A page requested by the kernel is handled by allocating a new page or simply returning an existing page. In a hybrid memory system, the Linux kernel assigns physical address spaces to all connected memories. In *ZonFS*, we distinguish space of DRAM and SCM by adding Storage Zone for SCM using Memory Zone Partitioning.

BIOS can get the physical memory address using the driver. When SCM is attached, the BIOS can retrieve the physical start and end addresses of the SCM through the SCM driver. This information includes the usage of each memory range. In the x86 architecture, E820 memory map contains the information, as shown in Table I. Especially, we have added a new E820 entry called $E820\_STORAGE$ for SCM storage usage of *ZonFS*. $E820\_STORAGE$ entry is used only for file allocation in *ZonFS* whereas $E820\_RAM$ and $E820\_RESERVED\_KERN$ entries are used for system memory. Note that in our implementation for *ZonFS*, we simulate SCM by using part of DRAM. Hence, we have manually set the memory range of $ZONE\_STORAGE$ in the kernel code, without the aid of the driver.

Linux kernel initializes the variables, max_pfn and max_low_pfn based on E820 memory map. The values of max_pfn and max_low_pfn indicate the maximum and minimum page frame numbers that can be used for system memory. Then it divides the memory zone using these variables. Figure 2 shows memory zone structure with Storage Zone for SCM. We have created a new memory zone, $ZONE\_STORAGE$ that lies on the whole range of $E820\_STORAGE$. It prevents the storage zone from being used as system memory.

B. I/O Flows for Write and Read Requests

In Linux, file I/O data goes through page cache. Accessing the disk for every file request is inefficient. OS stores the data

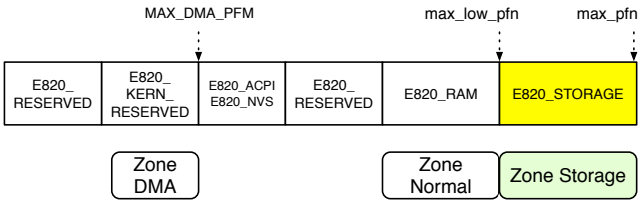


Fig. 2: Linux Memory Zone Partitioning with Storage Zone.

in memory's page cache at the first access of the data. By this way, we can access the same file data from memory for later requests. If page cache pages need to be used for other purpose when the physical memory is low, kernel write-backs the pages into backing store in case of the dirty page. Since Ramfs does not have backing store, all of the files of Ramfs are stored only in the page cache. Therefore, page cache pages of Ramfs are never write-backed.

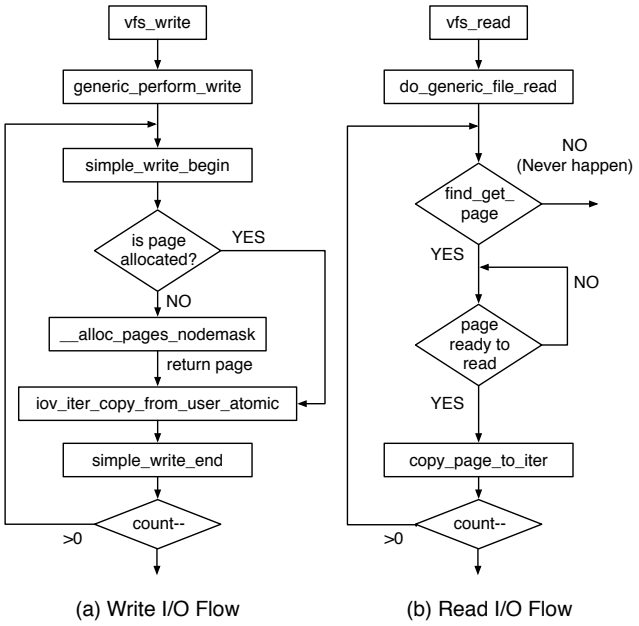
ZonFS is based on Ramfs, thus we investigate the I/O path of Linux Ramfs in detail. Figure 3 describes Ramfs file I/O paths in kernel function level starting from virtual file system operations. Write requests are initially handled by the virtual file system, which then calls the file system's own write function. Since Ramfs takes advantage of kernel's generic I/O functions, $generic\_perform\_write()$ is called. Then it follows $simple\_write\_begin()$, where the file is locked and the page searching occurs.

