Cluster Computing
https://doi.org/10.1007/s10586-018-2833-4

Understanding the performance of storage class memory file systems in the NUMA architecture

Jangwoong Kim[1] · Youngjae Kim[1] · Awais Khan[1] · Sungyong Park[1]

Received: 20 March 2018 / Revised: 26 May 2018 / Accepted: 17 July 2018

Abstract
Recent developments in storage class memory (SCM) such as PCM, MRAM, resistive RAM (RRAM), and spin-transfer torque (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 have unnecessary overheads when adopted for SCM file system. Therefore, we propose a new memory-based file system using Memory Zone Partitioning called *ZonFS*, 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 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. Our extensive evaluations indicate that *ZonFS* has up to 9.1 and 14.1% higher I/O throughputs than native Ramfs and Tmpfs. Moreover, we also analyze performance behavior of *ZonFS* under the non-uniform memory access architecture of SCMs on a 40 manycore machine with various configurations such as file sharing level and file stripping level. Our evaluations show that memory controller contention and inter-node link congestion significantly affect the file system's performance and scalability.

Keywords File systems · Storage class memory · Non-uniform memory access (NUMA)

1 Introduction

Emergences of non-volatile memory such as STT-RAM (spin-transfer torque) [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. SCMs fundamentally differ from traditional block devices such as hard disk drives and solid-state drives, which should be accessed through the I/O block layer in OS. On the other hand, the SCM can be accessed through memory load and store instructions by CPU. SCM is non-volatile and provides low latency near DRAM latency. In order to solve the high energy problem caused by the DRAM main memory system, a hybrid memory system combining DRAM and SCM has been proposed as shown in Fig. 1 [5–7]. 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 the DRAM. These file systems are basically implemented using Linux memory management techniques. When a file is created, memory pages are allocated by OS. When the file is read, its corresponding pages on the main memory (DRAM) are referred. This allows the file system to be easily implemented in main memory. However, traditional Linux-based memory file systems, such as Tmpfs and Ramfs have not been implemented considering such a hybrid memory design. It allocates pages without considering a memory

✉ Sungyong Park
parksy@sogang.ac.kr

Jangwoong Kim
ski6812@sogang.ac.kr

Youngjae Kim
youkim@sogang.ac.kr

Awais Khan
awais@sogang.ac.kr

[1] Department of Computer Science and Engineering, Sogang University, 35 Baekreom-ro, Mapo-gu, Seoul, Korea

Published online: 24 July 2018

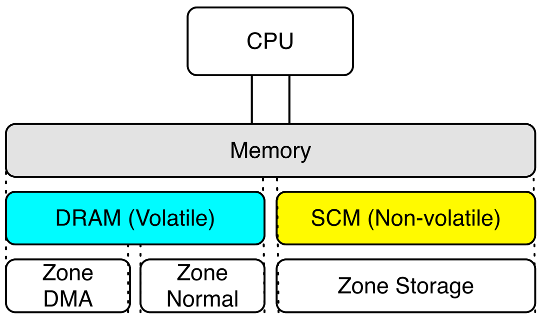


Fig. 1 Illustration of memory zone partitioning for hybrid memory combining DRAM and SCM

type. Hence, it is not possible to know in which area of the memory the page is allocated. Moreover, when a system power failure occurs, memory objects of data in the non-volatile SCM region should be preserved. However, if file control blocks (inode cache) corresponding to memory objects in DRAM, the memory objects in SCM can't be recovered because metadata is lost. 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.

In this paper, we propose *ZonFS*, a memory file system for SCM in a hybrid memory using *Linux Memory Zone Partitioning* [8]. 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 [9]. In particular, the 64-bit kernel does not use the HighMem Zone. To isolate SCM from DRAM, we define Storage Zone for SCM and specify the entire SCM address space as the Storage Zone. Figure 1 illustrates our proposed Memory Zone Partitioning technique for the hybrid memory employing DRAM and SCM. When file systems are built upon SCM, memory pages of the file system in SCM will be separately managed from normal pages allocated from DRAM.

These SCMs can be directly attached to the memory slot along with DRAMs. Recent Linux kernels also have drivers that automatically detect non-volatile memory regions [10]. However, under the non-uniform memory access (NUMA) architectures, where memory devices are distributed to multiple nodes, memory access latency is dependent on the location of destination memory. We also studied the impact of the NUMA architecture to *ZonFS*.

In this work, we make the following contributions:

- *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 emulate 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() take Linux kernel's full generic file I/O path including functions such as *generic_perform_write*(), unnecessary overheads are entailed. These include (i) checking dirty pages and (ii) inserting pages into LRU list. When DRAM is used for page cache, if the number of dirty pages are bigger than a certain threshold, kernel's generic file I/O operations will write them back to disk and make dirty pages under that threshold value. 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 not only extending Linux Ramfs but also 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 [11]. As a representative example, for write operations, *ZonFS* showed up to 9 and 13% performance improvements over native Ramfs and Tmpfs.
- *Analyzing the NUMA behavior of* ZonFS we also analyzed the performance behavior of *ZonFS* on the 40 core NUMA machine. Since remote access, memory controller contention and inter-node link congestion are influencing factors to memory-based file systems, we conduct various tests to understand the effects of aforementioned factors under several circumstances. Experimental results show valuable findings as following: (1) memory controller contention might hurt the file system performance as the load grows. (2) Inter-node link overheads also hinder file systems, especially for overloaded and remote access-dominant circumstances. (3) In the overloaded circumstance, distributing SCMs and stripping file data to memory nodes can lead to improved throughput and affect scalability (either positively or negatively).

The rest of this paper is organized as follows. Section 2 provides some basic understandings about Linux page cache and NUMA architecture. Section 3 describes the memory access method and its implementation for *ZonFS*, and we propose considerations that memory file systems need to take into account under NUMA architecture. Section 4 shows performance comparison results of *ZonFS* with Linux memory file systems, Tmpfs and Ramfs along with experimental results of *ZonFS* under NUMA many-core systems with various configurations. Section 5 presents the related work. We conclude our work and suggest design principles of memory file system in Sect. 6.

2 Background

In this section, we first discuss the memory area management in Linux kernel. By tracking the Linux kernel function calls of write and read in Ramfs, we explain the code-level I/O behavior of the page cache based file system. Second, we discuss the NUMA architecture and its performance impact on the file systems relying on page cache in OS.

