SCMFS : A File System for Storage Class Memory

Xiaojian Wu
Department of Electrical and
Computer Engineering, Texas A&M University
tristan.woo@neo.tamu.edu

A. L. Narasimha Reddy
Department of Electrical and
Computer Engineering, Texas A&M University
reddy@ece.tamu.edu

ABSTRACT

This paper considers the problem of how to implement a file system on Storage Class Memory (SCM),that is directly connected to the memory bus, byte addressable and is also non-volatile. In this paper, we propose a new file system, called SCMFS, which is implemented on the virtual address space. In SCMFS, we utilize the existing memory management module in the operating system to do the block management and keep the space always contiguous for each file. The simplicity of SCMFS not only makes it easy to implement, but also improves the performance. We have implemented a prototype in Linux and evaluated its performance through multiple benchmarks.

1. INTRODUCTION

In this paper, we focus on non-volatile memory which can be attached directly to the memory bus and is also byte addressable. Such nonvolatile memory can be used in the computer system for the main memory as well as for persistent storage of files. The promising nonvolatile memory technologies include Phase Change Memory(PCM)[21], memristor[24], and offer low latencies that are comparable to DRAM and are orders of magnitude faster than traditional disks.

The emerging of nonvolatile memory technologies bring many new opportunities for researchers. The emerging nonvolatile memory can be attached to memory bus, thus reducing the latencies to access persistent storage. These devices also enable processor to access persistent storage through memory load/store instructions enabling simpler and faster techniques for storing persistent data. However, compared to disk drives, these devices usually have much shorter write life cycles. A lot of work has been done on how to reduce write operations and to implement wear leveling on such devices [19, 28, 15, 14]. Since SCM's write endurance is usually 100-1000X+ order of NAND flash, the lifetime issues are expected to be less problematic. In this paper, we investigate how the characteristics of SCM devices should impact the design of file systems. SCM devices have very low access latency, which is much better than the existing other persistent storage devices and we consider them attached to the memory bus directly (as shown in Figure.1).

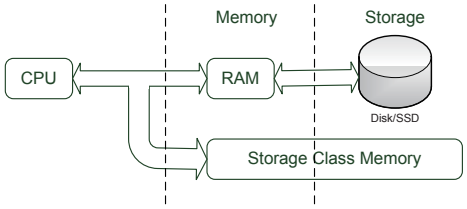


Figure 1: Storage class memory

To use SCM as a persistent storage device, the most straightforward way is to use RamDisk to emulate a disk device on the SCM device. Then it becomes possible to use a regular file system, such as Ext2Fs, Ext3Fs, etc.. The traditional file systems assume the underlying storage devices are I/O-bus attached block devices, and are not designed for memory devices. In this approach, the file systems access the storage devices through generic block layer and the emulated block I/O operations. The overhead caused by the emulation and the generic block layer is not necessary, since a file system specially designed for memory devices can be built on top of the memory access interface directly. In the traditional storage hierarchy, the additional overhead is ignorable since the latency to access storage devices is much higher than that to access memory. When the storage device is attached directly to the memory bus and can be accessed at memory speeds, these overheads can substantially impact performance and hence it is necessary to pay attention to avoid such overheads when ever possible. In addition, when storage devices are attached to the memory bus, both the storage device and the main memory will share system resources such as the bandwidth of the memory bus, the CPU cache and the TLB. In this case, the overhead of file systems will impact the performance of the whole system, and file systems for SCM should consider these factors. In our file system, we will eliminate unnecessary overheads in the hierarchy.

Another choice is to modify the existing memory based file systems, such as tmpfs [23], ramfs. These file systems are designed to use main memory to store the files, and are not for persistent storage devices. So, these file systems do not harden any data on persistent devices to let the system restore the data from rebooting. All the metadata are maintained by the in-memory data structures, and the file data are stored in the temporarily allocated memory blocks. It is

SC11, November 12-18, 2011, Seattle, Washington, USA

not harder to design a new file system from scratch than to adapt these file systems to SCM devices.

In this paper, we propose a new file system - SCMFS, which is specifically designed for SCM. With consideration of compatibility, this file system exports the identical interfaces as the regular file systems do, in order that all the existing applications can work on it. In this file system, we aim to minimize the CPU overhead of file system operations. We build our file system on virtual memory space and utilize the memory management unit (MMU) to map the file system address to physical addresses on SCM. The layouts in both physical and virtual address spaces are very simple. We also keep the space contiguous for each file in SCMFS to simplify the process of handling read/write requests in the file system. We will show, through results based on multiple benchmarks, that the simplicity of SCMFS makes it easy to implement and improves the performance. We focus on building a file system on a single processor in this paper. Our approach can be extended to larger, multiprocessor systems through distributed shared memory.

The paper makes the following significant contributions: (a) proposes a new file system for Storage Class Memory, (b) presents the details of a prototype implementation on Linux and (c) evaluates its performance by using both micro benchmarks and application level benchmarks.

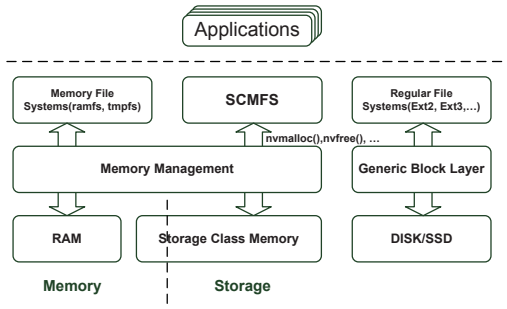


Figure 2: File systems in operating systems.

The remainder of this paper is organized as follows. Section 2 gives details of the related work. Section 3 describes the design of SCMFS and our prototype implementation. Section 4 presents the results of the evaluation. In section 5, we discuss about some limitations of SCMFS and present the future work. Section 6 concludes the paper.

2. BACKGROUND AND RELATED WORK

A number of file systems have been designed for flash devices([1, 26]). BPFS [7] is proposed as a file system designed for non-volatile byte-addressable memory, which uses shadow paging techniques to provide fast and consistent updates. It also requires architectural enhancements to provide new interfaces for enforcing a flexible level of write ordering. Our file system aims to simplify the design and eliminate the unnecessary overhead to improve the performance. Since there is no kernel level implementation of BPFS, we did not compare our performance to BPFS. DFS[10] is the most similar file system to our file system. DFS incorporates the functionality of block management in the device driver and firmware to simplify the file system, and also keeps the files contiguous in a huge address space. It is designed for a PCIe based SSD device by FusionIo, and relies on specific features in the hardware.

