

# Storage Systems for Storage-Class Memory

Haris Volos

University of Wisconsin–Madison

hvolos@cs.wisc.edu

Emerging device technologies including phase-change-memory (PCM), spin-torque transfer RAM (STT-RAM) and memristors promise high-speed storage. These technologies collectively are termed *storage-class memory* (SCM) 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.

**The Case for 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 keep protection of the block device within the kernel, so disk access still requires invoking a kernel-mode device driver [2, 3].

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.

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. As storage speeds increase, this overhead may dominate the cost of accessing data.

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.

Recently, Condit et al describe a file system leveraging SCM’s properties [1]. Still, the file system does not provide direct access to storage from user mode and no flexibility in file organization.

**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. Our file system design is driven by the two goals of providing flexibility and improving performance. Existing user-mode file-system implementations retain the shared service model, executing as an independent process, and hence provide flexibility but not performance as they do not avoid costly transitions to the kernel [3].

We meet these goals via two design principles. First, the kernel is responsible only for safely multiplexing storage. This includes allocating SCM blocks and setting up hardware enforced permissions to protect allocated blocks. Other file system services such as naming and indexing should be provided in user-mode. This separation of concerns between user- and kernel-mode enables the maximum level of flexibility possible without jeopardizing safety, and also improves performance as it avoids invoking the kernel for operations that can be done in user-mode. Second, file system participants should work locally for as long as possible. Participants only communicate between them or with a centralized server only when need to share state. Such a distributed design further improves performance by avoiding communication costs whenever possible. In contrast, a centralized kernel-mode file system always requires file system users communicate their updates to the kernel, which makes updates globally visible.

Our design principles have important implications to integrity and concurrency. A file system must protect the integrity of its metadata, such as inode and directory structures. To that extent, we cannot in general trust updates performed by clients. This implies that all updates have to be performed by a trusted server but this would violate the second principle of distributed design. We allow clients perform updates locally but require clients verify state modified by untrusted clients. Concurrency control is necessary to reason what happens with concurrent reads and writes to the same portion of the file system such as concurrent updates to a directory. We support a basic concurrency model based on pessimistic concurrency control, which allows a single writer and multiple readers per each SCM page. If multiple concurrent writers per page are needed, then updates have to be performed through a central server.

We are currently building the *Memory File System*, a user-mode file system prototype 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.

## References

- [1] 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.
- [2] 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.
- [3] M. Szeredi. Fuse: Filesystem in userspace. <http://fuse.sourceforge.net>, 2005.