2.1 Memory management in Linux kernel

2.1.1 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 0–16 MB of the memory area.

The Normal zone is used for general memory allocations. The purpose of the HighMem zone is to relieve 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 zones 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 1. Especially, we have added a new E820 entry called *E820_STORAGE* for SCM storage usage of *ZonFS*.

Table 1 E820 memory map with Storage Zone

E820 types	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

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.

2.1.2 I/O flows for write and read requests

In Linux, file I/O data goes through page cache. Since accessing the disk for every file request is inefficient, OS stores the data 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 another purpose when the physical memory is low, kernel write-backs the pages into the 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. To address our problem definitions and solutions, we

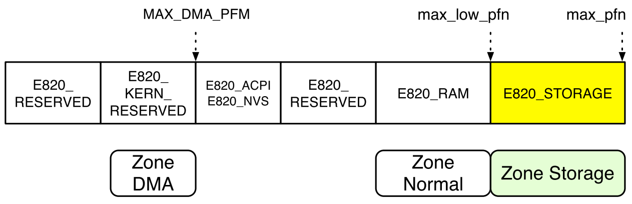


Fig. 2 Linux memory zone partitioning with storage zone

investigate the I/O path of Linux Ramfs in detail from which *ZonFS* originates. 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. If it acquired that page, function *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 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 Fig. 3a, b. page cache. Therefore, page cache pages of Ramfs are never write-backed.

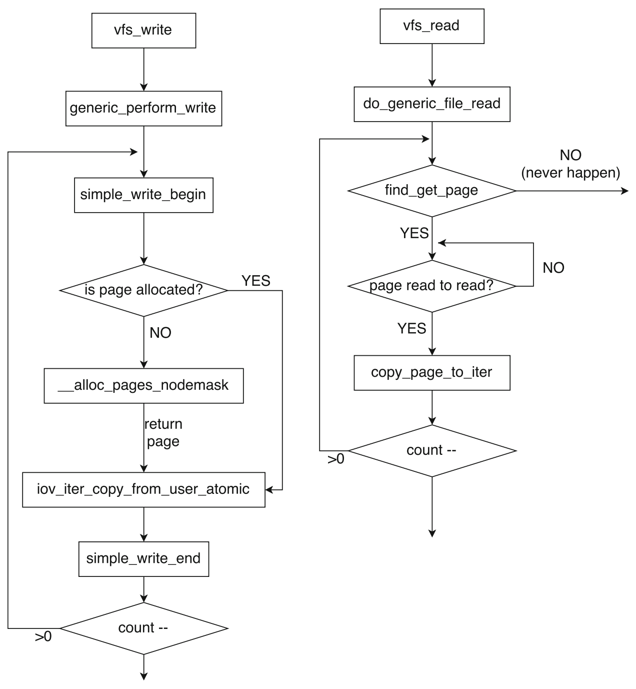


Fig. 3 Write and read I/O flow of Ramfs

2.2 Non-uniform memory access

In the traditional Von Neumann architecture, memory plays an intermediate role between fast CPU(s) and slow disk. Therefore, the performance of the memory system has a significant impact on the entire system. With the help of the memory system, the data on the disk can be cached in memory, which accelerates data access time. However, after the advent of multi-core systems for parallel computing, memory bottlenecks occur. This is because only one CPU can access memory at a time. In spite of the architectural techniques to mitigate the effect of long memory latencies such as pre-fetching, out-of-order execution, speculation and multi-threading, memory wall problems still remain as a performance degradation factor that hinders the optimal use of processor's capabilities gained by multi-core technologies [12].

To overcome the problem, memory devices began to be partitioned into groups of memories along with private CPU(s) as shown in Fig. 4. Each group is called a memory node. Memory accesses by a CPU on the same node is called local access and on another node, is called remote access by a CPU.

Each memory node has its memory controller which manages incoming memory requests. Two nodes are connected via inter-node link through which memory requests

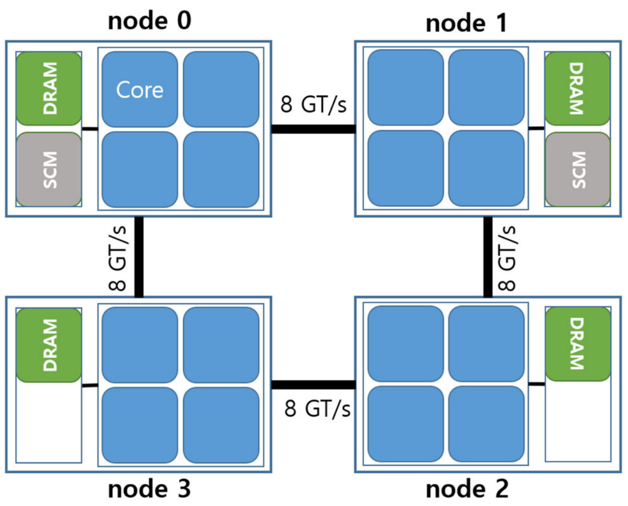


Fig. 4 An example of NUMA architecture with four sockets (Testbed-II)

are transferred. Two nodes have *n-hop* distance if they are able to reach each other by traversing at most *n* inter-links. *2-Hop* distance incurs longer latency than *1-hop* or *0-hop* distance.

Since remote access needs to pass through inter-node link, remote access is slower than local access In such NUMA systems, remote node memory access, memory controller and inter-node link contention can adversely affect the performance of the memory file systems that we have designed for SCM. Under the NUMA environment, SCM devices can be directly attached to the system via memory slots along with DRAMs as shown in Fig. 4. In such cases, both DRAMs and SCMs together contribute to constructing the memory address space based on the order in which each memory is physically attached.

3 Storage class memory file systems

3.1 Design and implementation for *ZonFS*

In this section, we describe the design principles for the SCM file system using the Linux page-cache. We implement *ZonFS* with the following design goals—(i) *ZonFS* manages the pages in storage zone separately from those in DRAM memory zone, and (ii) it 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.

ZonFS uses Linux page cache to store file data. In current Linux kernel, all the page cache pages are the potential victims of virtual memory management. They can be swapped out by kernel when the physical memory is low. The kernel periodically checks dirty pages and writes back to save VM resources for later use. However, although a page in page cache that belongs to *ZonFS* is selected as the victim of above VM management, *PG_UNEVICTABLE* flag, set to all of pages for *ZonFS*, prevents the page from being swapped out. Therefore, a *ZonFS* page in page cache will not be write-backed by VM management.