A number of projects have previously built storage systems on non-volatile memory devices. Rio[6] and Conquest [9] use a battery-backed RAM in the storage system to improve the performance or provide protections. Rio uses the battery-backed RAM to store the file cache to avoid flushing dirty data, while Conquest uses it to store the file system metadata and small files. In the eNVy storage system [27], the flash memory is attached to the memory bus to implement a non-volatile memory device. To make this device byte addressable they designed a special controller with a battery-backed RAM buffer. Our work assumes that non-volatile memory is large enough for both data and metadata and focuses on leveraging memory management infrastructure in the system. A data structure level approach to achieve data consistency on non-volatile memory is described in [25].

Solutions have been proposed to speed up memory access operations, to reduce writes, and for wear-leveling on PCM devices. Some of these solutions improve the lifetime or the performance of PCM devices at the hardware level [12, 13]. Some of them use a DRAM device as a cache of PCM in the hierarchy. [20] presents a page placement policy on memory controller to implement PCM-DRAM hybrid memory systems. Several wear-leveling schemes to protect PCM devices from normal applications and even malicious attacks have been proposed [15, 16, 22, 28]. Since our work focuses on the file system layer, all the hardware techniques can be integrated with our file system to provide better performance or stronger protection.

Data deduplication based approaches have been proposed for reducing writes to SSDs at the device level [5, 8, 18] and for employing SSDs in deduplication file systems [4].

3. SCMFS

In this section, we present the design of SCMFS and a prototype implementation on Linux 2.6.33 kernel.

3.1 Design

In this work, we aim to design a file system for SCM devices. With traditional persistent storage devices, the overhead brought by I/O latency is much higher than that of file system layer itself. So the storage system performance usually depends on the devices' characteristics and the performance of I/O scheduler. However, in the case that storage device is directly attached to the memory bus, the storage device will share some critical system resources with the main memory. They will share the bandwidth of the memory bus, the CPU caches and TLBs for both instruction and data. We believe that the lower complexity of the file system can reduce the CPU overhead in the storage system and then improve the total performance. Our design is motivated by the need to minimize the number of operations required to carry out file system requests, in such a system.

3.1.1 Reuse Memory Management

Current file systems spend considerable complexity due to space management. For example, Ext2fs spends almost 2000 SLOCs (source lines of code) on it. Since SCM will be visible through memory bus, it is possible to reuse the memory management module within the operating system to carry out these functions. Memory management has hardware support in the form of TLB and MMU caches to speed

up operations of translating from virtual addresses to physical addresses, providing protection mechanisms across users etc. It seems natural to exploit this infrastructure to speed up file system operations as well when storage will be accessible through memory bus. SCMFS is designed to reuse the memory management infrastructure, both in the hardware and the Operating System. It is expected that such a design would benefit from the future enhancements to memory management infrastructure within the processor, through increased TLB sizes and MMU caches.

In our design, we assume the storage device, SCM, is directly attached to CPU, and there is a way for firmware/software to distinguish SCM from the other volatile memories. This assumption allows the file systems be able to access the data on SCM in the same way as normal RAM. With this assumption, we can utilize the existing memory management module in the operating system to manage the space on the storage class memory. As shown in Figure. 2, regular file systems are built on top of the generic block layer, while SCMFS is on top of the modified memory management module.

When the file system relies on the MMU for mapping virtual addresses to physical addresses, these mappings need to be persistent across reboots in order to access the data after a power failure, for example. It is not sufficient to allocate the page mapping table on the SCM since these mappings can be cached at various locations before being written to memory. We need to immediately harden the address mappings whenever space is allocated on SCM. We made enhancements to the kernel for this purpose, as explained later in Section 3.4.

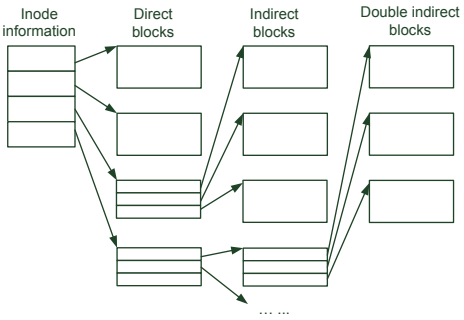


Figure 3: Indirect block mechanism in Ext2fs

3.1.2 Contiguous File Addresses

Current file systems employ a number of data structures to manage and keep track of the space allocated to a file. The file systems have to deal with the situation that a large file is split into several parts and stored in separate locations on the block device. For example, Ext2fs handles this by using indirect blocks, as shown in Figure.3. This makes the process of handling the read/write requests much more complicated, and sometimes requires extra read operations of the indirect blocks.

In order to simplify these data structures, we design the file system such that the logical address space is contiguous within each file. To achieve this, we build the file system on virtual address space, which can be larger than the physical address space of the SCM. We can use page mapping to keep all the blocks within a file to have contiguous logical addresses. In SCMFS, with the contiguity inside each file, we do not need complicated data structures to keep track of logical address space, and simply store the start address and the size for each file. This mechanism significantly simplifies the process of the read/write requests. To get the location of the request data, the only calculation is adding the offset to the start address of the file. The actual physical location of the data is available through the page mapping data structures, again leveraging the system infrastructure.

As described above, putting the file system on virtual address space can simplify the design and reduce overheads. However, it also has a side affect that it may cause more TLB misses. Operating systems sometimes map the whole memory in the system to a linear address space using a larger page size (e.g., 2MB), resulting in smaller number of TLB misses. In our current implementation, to minimize the internal fragmentation we use a page size of 4K bytes. Hence, we may incur more TLB misses than if we were to employ linear mapping of the virtual address space corresponding to the file system. We will see its impacts in Section 4.

3.2 File System Layout

Figure. 4 shows the layout of both virtual memory space and physical memory space in SCMFS. The "metadata" in physical memory space contains the information of storage, such as size of physical SCM, size of mapping table, etc. The second part of the physical memory is the memory mapping table. The file system needs this information when mounted to build some in-memory data structures, which are mostly maintained by memory management module during runtime. Any modification to these data structures will be flushed back into this region immediately. Since the mapping information is very critical to the file system consistency, all the updates to this region will be flushed immediately by using the procedure "clflush_cache_range" described in Section 3.6. The rest of the physical space is mapped into virtual memory space and used to store the whole file system.