For write operations, there are two scenarios: *initial* write and *normal* write (update). Initial writes happen when pages for required file offset are not previously allocated, and normal writes happen when already allocated. In case of an initial write, $pagecache\_get\_page()$ is called to check whether the desired page exists. If the page was not allocated, $\_\_page\_alloc\_node\_mask()$ determines proper memory zone and allocates the page. Without any specified zone option, $ZONE\_NORMAL$ is the default allocation area. However, *ZonFS* can assign pages to $ZONE\_STORAGE$. For updates, since desired pages were already allocated, page allocation is not needed. It acquired wanted page, $iov\_iter\_copy\_from\_user\_atomic()$ performs actual write and $simple\_write\_end()$ releases the lock. These steps are repeated until all the requested data is written.

Read operations follow a similar process like the write operations. But, it never allocates pages because the required pages always exist. This is obvious because absence of pages means that they need to be copied from backing store, which is not the case of memory-based file systems. Even after acquiring required pages, actual read is delayed until the pages are ready to use, since other processes may be writing on the pages. Data is read in $copy\_page\_to\_iter()$. The details of kernel function calls for complete read and write I/Os are shown in Figure 3(a)-(b).

C. Linux Kernel Code Modification for ZonFS

ZonFS uses Linux page cache with pages in Storage Zone to store file data. In current Linux kernel, all the page cache pages



(a) Write I/O Flow (b) Read I/O Flow

Fig. 3: Write and Read I/O Flow of Ramfs.

are the potential victims of virtual memory management: 1) they can be swapped out by kernel when the physical memory is low, and 2) the kernel periodically checks the dirty pages and write backs to the backing store to save VM resources for later use. These tasks are not required in *ZonFS*. They can be bypassed, when pages are allocated from SCM for storage. Ramfs and Tmpfs set $PG\_UNEVICTABLE$ flag for pages in page cache not to swap them out, but they still remain to be the targets of VM management, which is an overhead in *ZonFS*. Therefore, in *ZonFS*, we make pages in Storage Zone, free from VM management.

Unnecessary operations on the above-mentioned memory access paths include the followings: First, when they allocate pages, although pages of the page cache's flag $PG\_UNEVICTABLE$ is set to prevent them from being swapped-out, they needlessly add the pages into LRU list that contains page replacement candidates. This causes LRU list maintaining overhead. Second is dirty page check overhead. Write processes periodically counts the number of dirty pages in the page cache. If the number exceeds the threshold, it write-backs all the dirty pages or in worse case, throttles the process for some time.

Figure 4(a) describes tasks. When a write request arrives, if it is the first write, a new page is allocated from page cache. It then sets the page with PG_Dirty flag and adds the page to the LRU list for later VM management. Then, it checks if write-back operations for dirty pages need to be performed. Definitely, these tasks are essential for virtual memory management that critically affects the entire system performance. However, pages allocated from SCM for storage are not subject to management. Therefore, we follow different memory access paths for DRAM and SCM requests in VM

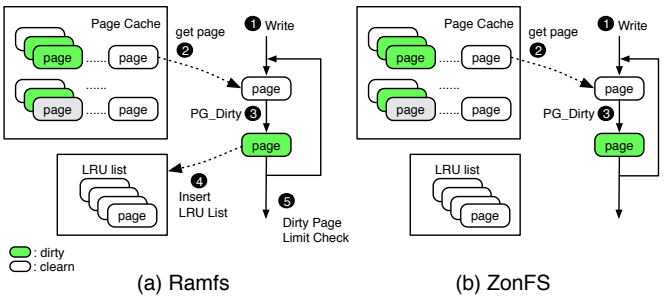


Fig. 4: Illustration of the relationship between Write operation and Page Cache. (a) and (b) show the operation procedures for Ramfs and *ZonFS* respectively.

management. DRAM accesses are triggered by application memory requests, and SCM accesses by requests for files. For DRAM accesses, we fully exploit Linux virtual memory layer. However, for SCM accesses, we provide a simple access path which uses page cache to store files but prevents them from being virtual memory management targets.