Unnecessary operations on the above-mentioned memory access paths include the followings: first, when a page cache page is allocated, although the page is set flag *PG_UNEVICTABLE* which prevents the page from being swapped-out, the kernel needlessly adds pages into LRU list which is a list containing page replacement candidates. This causes the LRU list maintaining overhead. Second, it has dirty page check overhead. Write processes periodically count dirty pages in page cache. If the number of the dirty pages exceeds a certain threshold, it write-backs all the dirty pages and in the worse case, throttles the process for some time.

Figure 5a describes the step-by-step process of memory management. When a write request arrives, if it is the first write, then a new page is allocated from OS page cache. It then sets the page with *PG_Dirty* flag and adds the page to 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 has an impact on 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 management. DRAM accesses are triggered by application memory requests, and SCM accesses by requests for files. For DRAM accesses, we fully exploit the Linux virtual memory layer. However, for SCM accesses, we provide a simple access path which uses the OS page cache to store files but prevents them from being virtual memory management candidates.

To differentiate memory access paths 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 implement our design by only using DRAM, we have allocated new storage zone in a specific range of DRAM. All the flags *__GFP_STORAGE* corresponding to the file inodes in *ZonFS*, are set 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 process for the pages of *ZONE_STORAGE*. LRU list insertion was done after page cache allocation. We do not add the pages to the list in the case of file pages residing in *ZONE_STORAGE*. We also skip dirty page check for the file page writes in *ZONE_STORAGE*. We have restricted write operation to execute dirty check for process memory requests only, thus preventing write-back operations and I/O process throttling. The new management of page cache for *ZonFS* is described in Fig. 5b.

The addition of new SCM zone also lessens zone contention. For a memory zone, each CPU has per-CPU page list for that zone. When a memory request arrives, the per-CPU list for that CPU is locked and the page allocation is handled, while other memory requests to that CPU blocked. However, by adding a new zone, we can reduce per-CPU page list contention, since the zone is devoted to only SCM page requests but not for DRAM requests. This allows parallelized allocation request for DRAM and SCM from a same CPU.

Note that storing only file contents into *ZONE_STORAGE* does not make the data durable after crash and File metadata, inodes must be kept in that zone. Failure of this will result in inaccessible isolated data. In

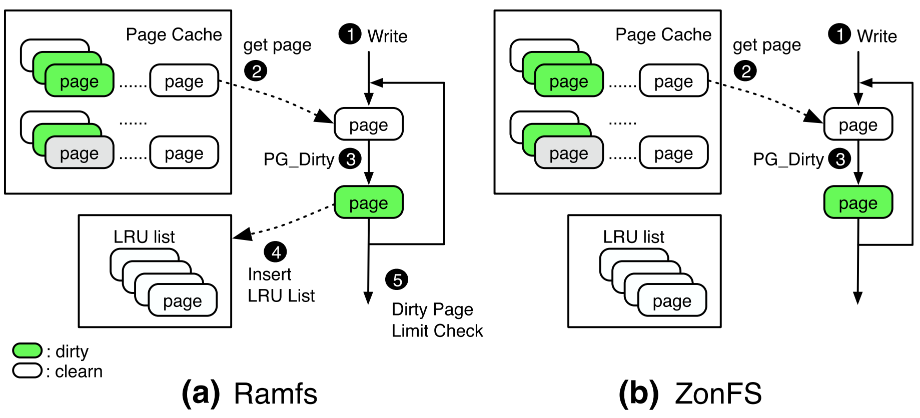


Fig. 5 Illustration of the relationship between Write operation and Page Cache. **a** and **b** Operation procedures for Ramfs and *ZonFS*, respectively

Linux kernel, inode structure is allocated by slab allocator, which is used to allocate frequently allocated structures such as inode, per-thread structure, etc. Slab allocator reduces the allocation and de-allocation overhead. It pre-allocates caches dedicated to a certain structure, and actual structure is allocated from those pre-allocated caches. The 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, all the inode structures are persistent.

3.2 Design considerations for NUMA-aware memory file systems

In this section, we discuss several consideration points that memory file systems need to take into account when implementing the NUMA-aware *ZonFS*.

NUMA latency the asymmetric NUMA latency of memory access makes it difficult to guarantee stable file request time [13], since latency is significantly affected by the location of required memory. Memory accesses to remote memory node incur far slower latency. Therefore, for I/O intensive applications, excessive remote accesses can hurt the performance. Placing data and threads on the same socket is recommended for high performance. In current Linux kernel, page allocation in page cache follows *first-touch* policy, where a page allocation occurs at the moment when data is initially “touched”, or accessed. Therefore, pages for files are allocated in the node of the thread that initially writes to those pages. This *first-touch* policy, however, can not guarantee data-thread affinity, since thread scheduler or memory subsystem can migrate threads or data to another NUMA node, which leads to sub-optimal I/O performance. JeriFS [14] dynamically migrates threads upon issuing I/O request, thus fully achieving thread-data affinity.

Memory controller congestion the degree of memory controller congestion also affects the performance of the file system. Chandru and Mueller [15], addresses the detrimental effect of memory controller contention to manycore systems. Each memory node has a maximum memory controller bandwidth. Since the “local only” data placement policy can cause congestion of the controller, file system must be carefully designed to ensure that the controller on each node is not overloaded. There can be a file system solution to reduce the contention of a memory controller. For example, in case file requests begin to be concentrated to a single NUMA node, it possibly is beneficial if it allocates data to another node for later *first-touch*, although it will generate remote memory requests. When the congestion is extremely intensive, migrating residing local data to another node can also help to reduce controller contention.

Inter-node link congestion along with controller contention, file system I/O can occur due to inter-node link congestion. When a file is requested from a remote thread, data of the file is requested to be transferred through the inter-node link. Each link also has a maximum data transfer rate, which is usually lower than maximum memory bandwidth. Therefore, this also restricts us to design memory file system by considering the contention of inter-node links. The effect is harmful, especially for applications with lots of remote accesses. For example, our testbed’s single node has 59.7 GB/s memory bandwidth, while a QuickPath Interconnect, Intel (QPI) has 7.87 GB/s data transfer rate [16]. Namely, 7.87 Giga-bytes file request per a second is sufficient to stress the inter-node link, which severely hurts the performance. In that case, where a certain links are highly contended, we can bypass the congested links by selecting an alternative path. For example, if node 0 is accessing data from node 2 and the links 0–1 and 1–2 are contended, redirecting the route to the path of links 0–3 and 3–2 goes through less congested paths, thus lowring file request latency.