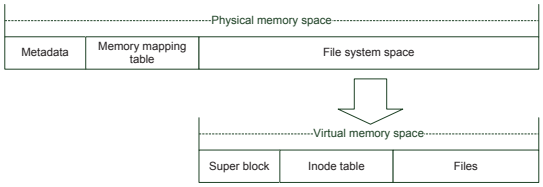


Figure 4: Memory space layout

In the virtual memory space, the layout of SCMFS is very simple and similar to existing file systems, and it consists of three parts. The first part, as in the regular file systems, is the super block, which contains the information about the whole filesystem, such as file system magic number, version number, block size, counters for inodes and blocks, the total number of inodes and blocks, etc.. The second part is the inode table, which contains the fundamental information of each file or directory, such as file mode, file name, owner id, group id, file size in bytes, the last time the file was accessed (atime), the last time the file was modified (mtime), the time the file was created (ctime), start address of the data for the file, etc.. The first item(with inode number 0) in the inode table is the root inode that is always a directory. All the content of the files in the file system are stored in the third part. In our prototype, the total size of virtual memory space for

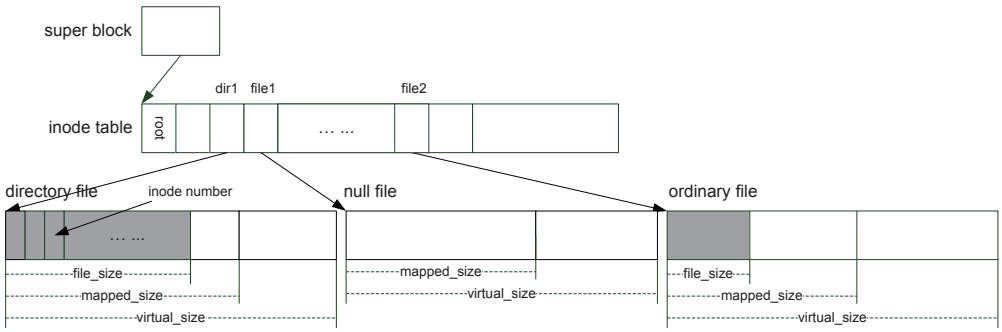


Figure 5: SCM file system Layout.

SCMFS is 2^{47} bytes (range: ffff000000000000 - ffff7fffffffffff), which is unused in original Linux kernel. The space can be larger if we re-organize 64-bit virtual address space. The structure of SCM file system is illustrated in Figure.5. In SCM file system, directory files are stored as ordinary files, except that their contents are lists of inode numbers. Besides ordinary files and directory files, in SCMFS, there is an additional type of file, null file, which will be described in 3.3. There is also a pointer to the start address of inode table in the super block. In the inode table, we use a fixed size of entry, which is 256 bytes, for each inode and it is very easy to get a file's metadata through its inode number and the start address of the inode table.

With the layouts, the file system can be easily recovered or restored after rebooting. First we check if the "metadata" is valid through the magic number and version number, and use the information in the "metadata" and "mapping table" to build the mapping between the physical addresses and the virtual addresses. Once we finish this, we can get the information about the file system from the super block in the virtual address space. It is noted that both the physical and the virtual address in the mapping table need to be relative instead of absolute to provide the portability between different machines and systems.

3.3 Space Pre-Allocation

In regular file systems, the data blocks are allocated on demand. The space is allocated to the files only when needed, and once any file is removed, the space allocated for it will be deallocated immediately. Frequent allocation and deallocation can invoke many memory management functions and can potentially reduce performance. To avoid this, we adopted a space pre-allocation mechanism, in which we create and always maintain certain amount of null files within the file system. These null files have no name, no data, however have already been allocated some physical space. When we need to create a new file, we always try to find a null file first. When a file shrinks, we will not de-allocate the unused space. And when we need to delete an existing file, we will not de-allocate its space but mark it as a null file. Through the space pre-allocation mechanism, we can reduce the number of allocation and deallocation operations significantly, and expect to boost the file system performance.

To support this mechanism, we need to maintain three "size"s for each file. The first one, "file_size" , is the actual size of the file. The second one, "virtual_size" is the size of the virtual space allocated to the file. The last one, "mapped_size", is the size of mapped virtual space for the file, which is also the size of physical space allocated to the file. The value of "virtual_size" is always larger than or equal to that of "mapped_size", whose value is always larger than or equal to that of "file_size".

The space unused but mapped for each file is reserved for later data allocations, and potentially improves the performance of further writing performance. However, these spaces are also likely to be wasted. To recycle these "wasted" spaces, we use a background process. This method is very similar to the garbage collection mechanism for flash based file systems. This background thread will deallocate the unused but mapped spaces for the files when the utilization of the SCM reaches a programmable threshold, and it always chooses cold files first.

3.4 Modifications to kernel

In our prototype, we have done some modifications to original Linux kernel 2.6.33 to support our functionalities. First, we modify the E820 table, which is generated by BIOS to report the memory map to the operating system[2]. We added a new address range type "AddressRangeStorage". This type of address range should only contain memory that is used to store non-volatile data. By definition, the operating system can use this type of address range as storage device only. This modification make sure the operating system has the ability to distinguish SCM from normal memory device.

Second, we add a new memory zone "ZONE_STORAGE" into the kernel. A memory zone in linux is composed of page frames or physical pages, and a page frame is allocated from a particular memory zone. There are three memory zones in original Linux: ZONE_DMA is used for DMA pages, "ZONE_NORMAL" is used for normal pages , and "ZONE_HIGHMEM" is used for those addresses that can not be contained in the virtual address space(32bit platform only). We put all the address range with type "AddressRangeStorage" into the new zone "ZONE_STORAGE".

Third, we add a set of memory allocation/deallocation functions, nvmalloc()/nvfree(), which allocate/deallocate memory from the zone "ZONE_STORAGE". The function nvmalloc() derives from vmalloc(), and allocates memory which is contiguous in kernel virtual memory space, while not necessary to be contiguous in physical memory space. The function nvmalloc() has three input parameters: size is the size of virtual space to reserve, mapped_size is the size of virtual space to map, write_through is used to specify if the cache policy for the allocated space is write-through or write-back. We also have some other functions, such as nvmalloc_expand() and nvmalloc_shrink(), whose parameters are

same as that of nvmalloc(). The function nvmalloc_expand() is used when the file size increases and the mapped space is not enough, and nvmalloc_shrink() is used to recycle the allocated but unused space.