To differentiate memory access path for DRAM and SCM, we must be able to distinguish file requests from process memory requests. As mentioned earlier, we have achieved this by adding a new memory zone in Linux called $ZONE\_STORAGE$, which is allocated to SCM for storage usage. But since we implemented our design by only using DRAM, we have allocated new storage zone in a certain range of DRAM. All the flags $\_\_GFP\_STORAGE$ corresponding to the file inodes in *ZonFS*, to let the kernel know this is a file and store its data into our new zone. This zone partitioning method enables us to bypass LRU list insertion for the pages of $ZONE\_STORAGE$. Previous LRU list insertion was done after page cache allocation, thus we do not add the pages to the list in the case of file pages in $ZONE\_STORAGE$. We also skip dirty page check for the file page writes. We have restricted write operation to execute dirty check for process memory requests only, thus preventing write-backs and I/O process throttling. The new management of page cache for *ZonFS* is described in Figure 4(b).

Note that storing only the file contents into our new zone is not sufficient for consistency of the file system, but also inodes must be kept in that zone. Failure of this will will result in inaccessible isolated data. In Linux kernel, inode structure is allocated by slab allocator, which we use when allocating frequently allocated structures such as inode, per-thread structure, etc. It reduces the allocation and de-allocation overhead. Slab allocator pre-allocates caches dedicated to certain structure, and actual structure is allocated from those pre-allocated caches. A structure cannot be de-allocated, but handed over to cache so that future inode allocations can occur from it. Since slab allocator allocates caches from normal zone, an unexpected power failure can cause loss of inodes, which makes it impossible to reach the corresponding files. Therefore, for *ZonFS*, we modify Ramfs such that the inode cache is allocated in $ZONE\_STORAGE$ at the mount time of the file system. Since actual inode structures are allocated from the inode cache, we can guarantee all the inode structures are persistent.

III. Evaluation

For the evaluation of *ZonFS*, we compare *ZonFS* with two linux memory file systems, Ramfs and Tmpfs. We show the results of *ZonFS* with varying record size, and then we explore the performance of *ZonFS* by increasing the number of I/O threads for scaling performance.

A. Experimental Setup

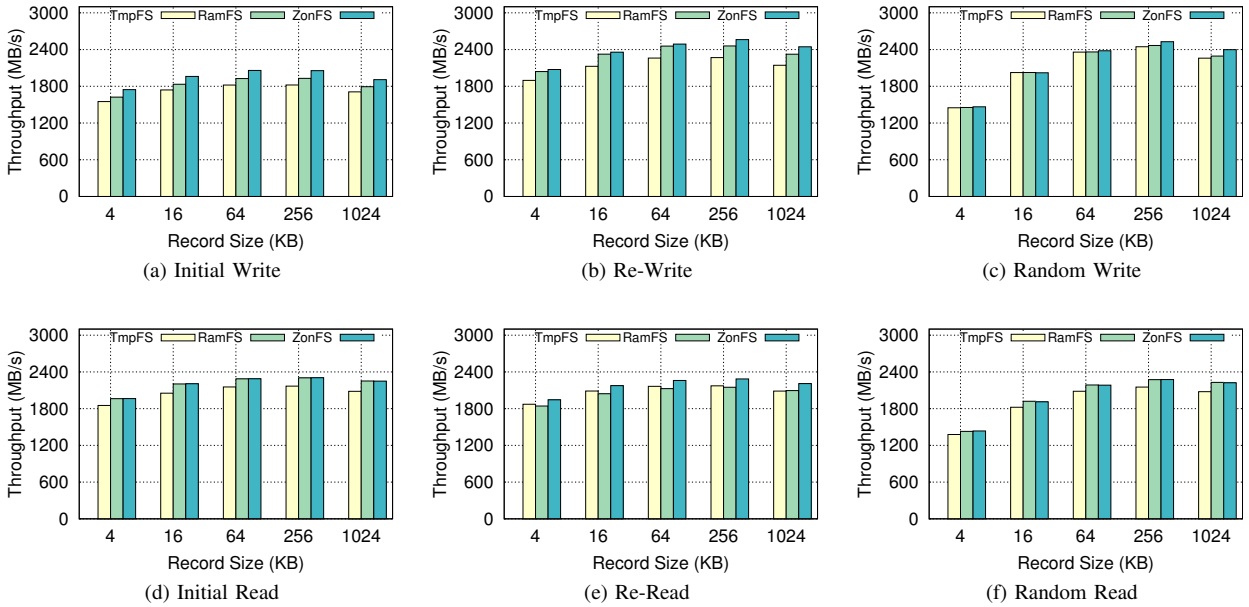
ZonFS has been developed by modifying the Linux kernel version 4.7.4 source code. The total number of modified kernel code lines is approximately 50. We use Intel server comprising of 8 cores with two 4-Core Intel Xeon Processor E5410 CPUs. The server is equipped with total of 16 GB DRAM, where 10 GB of DRAM area is assigned as storage zone to simulate non-volatile memory.