I/O buffer placement I/O buffer placement is an another factor that file systems need to take into account. POSIX-compliant file systems' requests use *read()* and *write()* system calls which entail buffer allocation. Afterwards, all of file requests are handled via the buffer, except for files which are memory-mapped by the *mmap()* system call. Buffer allocations also follow *first-touch* policy, therefore the location of the buffer significantly affects the entire I/O subsystem due to the NUMA effect. JeriCache [14] is a replacement of Linux page cache, and it partitions the DRAM cache into "slices". Each slice is dedicated to a specific NUMA node. This design maximizes data-buffer affinity and also enables user-level applications to place data to proper node, thus achieving layout-aware data placement.

4 Experimental results

In this section, we compare the performance of *ZonFS* with traditional memory file systems in Linux such as Ramfs and Tmpfs and evaluate the performance impact of NUMA architecture over *ZonFS*. Consistency guarantee is one of the most significant factors in designing file systems. However, since the primary contribution of this paper is to present Zone Partitioning technique to design SCM file systems, *ZonFS* currently does not support consistency guarantee.

4.1 Experimental setup

4.1.1 Testbed

To compare the performance of *ZonFS*, we use Intel server of eight cores with two 4-Core Intel Xeon Processor E5410 CPUs (Testbed-I). The server is equipped with a total of 16 GB DRAM, where 10 GB of DRAM area is assigned as storage zone to simulate non-volatile memory. In addition, to analyze the performance behavior of *ZonFS* in the NUMA server architecture, we use Intel server of 80 cores with four 20-Core Intel Xeon E5-4650 processor CPUs (Testbed-II). The server is installed with 240 GB DRAM, where 200 GB of DRAM area is allocated to emulate non-volatile memory.

4.1.2 Workloads

We have employed IOzone [11] benchmarking tool to evaluate the performance of *ZonFS* against other Linux memory file systems such as Ramfs and Tmpfs. 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 *ZonFS* using IOzone [11] by changing record size and the number of threads. We have varied the file record size from 4 KB to 1 MB and the number of threads from 1 to 40. For write operations in IOzone benchmark, we have not issued file flush operations.

To evaluate behavior of *ZonFS* under the NUMA server environment, we have developed an in-house benchmarking tool to measure file system throughputs. In particular, the benchmarking tool allows to measure throughputs of *ZonFS* by changing not only file striping level such as one, two, and four , but also file sharing level such as *low*, *medium* and *high*.

4.1.3 Implementation

We have modified the Ramfs in Linux kernel version 4.7.4. to develop *ZonFS*. The total number of modified kernel code lines is approximately 50. *ZonFS* source code is downloadable at https://github.com/wanyaworld/ZonFS.

4.2 Performance evaluation of *ZonFS*

In this section, we show our performance evaluation results for *ZonFS* with a variety of workload patterns.

4.2.1 Impact of record size

Figure 6 compares *ZonFS* with native Tmpfs and Ramfs for different file operations by varying record size. In this experiment, we have measured the performance of single I/O thread file operations on a single 10 GB file.

Figure 6a–c show results for write workloads. Figure 6a shows the performance comparison for initial write. We have observed that *ZonFS* shows the maximum performance improvement of 7.5 and 11.4% for 4K record size over Ramfs and Tmpfs, respectively. We have similar performance improvement for bigger record size as well. Figure 6b 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 have 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

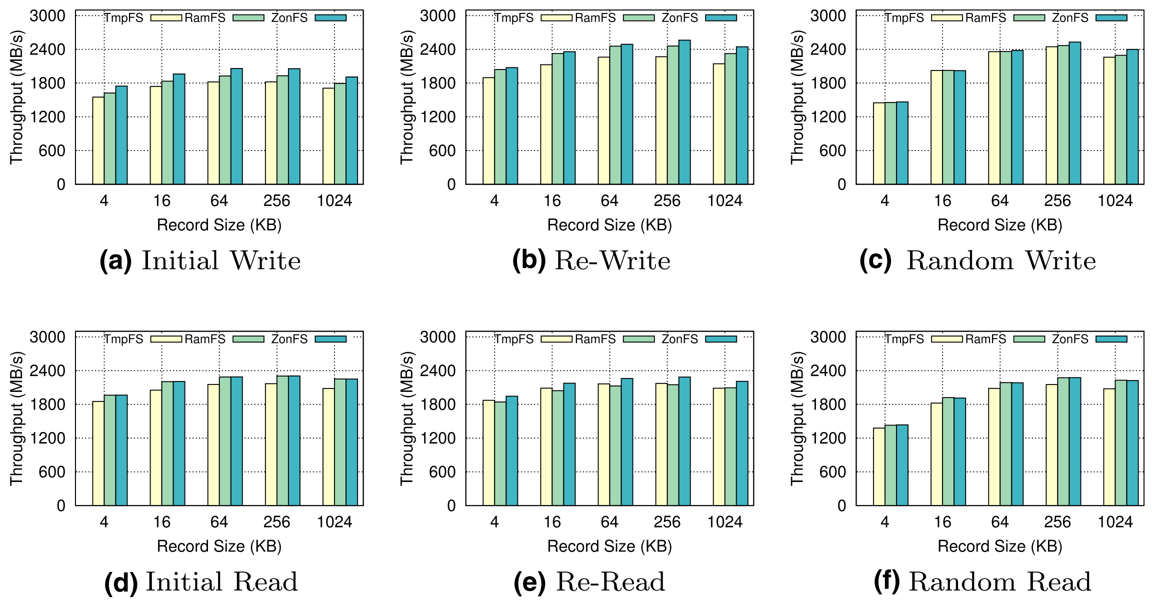


Fig. 6 Comparing *ZonFS* with Tmpfs and Ramfs for varying record size. We used a single thread for I/O operations on a 10 GB file

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* uses page cache as storage area, the data is also cached in CPU cache. For initial and random operations, however, caching effect barely helps since they have little locality. In Fig. 6c, we can see random write performance is same or slightly better than Ramfs for 64K, 256K and 1M records.

Figure 6d–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.