All the modifications involve less than 300 lines of source code in kernel.

3.5 Garbage Collection

As described above, the mechanism of pre-allocation is used to improve the speed of appending data to files. However it can result in wasting space when we pre-allocate space for files and may not write anymore data to them. To recycle the wasted space, we provide a garbage collection mechanism. Using a garbage collection in a file system is normal, especially for the flash file systems. To minimize its impact on the system performance, we implemented this mechanism in a background kernel thread. When the unmapped space on the SCM is lower than a threshold, this background thread will try to free the unnecessary space, that is mapped but not used. During the garbage collection, it will check the number of null files first. If the number exceeds a predefined threshold, it will free the extra null files. If we need to free more, this thread will consider the cold files first, that have not been modified for a long time, then the hot files. We can easily classify the cold/hot file through the last modified time. This thread also takes the responsibility of creating null files when there are too few null files in the system.

Even though our current system doesn't implement any wear leveling functions, we expect to incorporate wear leveling into a background process that can work with the garbage collection thread.

3.6 File System Consistency

File system consistency is always a big issue in file system design. As a memory based file system, SCMFS has a new issue: unsure write ordering. The write ordering problem is caused by CPU caches that stand between CPUs and memories [7]. Caches are designed to reduce the average access latency to memories. To make the access latency as close to that of the cache, the cache policy tries to keep the most recently accessed data in the cache. The data in the cache is flushed back into the memory according to the designed data replacement algorithm. And the order in which data is flushed back to the memory is not necessarily the same as the order data was written into cache. Another reason that causes unsure write ordering is out-of-order execution of the instructions in the modern processors. To address the problem of unsure write ordering, we can use a combination of the instructions MFENCE and CLFLUSH. This combination has been implemented with the function "clflush_cache_range" and used in the original Linux kernel. The instruction MFENCE is used to serialize all the load/store instructions that were issued prior to the MFENCE instruction. This instruction guarantees that every load/store instruction that precedes it is globally visible before any load or store instruction that follows it. The instruction CLFLUSH is used to invalidate the cache line that contains the specified address from all levels of the processor cache hierarchy. By using the function "clflush_cache_range", we can provide the ensured write order to any range of addresses. Our design doesn't assume the availability of any hardware or architectural mechanisms (such as epochs in BPFS[7]) beyond what is currently available.

In SCMFS, we always use the function "clflush_cache_range" when we need to modify the critical information, including "metadata", "superblock", "inode table" and "directory files". This simple mechanism will provide metadata consistency. As to the data consistency, we flush the CPU cache periodically. This provides similar guarantees as the existing regular file systems.

4. EVALUATION

To evaluate our ideas, we have implemented a prototype of SCMFS in Linux. This prototype consists of about 2700 source lines of code, which is only 1/10 of that of ext2fs in Linux. In this section, we present the results by using some standard benchmarks.

4.1 Bechmarks and Testbed

To evaluate SCMFS thoroughly, we use multiple benchmarks. The first benchmark, IOZONE [3], is a synthetic workload generator. This benchmark creates a large file, and issues different kinds of read/write requests on this file. Since the file is only opened once in each test, we use the benchmark IOZONE to evaluate the performance of accessing file data. The second benchmark, postmark [11] is an I/O intensive benchmark designed to simulate the operation of an e-mail server. This benchmark creates a lot of small files and performs read/write operations on them. We use this benchmark to evaluate SCMFS's performance on small files and metadata. In the experimental environment, the test machine is a commodity PC system equipped with a 2.33GHz Intel Core2 Quad Processor Q8200, 8GB of main memory. We configured 4GB of the memory as the type "AddressRangeStorage", and used it as Storage Class Memory. The operating system used is Fedora 9 with a 2.6.33 kernel.

In all the benchmarks, we compare the performance of SCMFS to that of other existing file systems, including ramfs, tmpfs and ext2fs. Since ext2fs is designed for a traditional storage device, we run ext2fs on ramdisk, which emulates a disk drive by using the normal RAM in main memory. It is noted that ramfs, tmpfs and ramdisk are not designed for persistent memory, and none of them can be used on storage class memory directly.

4.2 IOZONE Results

Using IOZONE, we have evaluated the sequential and random performance, and the results are shown in the Figure.6 (a,b) and Figure.7(a,b) respectively. We also used the performance counters in the modern processors, through the PAPI library [17], to see the detailed performance information related to CPU's functional units, including L1/L2 cache miss rate, Data/Instruction TLB misses. We show this information in the rest of Figure.6 and Figure.7.

In these figures, we can see that the performances of all the file systems decreases dramatically when the record length is more than 1 megabytes. This is because that when record length is too large, L2 cache miss rate and Data TLB misses increases significantly, as shown in the Figure.6(g,h,i,j) and Figure. 7(g,h,i,j).

We have noticed that the memory based file systems, including ramFs, tmpFs and SCMFS, performed generally much better than Ext2Fs on Ramdisk. For example, performance is doubled in all the workloads with the record

