Storage Systems for Storage-Class Memory

Haris Volos, Michael Swift
Computer Sciences Department, University of Wisconsin–Madison
{hvolos, swift}@cs.wisc.edu

We make the case for storage systems that expose storage-class memory directly to user mode programs, so that programs can read and write file data without kernel interaction but still maintain the sharing and protection features of file systems.

1. Introduction

Emerging device technologies including phase-change-memory (PCM), spin-torgue transfer RAM (STT-RAM) and memristors promise high-speed storage. These technologies collectively are termed *storage-class memory* (SCM) [3] as data can be accessed through ordinary load/store instructions rather than through I/O requests. Hence, user-mode code can access data directly, so there is no need for the OS to mediate every access. Furthermore, existing virtual memory hardware can protect access to individual data pages. The existing OS structure of file systems as a kernel-level service may no longer be necessary with storage class memory, and causes unneeded complexity and lower performance.

We propose rewriting the storage stack to create a new flexible, high-performance storage architecture enabled by storage-class memory. While existing file systems manage a shared resource inaccessible from user mode, storage-class memory can be mapped into user-mode address spaces. This provides the potential of allowing application to define their own file-system format without needing to extend the kernel, yet still allowing safe sharing between applications.

We next motivate and make the case for storage systems that expose SCM directly to user mode programs, and then we briefly overview our work in progress of building a user-mode file system.

2. User-mode Storage Systems for SCM

Despite rapid advancements in storage technology, the fundamental organization of an operating system's storage architecture has remained stable for decades: applications invoke the kernel to store and retrieve data, which invokes a file system, which invokes a block driver. There have been additions to this stack, such as logical volume managers and RAID controllers, but the structure has remained constant. While there have been attempts to move the file system code to user mode, these systems change the environment for file system code but do not change the layering of components [6, 9]. In addition, some applications, such as databases, bypass the file system but still rely on the lower levels of the storage stack.

Four features of past storage technologies require this design: (1) Disks and other common storage devices do not implement any protection mechanism. Thus, the operating system uses permissions on files to decide which processes have access to which blocks on disk, (2) Disks are accessed through a single shared queue and benefit significantly from scheduling, (3) Slow disks benefit from shared caching, so that each process need not fetch data itself from disk, (4) Disks use DMA to read/write data, which does not respect memory protection. Hence, user-mode access to disks presents a security vulnerability.

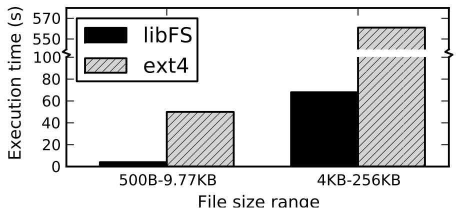

Figure 1. Execution time of the PostMark benchmark. Time axis is not continuous.

However, SCM suffers from none of these limitations: as memory, it can be protected by existing memory-translation hardware. Furthermore, it has much less need for scheduling, as there are not long seek or rotation latencies. Because SCM provides speeds near DRAM, caching data may be unnecessary. Finally, DMA may still be useful for offloading large transfers from the CPU, but can be initiated by the CPU and hence protected.

With these features in mind, this proposal seeks to redesign how operating systems support the file abstraction, which we believe will remain important for organizing data as it provides useful features such as naming, block indexing, protection or sharing data between processes. We seek two benefits from this redesign. First, direct access to file data from user-level can be lower latency than going through the kernel, as it avoids the costs changing mode and cache pollution from entering the kernel [2, 8]. As storage speeds increase, this overhead may dominate the cost of accessing the data. Recent work on solid-state storage showed that for high-speed connections, software overhead significantly limited the available bandwidth to storage [1].

To estimate the performance benefit of direct-access, we wrote a simple library file system (libFS) that stores data in memory. libFS is capable enough to support the PostMark benchmark. Figure 1 presents the execution time of PostMark running on the *ext4* (kernel-mode) and *libFS* (user-mode) file systems. To provide a basis of fair comparison we ran ext4 on top of RAMdisk. That is, both file systems store data in memory. PostMark parameters are 3000 files, 1000000 transactions, and 50/50 read/append and create/delete biases. We performed tests with two file size ranges, 500B-9.77KB (benchmark default) and 4KB-256KB. As shown in the figure, direct access enables up to one order of magnitude performance improvement. While this result should only be interpreted as an upper bound of the performance benefit of direct-access, we believe it demonstrates the potential behind this approach.

As a second benefit, there can be much greater flexibility in the organization of file system data. Currently, most operating systems use a single file system implementation for all processes. However, there are several examples of storage systems that store many large objects within a single file to avoid the file system overhead of separate files. For example, Google's GFS is often used to store many large objects within a single file. Similarly, Facebook's Haystack photo-storage architecture stores many images within a single file. These layouts provide faster indexing of file data and require less dynamic memory to hold per-file metadata, such as inodes, compared to kernel-level files. While a better kernel-level file system could provide similar benefits, the difficulties of supporting multiple file systems on a single machine and the complexities of kernel development encourage this application-level layering of objects within files. But, as a result the naming, protection, and concurrency benefits of files are lost.