4.2.2 Impact of multiple threads

Figure 7 shows the throughput comparisons of *ZonFS*, Ramfs and Tmpfs while changing/varying number of I/O threads. In this experiment, we have used 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 around 5 GB/s (write) and 5.6 GB/s (read) for 20 and 40 threads, respectively. The reason for this saturation is, we suspect, 10 threads sufficiently exploits memory bus bandwidth. Note that our experimental test-bed in this test was not a NUMA memory architecture. However, NUMA 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.

4.3 Evaluating the scalability of *ZonFS* under the NUMA architecture

In this section, we have evaluated the performance impact of *ZonFS* over the NUMA architecture.

4.3.1 Local and remote memory latency

To measure memory latency, we use Memory Latency Checker (MLC), a benchmark tool provided by Intel [17]. The test was conducted on Testbed-II, which is the NUMA architecture. MLC measures memory latencies for every possible pair of cores and calculates the average latencies between nodes. Specifically, a thread is bound to a node, and the thread in the source node issues a memory access to

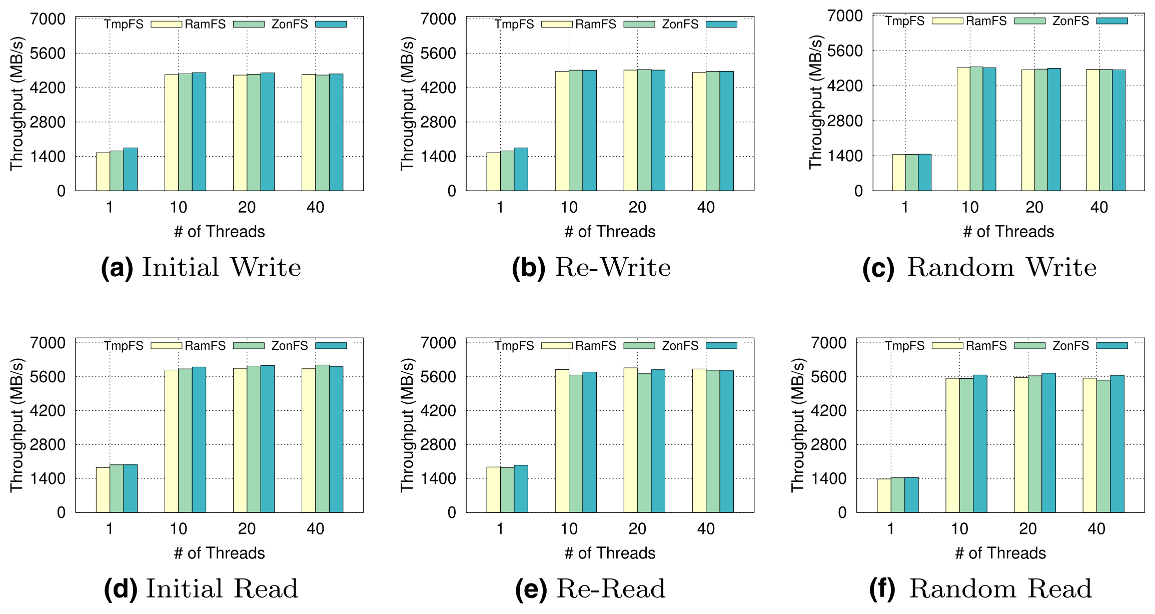


Fig. 7 Scaling performance comparisons of *ZonFS* with Tmpfs and Ramfs by varying the number of I/O threads

the destination memory node. Table 2 shows memory latency measurements from source memory nodes to destination nodes. The value of each cell in the table represents measured memory latency for the pair of source and destination nodes. For example, the cell value of (*src*:*dest*) = (0:0) is a latency of memory access from node 0 to 0, which is a local memory latency, and (*src*:*dest*) = (0:1) represents a latency from node 0 to 1, which is a *1-hop* remote memory latency, since the memory access goes through only one inter-node link. In the same manner, (*src*:*dest*) = (0:2) indicates a *2-hop* remote memory which traverses two inter-node links. The average latencies of *local*, *1-hop* and *2-hop* are on average 75, 195 and 220 ns, respectively. From Table 2, we can observe that memory accesses which span several inter-node links lead to higher latency.

4.3.2 Single SCM module and distributed SCM modules

In the NUMA architecture, memory controller congestion and inter-node link congestion may occur depending on the memory access location of a thread. In order to identify these problems, we have experimented with the following three configurations:

- *Single* in this configuration, only one memory module is active. That is, a thread accesses a local memory only from one node, and accesses remote memory from three other nodes. Therefore, only 25% of memory accesses are local while 75% are remote.
- *Local* this configuration distributes the memory modules one by one to each of the four memory nodes. All threads in this configuration perform local memory accesses.
- *Fair* this configuration also allows each node to have one memory module as a storage, similar to *Local*. However, in this setting, threads perform 25% of local memory accesses and 75% of remote accesses. This remote access ratio is equal to that *single*. The only difference is contention of inter-node links. In configuration *single*, every remote access is transferred through inter-node links connected to one node, while in *fair*, they are uniformly transferred via all inter-node links. This configuration can contribute to different contention of inter-node links, which helps understand the effect of inter-node contention to memory file system performance.

Figure 8 shows the results of the experiments to compare throughputs measured for three different configurations by varying the number of threads. Each thread repeatedly reads two files thus for 80 threads and 160 files are read in total. Specifically, this experiment aims to observe an increase in memory access time when the memory

Table 2 Memory latency of NUMA system

Source nodes (ns)	Destination nodes			
	0	1	2	3
0	76.9	212.2	228.3	210.9
1	196.3	75.6	194.9	240.1
2	227.4	195.1	75.4	193.9
3	195.0	227.2	195.9	76.1

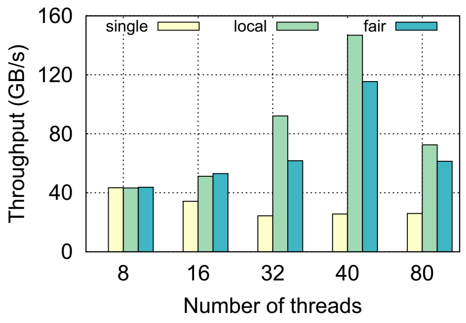


Fig. 8 Evaluation of distributed storage memory

controller in the memory module becomes congested or the inter-node links between nodes become congested.