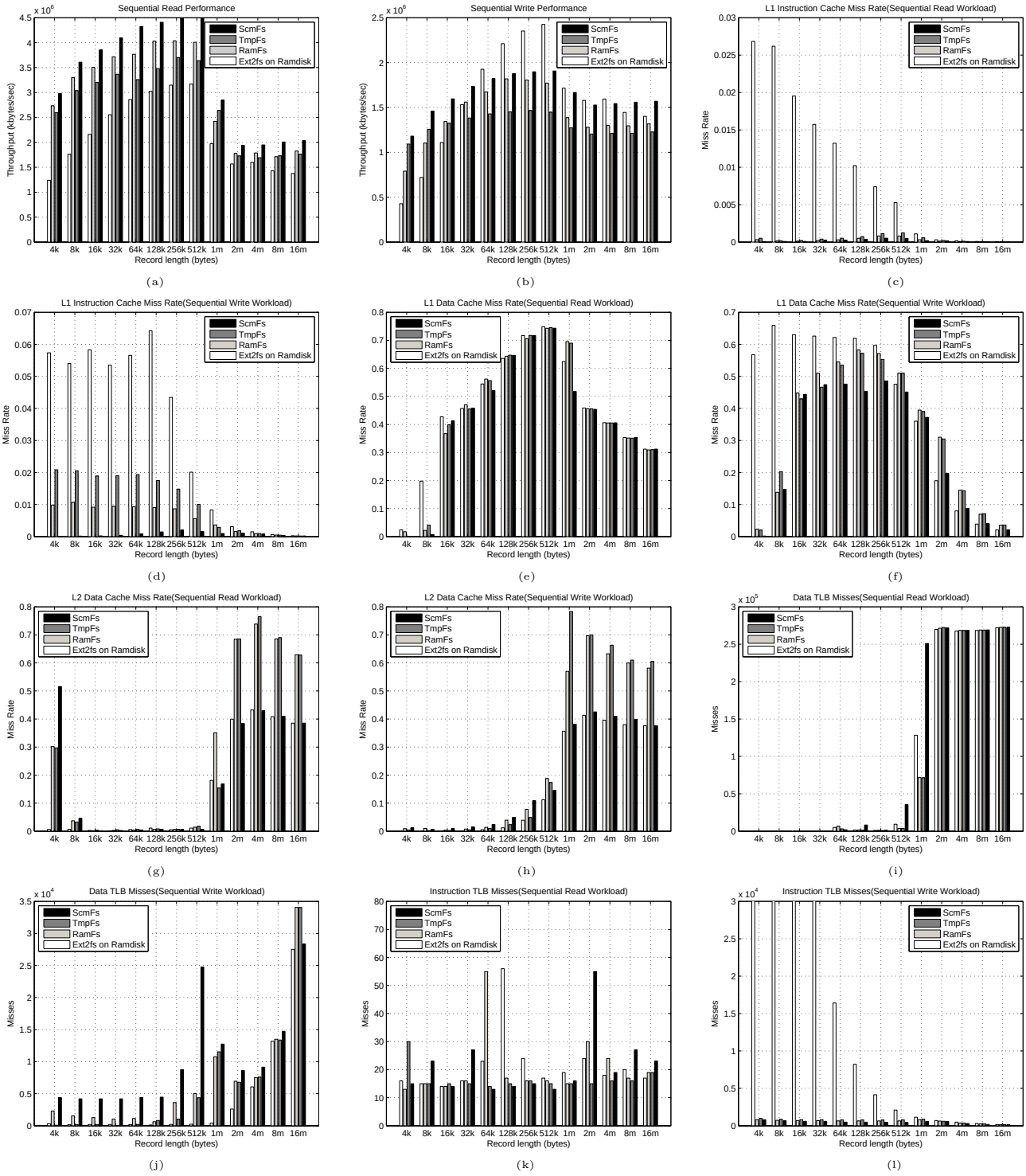


Figure 6: IOZONE results(Sequential workloads)

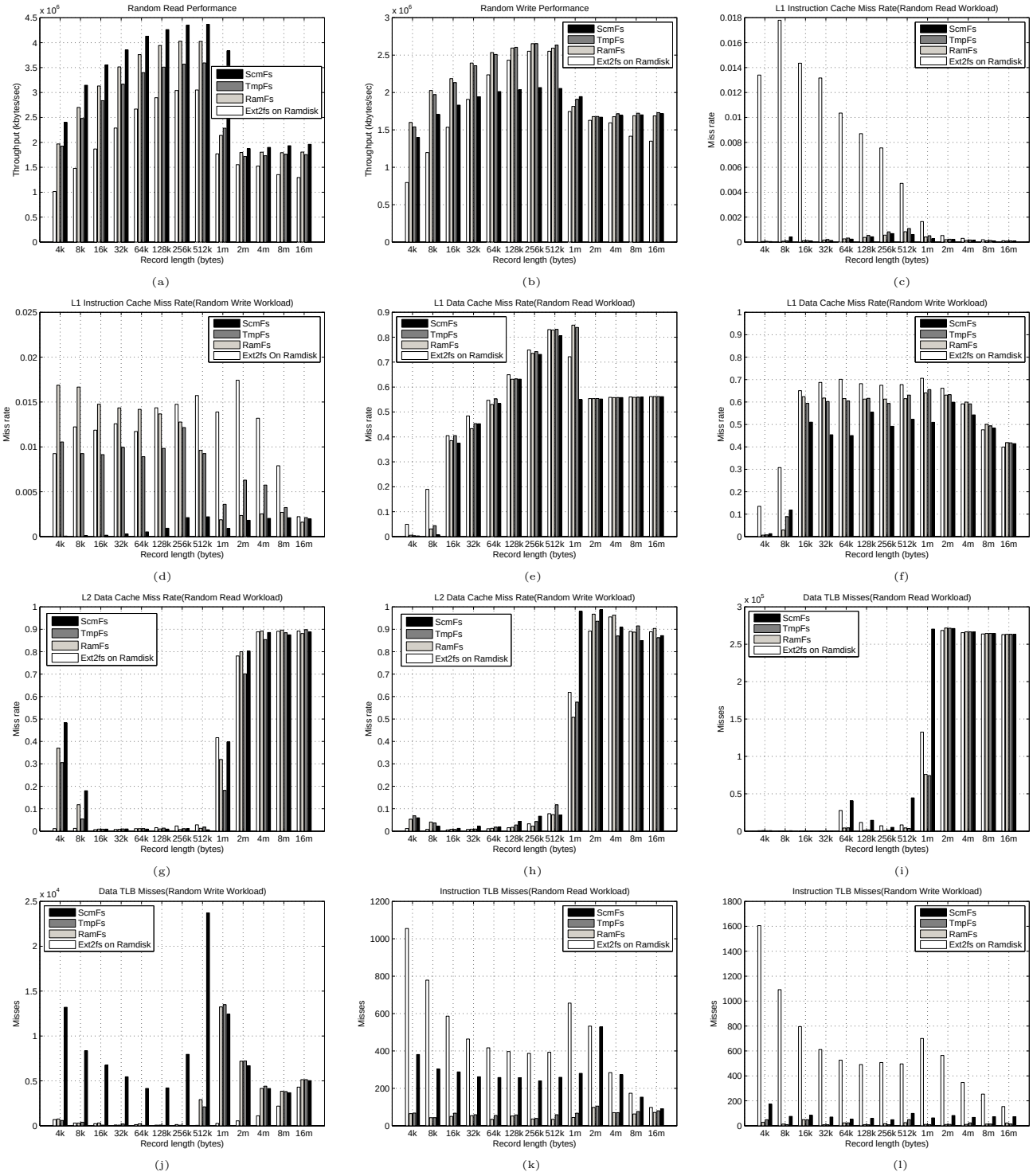
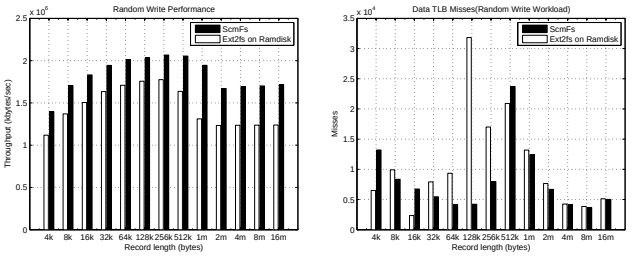