3. Towards a User-mode File System

We propose that file systems for SCM should be implemented largely as a library linked into an application, rather than as a shared kernel-level service. Implementing a file system at user level provides flexibility, as an application can determine a layout that is most efficient for its data. In addition, for SCM, it can improve performance by avoiding costly transitions to the kernel. Existing user-mode file-system implementations retain the shared service model, executing as an independent process, and hence provide flexibility but not performance [6, 9]. Past objections about user-mode file systems [10] focus on the need to re-enter the kernel to access a slow device, and the overhead of IPC to a file-system server. However, with SCM and a library file system, neither objection still holds.

We are building the *Memory File System* (MFS) that allows applications to retain the existing file abstraction but perform most file operations, such as opening files and reading/writing data, without kernel involvement. In MFS, a small kernel component, the *MFS kernel* provides access to data stored in SCM. A user-mode library, *MFS lib* is linked to applications and provides the file system API.

The MFS kernel's role in file access is to securely record and enforce resource usage, but not to determine how it is used. The MFS kernel exposes SCM to the MFS lib in the form of memory extents that the MFS lib uses to store data and metadata. While the MFS kernel records the memory extents in use by a file so that a process with permission to read a file can access those extents, it is the responsibility of MFS lib to decide how file data maps into extents. The MFS lib should therefore manage its own metadata to store the mapping of file offsets to extends. Finally, the MFS kernel lets the MFS lib to group a set of files into a collection for sharing and protection. While the MFS kernel ensures that a process has access to all the extents in a file or collection of files, it completely delegates responsibility for naming and indexing to the MFS lib.

3.1 Challenges

Obviously, the above description is incomplete and there are several remaining challenges and open research questions that need to be addressed before having a user-mode file system that is as robust and capable as its kernel-mode counterpart. Below we identify a couple of them.

Protection. While page-based protection provided by the MMU may be sufficient for enforcing access control to data, it may be cumbersome for metadata, which may benefit from a more fine-grain protection mechanism.

Integrity. A file system must protect the integrity of its metadata, such as inode and directory structures. With MFS, this code is implemented by the MFS lib rather than in the kernel. Some file systems may be not satisfied with enforcing integrity in user-mode libraries.

Concurrency. Different file systems enforce different concurrency semantics, such as what happens with concurrent reads and writes to the same portion of the file. There must be therefore support for concurrency control so that multiple processes can atomically write to a file or insert files into a directory simultaneously.

4. Related Work

There have been several prior projects investigating the integration of storage-class memory into file systems. Initially cast as non-volatile RAM (NVRAM), these memories can be used as persistent write buffers to reduce the latency of writing data [4] or hold frequently changing metadata [7]. However, the fundamental file system and storage architecture are left unchanged. More recently, Condit et al describe a file system leveraging SCM's properties [2]. Still, the file system does not provide direct access to storage from user mode and no flexibility in file organization.

The Exokernel [5] and Fuse [9] projects both allow implementation of file systems in user space, similar to our proposed work. Fuse, like other user-level file systems [6, 11], retains the model of a shared file system that accesses storage through a shared device, such as a network or disk. The Exokernel approach is most similar, as it exposed a block interface to applications. However, it still maintained protection of the block device within the kernel, so disk access still required invoking a kernel-mode device driver.

5. Conclusion

We presented our work-in-progress of building storage systems that expose storage-class memory directly to user mode programs, so that programs can read and write file data without kernel interaction but still maintain the sharing and protection features of file systems. Initial experimentation with a simple library file system shows that there is potential in exploring this approach.

References

[1] A. M. Caulfield, A. De, J. Coburn, T. Mollov, R. Gupta, and S. Swanson. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *MICRO 43*, Dec. 2010.

[2] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. Better I/O through byte-addressable, persistent memory. In *SOSP 22*, Oct. 2009.

[3] R. F. Freitas and W. W. Wilcke. Storage-class memory: the next storage system technology. *IBM J. Res. Dev.*, 52(4):439–447, 2008.

[4] D. Hitz, J. Lau, and M. Malcolm. File system design for an nfs file server appliance. Technical Report TR 3002, NetApp, 2005.

[5] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Brice no, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on exokernel systems. In *SOSP 16*, pages 52–65, Oct. 1997.

[6] D. Mazières. A toolkit for user-level file systems. In *USENIX ATC*, June 2001.

[7] E. Miller, S. Brandt, and D. Long. HeRMES: High-performance reliable MRAM-enabled storage. In *HotOS 8*, May 2001.

[8] L. Soares and M. Stumm. FlexSC: Flexible system call scheduling with exception-less system calls. In *OSDI 9*, Oct. 2010.

[9] M. Szeredi. Fuse: Filesystem in userspace. `http://fuse.sourceforge.net`, 2005.

[10] B. Welch. The file system belongs in the kernel. In *Proc. of Second USENIX Mach Symposium*, pages 233–250, 1991.

[11] M. Young, A. Tevanian, R. Rashid, D. Golub, and J. Eppinger. The duality of memory and communication in the implementation of a multiprocessor operating system. In *SOSP 11*, Nov. 1987.