We have used IOzone [9] benchmarking tool to generate workload datasets. All the experiments were conducted for basic file operations such as *Write, Re-Write, Random Write, Read, Re-Read,* and *Random Read*. We have evaluated the performance of *ZonFS* and native Ramfs and Tmpfs by changing record size and the number of threads. The record size of the files varied from 4 KB to 1 MB, and the number of threads used in the experiment ranges from 1 to 40.

B. Results

1) Impact of Record Size: Figure 5 compares *ZonFS* with native Tmpfs and Ramfs for different file operations by varying record size. In this experiment, we measured the performance of single I/O thread file operations on a single 10 GB file.

Figure 5(a)-(c) show results for write workloads. Figure 5(a) shows the performance comparison for initial write. We observe that *ZonFS* shows the maximum performance improvement of 7.5% compared to Ramfs, and a maximum improvement of 11.4% over Tmpfs for 4K record size. We have similar performance improvement for bigger record size. Figure 5(b) shows results for re-write. *ZonFS* shows improvement, but overall performance improvement of initial write is much bigger than that of re-write. This is because, for every Ramfs page allocation, always the page has to be pushed to LRU list whereas in *ZonFS* it is not. Since we eliminated that overhead, initial write shows much better improvement than re-write. In Tmpfs initial write case, there is some degree of performance degradation because it requires an additional step to check whether the allocated page has exceeded the file system capacity. For re-write, for every process that writes or re-writes to a Ramfs file, it validates whether it exceeded the dirty page limit of the system. Since *ZonFS* bypasses dirty check, it slightly improves re-write performance compared to Ramfs. For all file systems, re-write shows higher throughput than initial write. We suspect that this can be attributed to CPU cache effect: although *ZonFS* does not use page cache (it is itself the storage), the data is cached in CPU cache. For initial and random operations, however, caching effect can not help.



(a) Initial Write (b) Re-Write (c) Random Write

(d) Initial Read (e) Re-Read (f) Random Read

Fig. 5: Comparing *ZonFS* with Tmpfs and Ramfs for varying record size. We used a single thread for I/O operations on a 10 GB file.

In Figure 5(c), we see random write performance is same or slightly better than Ramfs for 64K, 256K and 1M records.