Figure 7: IOZONE results(Random workloads)

lenth 4k bytes in these file systems compared to Ext2Fs on Ramdisk. The reason is that Ext2 file system is created on the generic block layer and has much higher complexity than the memory based file systems, as we describe in Section 3. Simplicity of the hierarchy significantly decreases the size of instruction set. As shown in Figure.6(c,d,k,l) and Figure.7(c,d,k,l), memory based file systems have much lower instruction cache miss rate and instruction TLB misses than Ext2 file system. For example, in the random read workload with 4k bytes record length, the L1 instruction cache miss rate is 0.0134 in Ext2Fs and 0.000042 in SCMFS while in the random write workload with 4k bytes record length, it is 0.0093 in Ext2Fs and 0.000061 in SCMFS. Compared to tmpfs and ramfs, SCMFS also has much lower instruction cache miss rate, especially in write workload. For example, in the random write workload with 4k bytes record length, the L1 instruction cache miss rate is 0.0169 in ramfs, 0.0105 in tmpfs, and 0.000061 in SCMFS. In both sequential and random workloads, SCMFS shows significant advantages on instruction cache misses, which makes SCMFS performs beyond the other file system in most workloads.

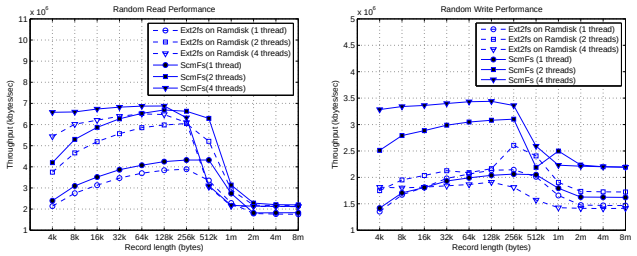
We also notice that in the random/sequential write workload, Ext2 file system performs better than SCMFS when the record length was between 64k and 512k bytes. We believe this is because SCMFS has much higher TLB misses than Ext2 file system, as shown in Figure. 6(j) and Figure. 7(j). The reason why SCMFS has much more TLB misses than the other file systems is we operate the data in SCMFS on virtual address space while the others employ device level linear address space. Modern processors usually support a feature, Page Size Extension (PSE) that allows for the pages with larger size than the traditional 4KB. In our environment, the address space SCMFS resides in is mapped by using 4 KB pages while the linear address space by 2 MB pages. To confirm it is the large page size that reduces data TLB misses in Ext2 file system, we ran the same workload again with PSE disabled. In the results in Figure. 8, we can see that, without PSE, the TLB misses in the both file systems are comparable and SMCMFS performs better than Ext2 file system with all the request sizes.



(a) (b)

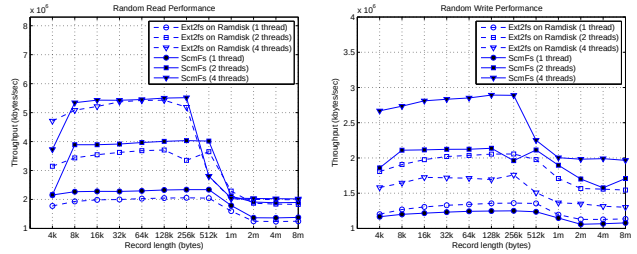
Figure 8: IoZone results with PSE disabled(random workload).

We also have done experiments with IOZONE by using multiple threads. Figure.9 shows the result where we use normal read()/write() interfaces, while Figure.10 shows the results with mmap() interfaces. When we use map() interface, we enable XIP features for both the file systems. We can see, in both cases SCMFS performs better than Ext2Fs and obtains higher throughput with more threads. Another observation is that using mmap()/bcopy() does not perform beyond normal read()/write() interfaces. Through our investigation, we believe this is also caused by high TLB misses. In Ext2fs on Ramdisk, using mmap() will map the address into user address space, which is not using large page size. By using performance counters, we find that the number of TLB misses with read()/write() interfaces is only around 200(Ext2fs) or 4,000(SCMFS), while it is more than 2,000,000 with mmap() interfaces in the same workload.



(a) (b)

Figure 9: Iozone results with multi-thread (Random workload)



(a) (b)

Figure 10: IOZONE results with multi-thread, using mmap (Random workload)

It is observed from Figure. 9 that SCMFS obtains up to 7GB/s read throughput, about 70% of the memory bus bandwidth of 10GB/s on our system. It is observed that the read throughput generally saturates at twice the saturation throughput of writes, since writes require two memory operations compared to one operation on read requests in the IOZONE benchmark.

4.3 Postmark Results

We show the results of postmark in Figure.11. We use postmark to generate both read intensive and write intensive workloads. The file size in the workload is varied between 4k and 40k bytes. In each workload, we created 10,000 files under 100 directories and performed 400,000 transactions. We again used the PAPI library to investigate the detailed performance information.

In this test, we not only have evaluated original SCMFS, but also SCMFS with pre-allocation mechanism, as described in 3.3, and with file system consistency, as described in 3.6. We do not include these with the IOZONE workload, since IOZONE workload operates on one file and does not exhibit much difference in performance with these mechanisms.

