A File System for Storage Class Memory

Xiaojian Wu, A. L. Narasimha Reddy
Department of Electrical and Computer Engineering, Texas A&M University
College Station, Texas 77843
Email: tristan.woo@neo.tamu.edu, reddy@ece.tamu.edu

I. EXTENDED ABSTRACT

In this work, we aim to design a file system for storage class memory (SCM). 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, if the access latency of the storage device is comparable to that of normal memory, then the design complexity of the file system may impact the whole performance.

Current file systems spend considerable complexity due to space management. We focus on simplifying storage management functions within the file system. Our design reuses memory management modules in OS to manage space in SCM. To do so, in addition to modifying some source code in kernel, we need to harden the mapping between logical addresses and physical addresses into SCM.

Traditional file systems assume the underlying persistent storage devices are block devices. On hardware layer, a block device is hidden behind an I/O controller. On software layer, the file systems can access a block device only through the generic block layer in the OS kernel. In our design, we assume storage class memory is directly attached to CPU, and there is a way for firmware/software to distinguish SCM from the other volatile memories. This assumption lets the file systems be able to access the data on the storage class memory in the same way as access normal RAM. With this assumption, we utilize the existing memory management module in the operating system to manage the space on the storage class memory.

II. DESIGN OF SCMFS

A. Layout

Fig. 1 shows the layout of both virtual memory space and physical memory space in SCM File System (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, this memory region is configured with write-through cache policy. The rest of the physical space is mapped into virtual memory space and used to store the whole file system.

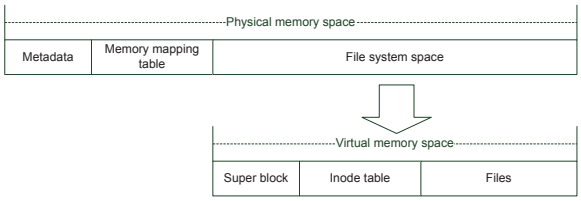

Fig. 1. Memory space layout

In the virtual memory space, the layout is very simple and similar to existing file systems. The super block contains the information about the whole filesystem, such as the block size of the filesystem, the total number of inodes and blocks, etc. The inode table contains the fundamental information of each file or directory, such as file name, size, mode, recent modification timestamp, owner user id, etc. The inode information also includes an offset, through which we can locate the file's contents in the virtual space. The first item in the inode table is the root inode. In our prototype, the total size of virtual memory space for SCMFS is 2^{47} bytes (range: ffff000000000000 - ffff7fffffffffff), which is unused in original Linux kernel.

The layout of SCM file system is illustrated in Fig. 2. In SCM file system, directory files are stored as ordinary files, except that their contents are lists of inode numbers.

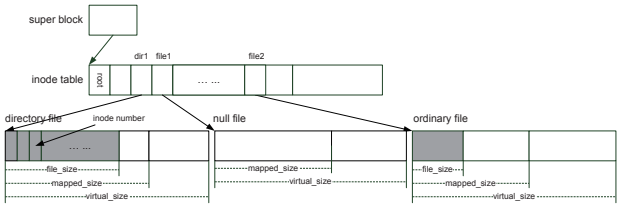

Fig. 2. SCM file system Layout.

B. Space Pre-Allocation

In our original design, all 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. In some situations, this design may cause too frequent allocation/deallocation operations. 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 NULL file. Through space pre-allocation mechanism, we can reduce the number of allocation/deallocation operations significantly.

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 process 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.

C. Modifications to the kernel

In our prototype, we did some modifications to original linux kernel 2.6.31 to support our functionalities. First, we modified the E820 table, which is used by BIOS to report the memory map to the operating system. 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.

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 added a set of memory allocation/deallocation functions, nvmalloc()/nvfree(), which allocate 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.

D. File System Consistency

File system consistency is always a big issue in files 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. 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.

A simple solution to write ordering problem is to configure all the SCM with write-through cache policy. Undoubtedly, this will hurt the writing performance dramatically. In SCMFS, we adopted a compromise solution, in which we only configure the file system metadata and all the directory files with write-through cache policy. This will provide metadata consistency. As to the data consistency, we flush the CPU cache periodically. This provides similar guarantees as the current file systems.

III. Preliminary Results

We use Iozone and Postmark to evaluate the performance of SCMFS and compare it to some 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. Our preliminary experiments show: SCMFS is easy to implement, and our prototype only consists of about 2700 lines of source code. SCMFS's overhead is little, and performs very close to and even better ramfs and tmpfs. SCMFS's performance is much better than and more than twice of that of EXT2FS on Ramdisk, especially with small request sizes. The pre-allocation mechanism significantly speed up (2 times) small file creation and append operations. Our file system consistency mechanism decreases the performance, especially with large number of small files.