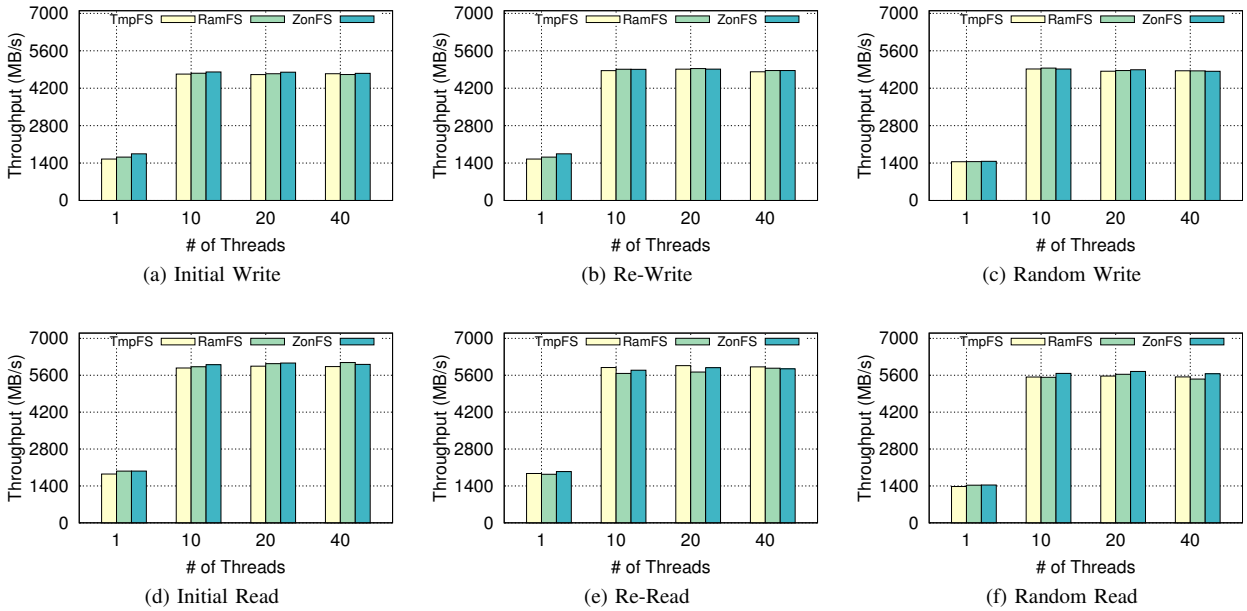
Figure 5(d)-(f) show results for read operations. For the read operation, no page allocation occurs because it is performed on an existing file. Therefore, it shows similar performance to Ramfs. But re-read results show noticeable improvement of maximum 6.4% and 5.8% compared to Ramfs and Tmpfs.

We have also observed that the performance gradually improved as the size of the record grows, whereas the performance degraded for the 1 MB record. To explain this performance variance, we need to take into account the effect of record size. Note that, as the size of the record grows, a smaller number of requests occur because the size of the data requested at a time increases. Thus, it reduces the number of function calls, resulting in performance gain. The reason for the performance degradation in 1 MB records can be explained by cache effect: bigger record size means greater CPU cache miss penalty, which can negate the benefit of decreased number of write requests.

2) Impact of Multiple Threads: Figure 6 shows the throughput comparisons of *ZonFS*, Ramfs and Tmpfs with varying number of I/O threads. In this experiment, we use a mix of read and write threads for files of different sizes. In all cases, there was no significant performance difference as compared to the existing Linux memory-based file systems. We can see dramatic performance gain as the number of threads increases from 1 to 10. This is because multiple threads leads to parallel file I/Os. But those gains are saturated to around 5 GB/s (write) and 5.6 GB/s (read) for 20 and 40 threads. The reason for this saturation is, we suspect, 10 threads sufficiently exploits memory bus bandwidth. Note that our experimental test-bed was not a NUMA memory architecture. However, NUMA (Non-Uniform Memory Access) system enables us to solve the problem of memory bus or controller contention by assigning independent memories to each processor node. Hence throughput will increase after 10 threads for NUMA, but it will compel NUMA-aware file page allocations.

IV. Related Work

There have been several prior studies on the file system for non-volatile memory [10], [11], [11]. BPFS [10] is a file system for non-volatile byte-addressable memories. It focuses on the problems of copy-on-write in file systems and proposes shadow paging technique to consistently update changes on the file system tree at fine granularity. The measured file system performance was too low to be used for actual SCM file system. SCMFS [12] is another memory file system for storage class memory connected to a memory bus. SCMFS is a file system developed using the Linux memory manager, and suggests a simple file system structure. SCMFS uses the virtual memory page as a file system page, thus significant TLB miss overhead can degrade overall file system performance. SCMFS also proposes a technique for partitioning memory into zones. Conquest [11] uses a battery backed DRAM for storing file metadata and small files to improve the overall file system performance. Unlike these file systems, *ZonFS* is developed by extending Ramfs, which implements the file system in the page cache. *ZonFS* also uses the Memory Partition technique. However, Ramfs is a DRAM based file system, and it can not be used as a file system for SCM. So we modify the Linux kernel code to develop SCM-specific file system. In particular, it minimizes unnecessary calls to kernel code by separating the DRAM memory pages in the page cache used for various purposes and the SCM pages for file system use.



(a) Initial Write (b) Re-Write (c) Random Write

(d) Initial Read (e) Re-Read (f) Random Read

Fig. 6: Scaling performance comparisons of *ZonFS* with Tmpfs and Ramfs by varying the number of I/O threads.