In the figures, we can see that the performance of all the file systems is close to each other in the Postmark workload. Postmark workload has many more metadata operations than the IOZONE workload and hence these metadata operations dominate the file sysem performance. Since the files in the Postmark workload are small, the possibility to

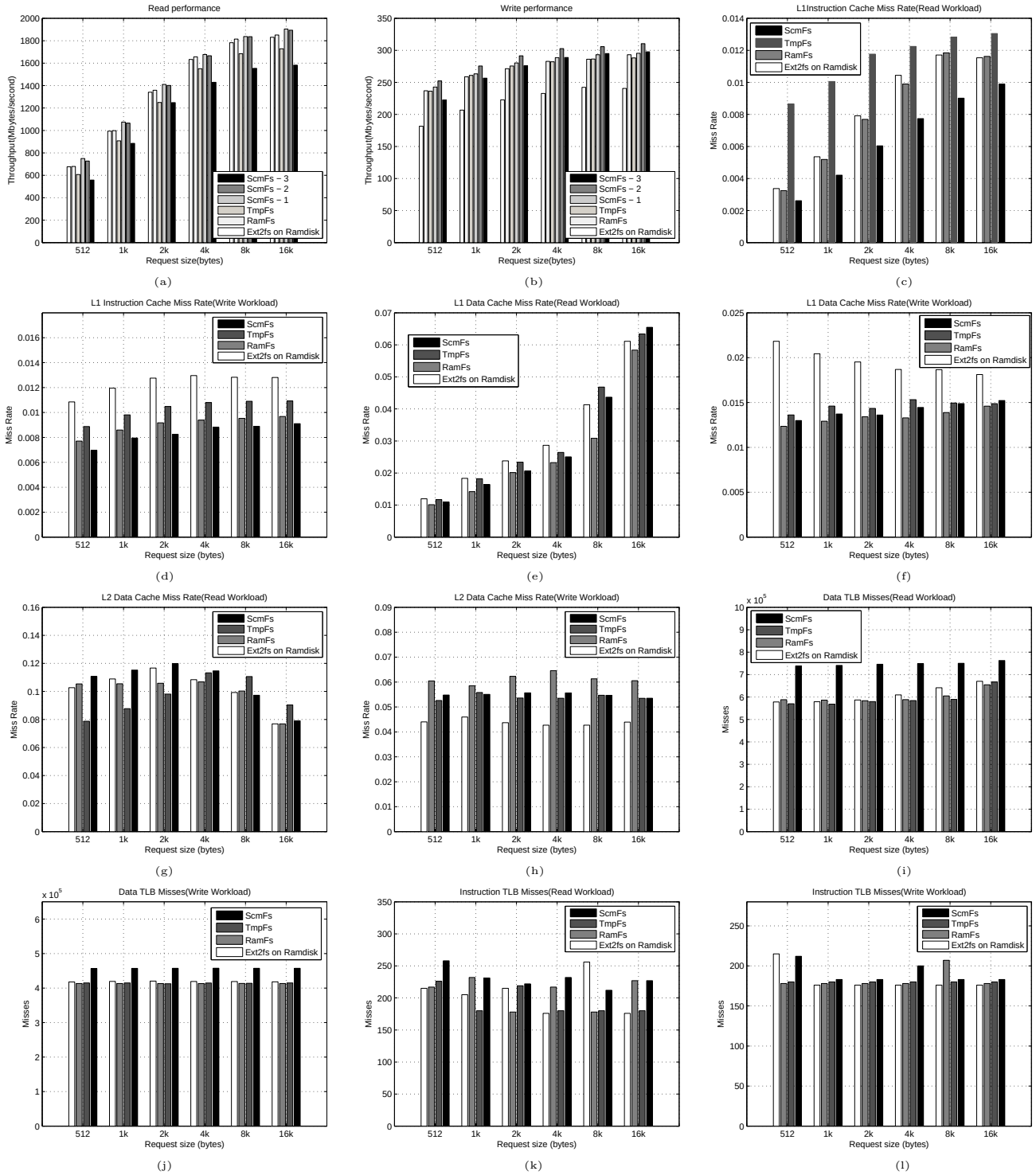


Figure 11: Postmark results. SCMFS-1, original SCMFS. SCMFS-2, SCMFS-1 with space pre-allocation. SCMFS-3, SCMFS-2 with file system consistency.

use indirect blocks in Ext2fs is very low, and SCMFS doesn't have much advantage over Ext2fs. Through the results, we can see that SCMFS still has lower instruction cache miss rate than Ext2fs, especially in the write workload. Even though SCMFS has higher data TLB misses, SCMFS provides higher performance beyond ext2fs.

When we add the pre-allocation mechanism, the read performance of SCMFS drops slightly and the write performance improves. The reason why the read performance drops is that the file system allocates more space than needed and the benchmark spend more time on initialize the files. As we describe in Section 3.3, the pre-allocation mechanism helps reduce the time to allocate space for new data when appending the files. It improves the write performance by about 5% as shown in Figure.11(b). In the last configuration, we add the support of file system consistency that is described in Section 3.6. As anticipated, the performance of SCMFS drops significantly when write ordering issues are addressed. In the write workload, SCMFS still performs better than Ext2fs by about 22%. In the read workload, even though the content of the files are not changed, the latest access time of each file needs to be updated. Each time the file metadata gets udpated, the costly function "clflush_cache_range" is called to flush the cache. That is why the read performance decreases significantly. It is noted that Ext2fs on ramdisk does not support metadata consistency as SCMFS does.

In the Postmark workload, the saturation throughputs are lower than observed earlier with the IOZONE workload, because of higher metadata operations involved in the Postmark workload. It is observed that the TLB misses are significantly higher in the Postmark workload compared to the IOZONE workload.

5. DISCUSSION AND FUTURE WORK

In this research work, we have implemented a file system designed for Storage Class Memory and showed some advantages compared to existing file systems. However, this file system has some disadvantages and limits, and we will consider them in our future work.

In our current experiment environment, the size of simulated SCM is very small(4GB), so the size of required mapping table is also very small. The size of mapping table will become very large when the SCM is scaled to tens or even hundreds of Gigabytes. The large mapping table will significantly increase the time to mount the entire file system. To address this problem, we can delay the memory mapping process, which means only the virtual address space for metadata and inode table will be mapped during the time of mounting the file system. All the other address spaces will be mapped in background after the file system is mounted. If a request to an unmapped file is received, a page fault will be triggered. In the page fault handler, we will read the SCM mapping table and map the address. To achieve this, we also need to maintain bitmaps (or other compressed data structures) for MM to indicate those physical addresses and virtual addresses that are already used. The bitmaps are also loaded to MM module during the mount procedure. Another potential issue the large scale of SCM may cause is that the TLB cannot cover enough range of memory and results in many TLB misses. We may use superpages to increase the coverage of TLB and decrease the TLB misses that require expensive address translations.