Note that in *single*, every request is concentrated into a single node, which stresses the memory controller and two QPIs connected to that single node. In Fig. 8, we can observe that when the number of threads is eight, there is little difference in throughput between three configurations. It means that a small number of threads do not cause the memory controller or QPIs to become congested. However, as the number of threads increases, we can see that *single* begins to suffer from performance degradation while *local* and *fair* linearly increase throughputs with respect to the increased number of threads up to 40 threads. We suspect that it is due to the contention of the memory controller and QPI. For example, *fair* shows less than 1% higher performance than *single* for 8 threads, while 351% higher for 40 threads. Therefore, uniformly distributing SCMs into nodes in the NUMA architecture can help to obtain higher throughput by relieving the contention of memory controller and QPI especially for I/O intensive applications.

When comparing throughputs of *Local* and *Fair* in Fig. 8, we can see that *Local* shows lower throughputs than *Fair*. We suspect that the performance difference between them is due to the congestion of the inter-node links between the nodes. Inter-node links such as QPI or IF (Infinity Fabric, AMD) connecting NUMA nodes are responsible for transferring data between them. Cheng et al. [18], discusses data traffic stagnation due to QPI congestion. The Intel server of Testbed-II has QPIs of 8 GT/s transfer rate [16], which is approximatively equal to 7.87 GB/s. Note that in case of *Local*, threads do not cause remote request while in *fair* configuration, as shown in Fig. 8, the throughput is about 40 GB/s with eight threads, which means 10 GB/s for every single node and 25% of requests are remote accesses. Accordingly, roughly 7.5 GB of remote access is being transferred per a second in a single node, which does not congest the QPI of Testbed-II. However, with a larger number of threads, heavier I/O request arrivals begin to stress QPIs exceeding its maximum link bandwidth, 7.87 GB/s and the gap between *Fair* and *Local* gets bigger. This makes the throughput of *Fair* 49% worse than that of *Local* for 40 threads. It is a valuable finding in that we have discovered that the performance difference does not simply come from an inequality between local and remote memory latencies.

With *Local* and *Fair*, we also observe distributing storage memory modules to nodes scales up throughputs up to 40 threads, whereas *single* never shows any scalability. Memory controller congestion and QPI contention are possible factors that prevent performance scalability of *ZonFS*. For example, with 80 threads, every I/O rushes to a single node and stresses the controller and QPIs intensively. Although other two configurations show a degree of scalability, we can see that scalability limitation is 40 threads. We suspect that memory bandwidth limits the scalability of *ZonFS*. Testbed-II with four nodes is able to offer theoretical bandwidth, 238 GB/s [16]. Figure 8 shows that 40 threads are performing 160 GB/s throughput, which is roughly 61% of theoretical memory bandwidth. 80 Threads exceed the theoretical memory bandwidth along with controllers and QPIs extremely congested, which all could lead to degraded performance for 80 threads.

4.3.3 Impact of file stripping

We have evaluated the impact of file stripping of *ZonFS* to throughput with respect to the number of I/O threads. We have conducted experiments by stripping a file into different number of nodes and under different file sharing levels. Especially, we divide a file into chunks and distributed them to memories of NUMA nodes. In terms of file sharing level, we have considered three levels: *low*, *medium* and *high*.

- *Low* when a sharing level is *low*, threads perform I/Os with their private files.
- *Medium* in *medium*, threads can write on the same file. Specifically, each thread handles mutually exclusive chunks of the same file.
- *High* *high* is similar to *medium* because threads can write on the same file, however, all threads can write on the entire chunks of a file, thus two or more than two threads can access the same chunk of a file.

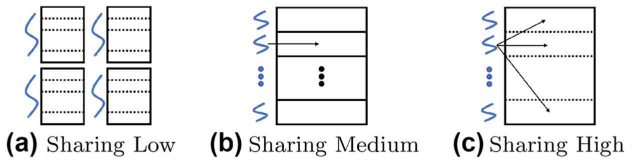


Fig. 9 Threads execute I/Os with private files, shared file of different parts and shared file of the whole part each for *low*, *medium* and *high*, respectively

Figure 9 shows results for three different sharing levels. When a file is not stripped, every memory page of the file is stored in local memory. For two-stripping configuration, pages are sequentially distributed both in local node and its next neighbor node. Four4-stripping uses all the nodes as storage memory. Therefore, one, two and four stripping settings generate 0, 50 and 75% of remote access, respectively. We have also changed the number of threads to evaluate the scalability for both node distribution and file stripping experiments. To fully understand the effect of *first-touch* policy, workloads are limited to read requests of 4 KB page size.

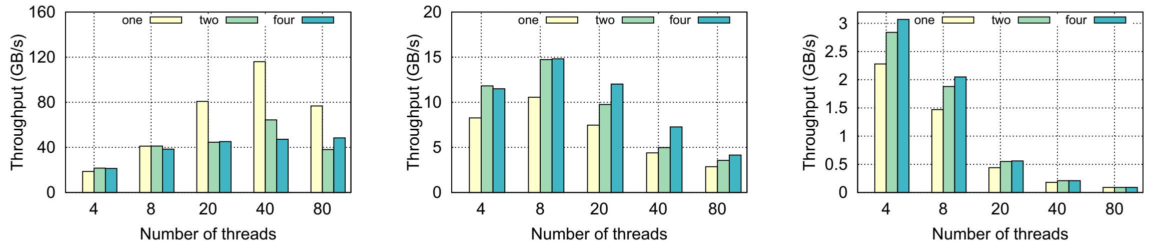
Figure 10 describes experimental results of file stripping with three different file sharing levels. Overall, higher sharing level leads to poorer performance. Sharing level *low*, which only performs I/Os with private files, shows better performance than the remaining levels. Son et al. [19], addressed that the update overhead of *atime* of inodes can degrade file system performance. When a file is accessed, either read or write, access time field of its inode is updated. Since in *high* and *medium*, all the I/Os belong to a single shared file, every request generates write traffic to update the *atime* of the inode [20]. We have to note that all the writes are to the same memory address of *atime* field, which might cause tremendous congestion. *Medium* and *high* also show a remarkable difference for all number of threads. Also, Min et al. [21], addressed the negative effect of overhead in updating reference count, when sharing level is high. When a page cache page is referenced, its reference count is atomically incremented by one. In *high*, all threads access the same pages, which makes atomic integer updates the performance bottleneck. In this experiment, reference counting overhead leads to lower throughput of *high* than *medium*.