V. Conclusion

In this paper, we proposed a memory file system to partition Linux memory zone to efficiently utilize storage class memory as storage. We have implemented the proposed memory file system, *ZonFS* by extending the Ramfs implementation. *ZonFS* improved the performance by preventing storage class memory (SCM) space for storing files from being used for other purposes on Linux and minimizing unnecessary overhead when using the SCM as storage in the page allocation process. We conducted series of experiments to evaluate the *ZonFS* using IOzone. The experimental results showed that the performance of initial write is improved up to 9.1% and 13.8%, respectively, as compared to Ramfs and Tmpfs. The read operation depicts a performance gain up to 8% when compared against Tmpfs. Ensuring file system consistency is an important issue in developing SCM file system. We will consider the crash-consistency problem in *ZonFS* for future work.

Acknowledgments

We thank Preethika Kasu for her proofreading and constructive comments that have improved the paper. This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (NRF-2017R1D1A1B03032763) and Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (No.B0101-16-0644, Research on High Performance and Scalable Manycore Operating System).

References

[1] D. Halupka, S. Huda, W. Song, A. Sheikholeslami, K. Tsunoda, C. Yoshida, and M. Aoki, "Negative-resistance Read and Write Schemes for STT-MRAM in 0.13 um CMOS," in *2010 IEEE International Solid-State Circuits Conference - (ISSCC)*, Feb 2010, pp. 256–257.

[2] H. Oh, B. Cho, W. Cho, S. Kang, B. Choi, H. Kim, K. Kim, D. Kim, C. Kwak, H. Byun, G. Jeong, H. Jeong, and K. Kim, "Enhanced Write Performance of a 64-Mb Phase-change Random Access Memory," in *ISSCC. 2005 IEEE International Digest of Technical Papers. Solid-State Circuits Conference, 2005.*, Feb 2005, pp. 48–584 Vol. 1.

[3] Z. Chen, H. Wu, B. Gao, P. Yao, X. Li, and H. Qian, "Neuromorphic Computing based on Resistive RAM," in *Proceedings of the on Great Lakes Symposium on VLSI 2017*, ser. GLSVLSI '17. New York, NY, USA: ACM, 2017, pp. 311–315. [Online]. Available: http://doi.acm.org/10.1145/3060403.3066873

[4] Intel, "Revolutionizing Memory and Storage," https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html.

[5] G. Dhiman, R. Ayoub, and T. Rosing, "PDRAM: A Hybrid PRAM and DRAM Main Memory System," in *Proceedings of the 46th Annual Design Automation Conference*, ser. DAC '09. New York, NY, USA: ACM, 2009, pp. 664–469. [Online]. Available: http://doi.acm.org/10.1145/1629911.1630086

[6] T. Kgil, D. Roberts, and T. Mudge, "Improving NAND Flash based Disk Caches," in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ser. ISCA '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 327–338. [Online]. Available: http://dx.doi.org/10.1109/ISCA.2008.32

[7] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable High Performance Main Memory System Using Phase-change Memory Technology," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ser. ISCA '09. New York, NY, USA: ACM, 2009, pp. 24–33. [Online]. Available: http://doi.acm.org/10.1145/1555754.1555760

[8] D. Bovet and M. Cesati, *Understanding The Linux Kernel*. Oreilly & Associates Inc, 2005.

[9] D. Capps, "IOzone Filesystem Benchmark," http://www.iozone.org.

[10] 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 *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: ACM, 2009, pp. 133–146. [Online]. Available: http://doi.acm.org/10.1145/1629575.1629589

[11] A.-I. A. Wang, G. Kuenning, P. Reiher, and G. Popek, "The Conquest File System: Better Performance Through a Disk/persistent-RAM Hybrid Design," *Trans. Storage*, vol. 2, no. 3, pp. 309–348, Aug. 2006. [Online]. Available: http://doi.acm.org/10.1145/1168910.1168914

[12] X. Wu and A. L. N. Reddy, "SCMFS: A File System for Storage Class Memory," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: ACM, 2011, pp. 39:1–39:11. [Online]. Available: http://doi.acm.org/10.1145/2063384.2063436