In current implementation, we reserve a large virtual space for SCMFS and did not consider the extreme case of fragmentation, in which there is enough physical space but there is no contiguous virtual space for a new file. In the future work, we consider to add defragmentation of virtual address space into the thread of garbage collection.

Most SCM technologies have limits on write cycles to individual memory locations. In our current work, we did not incorporate any algorithms for wear leveling of the underlying SCM. We plan to include this as part of allocation process in the future.

6. CONCLUSION

In this paper, we have presented the design of SCMFS, a new file system specially for the storage class memory. SCMFS utilizes the existing memory management module in the operating system to help the block management, and keeps the space for each file always contiguous in the virtual address space. The design of SCMFS simplifies its implementation, and improves the performance, especially for small size requests. We also have discussed the limits of SCMFS and our future work.

7. ACKNOWLEDGMENTS

This work is supported in part by NSF grants 0702012 and 0621410.

8. REFERENCES

[1] Yaffs. Available on July 2011 from http://www.yaffs.net/.

[2] *Advanced Configuration and Power Interface Specification 3.0*. 2004. Available on July 2011 from http://www.acpi.info/.

[3] Iozone file system benchmark. Available on July 2011 from http://www.iozone.org/, 2011.

[4] S. S. Biplob Debnath and J. Li. ChunkStash: Speeding up Inline Storage Deduplication using Flash Memory. In *Proc. of the USENIX Annual Technical Conference*, ATC'10, 2010.

[5] F. Chen, T. Luo, , and X. Zhang. CAFTL: A Content-Aware Flash Translation Layer Enhancing the Lifespan of Flash Memory based Solid State Drives. In *Proc. of the 9th USENIX Conference on File and Storage Technologies*, FAST'11, 2011.

[6] P. M. Chen, W. T. Ng, S. Chandra, C. Aycock, G. Rajamani, and D. Lowell. The rio file cache: Surviving operating system crashes. In *Proc. of the Architectural Support for Programming Languages and Operating Systems*, pages 74–83, 1996.

[7] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. C. Lee, D. Burger, and D. Coetzee. Better i/o through byte-addressable, persistent memory. In *Proc. of the Symposium on Operating Systems Principles*, pages 133–146, 2009.

[8] A. Gupta, R. Pisolkar, B. Urgaonkar, and A. Sivasubramaniam. Leveraging Value Locality in Optimizing NAND Flash-based SSDs. In *Proc. of the 9th USENIX Conference on File and Storage Technologies*, FAST'11, 2011.

[9] A. i A. Wang, P. Reiher, G. J. Popek, and G. H. Kuenning. Conquest: Better performance through a

disk/persistent-ram hybrid file system. In *Proc. of the 2002 USENIX Annual Technical Conference*, pages 15–28, 2002.

[10] W. K. Josephson, L. A. Bongo, D. Flynn, and K. Li. Dfs: A file system for virtualized flash storage. In *Proc. of the USENIX Conference on File and Storage Technologies*, volume 6, pages 85–100, 2010.

[11] J. Katcher. Postmark: A new file system benchmark. Technical Report TR3022. Network Applicance Inc. October 1997.

[12] B. Lee, P. Zhou, J. Yang, Y. Zhang, B. Zhao, E. Ipek, O. Mutlu, and D. Burger. Phase-Change Technology and the Future of Main Memory. *IEEE Micro*, 30(1):143 –143, Jan 2010.

[13] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting Phase Change Memory as a Scalable Dram Alternative. In *Proc. of the 36th Annual IEEE/ACM International Symposium on Computer Architecture*, pages 2–13, 2009.

[14] M.Qureshi, M. Franceschini, and L. Lastras-Montano. Improving read performance of phase change memories via write cancellation and write pausing. In *Proc. of the 16th IEEE International Symposium on High Performance Computer Architecture*, pages 1–11, 2010.

[15] M.Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali. Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling. In *Proc. of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 14–23, 2009.

[16] M.Qureshi, A. Seznec, L. Lastras, and M. Franceschini. Practical and secure pcm systems by online detection of malicious write streams. In *Proc. of the 17th IEEE International Symposium on High Performance Computer Architecture*, pages 478–489, 2011.

[17] P. J. Mucci, S. Browne, C. Deane, and G. Ho. Papi: A portable interface to hardware performance counters. In *Proc. of the Department of Defense HPCMP Users Group Conference*, pages 7–10, 1999.

[18] X. Ouyang, D. Nellans, R. Wipfel, D. Flynn, and D. K. Panda. Beyond Block I/O: Rethinking Traditional Storage Primitives. In *Proc. of the 17th IEEE International Symposium on High Performance Computer Architecture*, pages 301–311, 2011.

[19] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *Proc. of the International Symposium on Computer Architecture*, pages 24–33, 2009.

[20] L. E. Ramos, E. Gorbatov, and R. Bianchini. Page placement in hybrid memory systems. In *Proceedings of the international conference on Supercomputing*, ICS '11, pages 85–95, New York, NY, USA, 2011. ACM.

[21] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y. c. Chen, R. M. Shelby, M. Salinga, D. Krebs, S. h. Chen, H. l. Lung, and C. H. Lam. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development*, 52:465–480, 2008.

[22] N. H. Seong, D. H. Woo, and H.-H. S. Lee. Security refresh: Prevent malicious wear-out and increase durability for phase-change memory with dynamically randomized address mapping. In *Proc. of the 37th Annual IEEE/ACM International Symposium on Computer Architecture*, pages 383–394, 2010.

[23] P. Snyder. Tmpfs: A virtual memory file system. In *Proc. of the Autumn 1990 European UNIX Users' Group Conference*, pages 241–248, 1990.

[24] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams. The missing memristor found. *Nature*, 453(7191):80–83, May 2008.

[25] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proc. of the 9th Usenix Conference on File and Storage Technologies (FAST)*, pages 61–76, Feb 2011.

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

[27] M. Wu and W. Zwaenepoel. envy: a non-volatile, main memory storage system. In *Proc. of the Architectural Support for Programming Languages and Operating Systems*, pages 86–97, 1994.

[28] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. A Durable and Energy Efficient Main Memory Using Phase Change Memory Technology. In *Proc. of the 36th International Symposium on Computer Architecture*, pages 14–23, 2009.