We also observe that the effect of degree of congestion on the memory controller and QPI can be affected by the file striping level. Basically, stripping file data has a tradeoff; it incurs remote accesses and thus QPI contentions, and at the same time, it also relieves the stress of memory controller by distributing memory accesses to nodes. In case of *low*, it does not give any performance benefit by stripping files. Specifically, for 40 threads, stripping to four nodes degrades the performance by 145%, compared to the case without stripping, since unnecessary remote accesses cause QPI contentions. For four and eight threads, these effect are not seen, because workloads are not heavy enough to stress QPIs. Eight threads generate approximately I/Os at 5 GB/s for a single node, while a single QPI has 7.87 GB/s of data transfer rate.

However, for *medium* and *high*, which all have higher file sharing level, stripping to two or four nodes shows better throughput than one without stripping for every test. In detail, when sharing level is *medium*, four-stripping begins to outperform two-stripping from eight threads. The effect reaches a peak for 40 threads with 65% higher throughput. For *high*, four-stripping performs better than two-stripping for all number of threads with the biggest difference of 17% for 40 threads. When a file is shared by multiple threads, stripping helps because chunks of a file is evenly distributed to memory nodes, which leads to reduced contention of controllers and QPIs.

This experiment gives us another lesson: for applications that intensively share same files, especially in the manycore NUMA architecture, striping the files into nodes possibly helps improve the entire storage system performance, whereas it degrades the performance when files are rarely shared.

Aside from stripping, file sharing level remarkably affects the scalability of *ZonFS*. Overall, high file sharing degree hinders the scalability. In case of *low*, *ZonFS* scales up to 40 threads, which is the physical number of cores of the testbed. However, in *medium*, it fails to scale after eight threads, and for *high* it never shows any scalability. We also suspect that it was influenced by *atime* and reference count update overhead. Especially, the overhead of atomic reference count updates thoroughly throttles file system operations, leading low throughputs with 80 threads.



(a) Sharing *low* **(b)** Sharing *medium* **(c)** Sharing *high*

Fig. 10 Evaluation of file stripping with different sharing level

5 Related work

There have been several prior studies on the file system for non-volatile memory [22, 23]. Out of those, BPFS [22] is a file system designed 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 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 [24] is another memory file system for SCM connected to a memory bus. SCMFS is a file system which is developed using the Linux memory manager, and it suggests a simple file system structure. SCMFS uses the virtual memory page as a file system page, thus significant TLB miss overhead can occur, which leads to degraded overall file system performance. SCMFS also proposes a technique for partitioning memory into zones. Conquest [23] 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. Therefore, we have modified 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 and the SCM pages.

There also have been some NUMA-aware file systems. NOVA [25] is a hybrid memory file system under both SCM and DRAM. It allocates per-CPU data structures, which helps to acquire a degree of scalability. However, only per-CPU designs are not sufficient to fully exploit massively manycore NUMA machines. It is basically NUMA-oblivious because it never differentiates remote and local access. HydraFS [26] is a file system purposed to be optimized on NUMA machines. It adopts File-oriented Thread Binding (FTB), which schedules a thread to the core such that access of the thread optimally benefits from local access. Since, it does not consider memory controller and inter-node link congestion, it has potential to show suboptimal performance for overloaded workloads. Jericho [14] is a new I/O stack along with its own NUMA-aware file system. Especially, page cache is also replaced by JeriCache, its dedicated storage cache. Then, the cache is partitioned into slices, each of which mapped to a specific NUMA node. This helps to allocate the data to the optimal location by maximizing local access, and also gain high data-buffer affinity. The file system also dynamically migrates I/O threads to the node where the data resides, which helps to achieve data-thread affinity. However, this design requires rebuilding of the entire storage stack in Linux, which makes its general use less practical.

6 Conclusion and future work

In this paper, we have proposed a memory file system that uses partitioned Linux memory zones to efficiently utilize SCM as storage. We have implemented the proposed memory file system, *ZonFS* by extending Ramfs on Linux. For evaluation, we have used IOzone and our experimental results showed that the performance of initial write is improved up to 9.1 and 13.8%, as compared to Ramfs and Tmpfs, respectively and the read performance is gained up to 8%, when compared against Tmpfs. We have also evaluated the *ZonFS* for the NUMA manycore architectures and we have identified several factors that could cause to degrade the file system performance. Moreover, ensuring the file system consistency in case of power-off is an important issue in developing the SCM file system.

We have identified several future works. First, currently, *ZonFS* does not guarantee consistency after sudden power loss. We are considering the crash-consistency problem of *ZonFS* for the future work. Second, current Zone structure in Linux is locked for a memory request. Therefore, current ZonFS can suffer from Zone contention under simultaneous memory allocations. To this end, we are considering parallelizing memory allocations for a Zone using lock-free data structures as another future work.

Acknowledgements We thank Mr. Jae-Hoon Kim for his initial ZonFS implementation. We also thank Dr. Amol Jaikar for his careful proofreading and constructive comments that have significantly improved the paper. This work was supported by Institute for Information and Communications Technology Promotion (IITP) Grant funded by the Korea Government (MSIT) (No. 2014-0-00035, Research on High Performance and Scalable Many-Core Operating System).

References

1. Halupka, D., Huda, S., Song, W., Sheikholeslami, A., Tsunoda, K., Yoshida, C., Aoki, M.: Negative-resistance read and write schemes for STT-MRAM in 0.13 μm CMOS. In: 2010 IEEE International Solid-State Circuits Conference—(ISSCC), February 2010, pp. 256–257
2. Oh, H.R., Cho, B.H., Cho, W.Y., Kang, S., Choi, B.G., Kim, H.J., Kim, K., Kim, D., Kwak, C., Byun, H.G., Jeong, G.T., Jeong, H.: Enhanced write performance of a 64-Mb phase-change random access memory. IEEE J. Solid-State Circ. **41**(1),122–126 (2006)
3. Chen, Z., Wu, H., Gao, B., Yao, P., Li, X., Qian, H.: Neuromorphic computing based on resistive RAM. In: Proceedings of the on Great Lakes Symposium on VLSI 2017, GLSVLSI '17, New York, NY, USA, pp. 311–315. ACM (2017)

4. Intel: Revolutionizing Memory and Storage. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html
5. Dhiman, G., Ayoub, R., Rosing, T.: PDRAM: a hybrid PRAM and DRAM main memory system. In: Proceedings of the 46th Annual Design Automation Conference, DAC '09, New York, NY, USA, pp. 664–469. ACM (2009)
6. Kgil, T., Roberts, D., Mudge, T.: Improving NAND flash based disk caches. In: Proceedings of the 35th Annual International Symposium on Computer Architecture, ISCA '08, Washington, DC, USA, pp. 327–338. IEEE Computer Society (2008)
7. Qureshi, M.K., Srinivasan, V., Rivers, J.A.: Scalable high performance main memory system using phase-change memory technology. In: Proceedings of the 36th Annual International Symposium on Computer Architecture, ISCA '09, New York, NY, USA, pp. 24–33. ACM (2009)
8. Kim, J.W., Kim, J.-H., Khan, A., Kim, Y., Park, S.: ZonFS: a storage class memory file system with memory zone partitioning on Linux. In: 2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems (FAS* W), pp. 277–282. IEEE (2017)
9. Bovet, D., Cesati, M.: Understanding the Linux Kernel. O'Reilly and Associates, Inc., Sebastopol (2005)
10. Xu, J., Zhang, L., Memaripour, A., Gangadharaiah, A., Borase, A., Da Silva, T.B., Swanson, S., Rudoff, A.: NOVA-Fortis: a fault-tolerant non-volatile main memory file system. In: Proceedings of the 26th Symposium on Operating Systems Principles, pp. 478–496. ACM (2017)
11. Capps, D. IOzone Filesystem Benchmark. http://www.iozone.org
12. Natarajan, C., Christenson, B., Briggs, F.: A study of performance impact of memory controller features in multi-processor server environment. In: Proceedings of the 3rd Workshop on Memory Performance Issues: In Conjunction with the 31st International Symposium on Computer Architecture, pp. 80–87. ACM (2004)
13. Akram, S., Marazakis, M., Bilas, A.: NUMA implications for storage I/O throughput in modern servers. In: 3rd Workshop on Computer Architecture and Operating System Co-design (CAOS12) (2012)
14. Mavridis, S., Sfakianakis, Y., Papagiannis, A., Marazakis, M., Bilas, A.: Jericho: achieving scalability through optimal data placement on multicore systems. In: 2014 30th Symposium on Mass Storage Systems and Technologies (MSST), pp. 1–10. IEEE (2014)
15. Chandru, V., Mueller, F.: Reducing NoC and memory contention for manycores. In: International Conference on Architecture of Computing Systems, pp. 293–305. Springer (2016)
16. Intel: Intel Xeon Processor E5-4640 v2
17. Intel: Intel Memory Latency Checker v3.5. https://software.intel.com/en-us/articles/intelr-memory-latency-checker
18. Cheng, Y., Chen, W., Wang, Z., Yu, X.: Performance-monitoring-based traffic-aware virtual machine deployment on NUMA systems. IEEE Syst. J. **11**(2), 973–982 (2017)
19. Son, H., Lee, S., Won, Y.: Effect of timer interrupt interval on file system synchronization overhead. In: 2016 IEEE International Conference on Network Infrastructure and Digital Content (IC-NIDC), pp. 99–102. IEEE (2016)
20. Redhat: Configuring atime Update
21. Min, C., Kashyap, S., Maass, S., Kim, T.: Understanding manycore scalability of file systems. In: USENIX Annual Technical Conference, pp. 71–85 (2016)
22. Condit, J., Nightingale, E.B., Frost, C., Ipek, E., Lee, B., Burger, D., Coetzee, D.: Better I/O through byte-addressable, persistent memory. In: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, SOSP '09, New York, NY, USA, pp. 133–146. ACM (2009)
23. Wang, A.-I.A., Kuenning, G., Reiher, P., Popek, G.: The Conquest file system: better performance through a disk/persistent-RAM hybrid design. Trans. Storage **2**(3), 309–348 (2006)
24. Wu, X., Narasimha Reddy, A.L.: SCMFS: a file system for storage class memory. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC '11, New York, NY, USA, pp. 39:1–39:11. ACM (2011)
25. Xu, J., Swanson, S.: NOVA: a log-structured file system for hybrid volatile/non-volatile main memories. In: FAST, pp. 323–338 (2016)
26. Liu, Z., Sha, E.H.-M., Chen, X., Jiang, W., Zhuge, Q.: Performance optimization for in-memory file systems on NUMA machines. In: 2016 17th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT), pp. 7–12. IEEE (2016)

Jangwoong Kim is a Master Program Student in Sogang University, Seoul, South Korea. He received his B.S. degree in Computer Science from Sogang University. His research interests include parallel I/O and file system.

Youngjae Kim received his Ph.D. degree in Computer Science and Engineering from Pennsylvania State University, University Park, PA, USA in 2009. He is currently an Assistant Professor in the Department of Computer Science and Engineering at Sogang University, Seoul, Republic of Korea. Before joining Sogang University, Dr. Kim was a Staff Scientist in the US Department of Energy's Oak Ridge National Laboratory (2009–2015) and an Assistant Professor in Ajou University, Suwon, Republic of Korea (2015–2016). Dr. Kim received the B.S. Degree in Computer Science from Sogang University, Republic of Korea in 2001, and the M.S. Degree from KAIST in 2003. His research interests include distributed file and storage, parallel I/O, operating systems, emerging storage technologies, and performance evaluation.

Awais Khan is an Integrated Program Student in Sogang University, Seoul, South Korea. He received his B.S. degree in Bioinformatics from Mohammad Ali Jinnah University, Islamabad, Pakistan. He worked for one of leading Software Companies as a Software Engineer from 2012 to 2015. Currently, he is a Member in Laboratory for Advanced System Software at Sogang University Computer Science and Engineering Department. His research interests include cloud computing, cluster-scale deduplication, parallel and distributed file systems.

Sungyong Park is a Professor in the Department of Computer Science and Engineering at Sogang University, Seoul, Korea. He received his B.S. degree in Computer Science from Sogang University, and both the M.S. and Ph.D. degrees in Computer Science from Syracuse University. From 1987 to 1992, he worked for LG Electronics, Korea, as a Research Engineer. From 1998 to 1999, he was a Research Scientist at Telcordia Technologies (formerly Bellcore), where he developed network management software for optical switches. His research interests include cloud computing and systems, virtualization technologies, high performance I/O and storage systems, and embedded system software.