# Future File Systems
## A Position Paper

by
David Pease, IBM Almaden Research Center
Darrell Long, University of California, Santa Cruz

**Position Statement**

Relatively fast, affordable, non-volatile memory (storage-class memory) will become available in quantities large enough to replace traditional disk devices in the next decade. The availability of such memories will both enable and demand profound changes in file system architecture and implementation.

**Introduction**

Since the beginning of general-purpose computing, computing systems have run specialized software to manage the disparity in access speeds between the computer's internal memory and its external storage. In the early days, when tape ruled the external storage world, this software could have been as simple as an I/O Control System (IOCS) card deck placed on the back of an executable program deck; the IOCS would have provided simple callable tape positioning, read/write, and error recovery routines. As the computing world moved into the era of direct-access storage devices, dominated by disks drives, the specialized I/O software developed into what we recognize today as file systems.

Modern file systems come in many types, including local, distributed, and cluster, general- and special-purpose, and others. Despite their differences, however, these file systems provide many of the same capabilities and services. Some of these are reviewed below.

The aspect of a file system that is the most important to programmers is its application programming interface, or API. The API typically gives the programmer a rich set of standard calls that can be used to access and manipulate data. An almost ubiquitous example of a file system API is the POSIX-style file system interface; it is supported not only by file systems that run on Unix-like operating systems, but also by most C compiler standard libraries.

Another very visible aspect of a file system is its name space. The name space gives users and programmers a way to name, organize, and refer to specific sets of data (files or data sets). A directory-based, tree-structured organization is the most common type of name space, though there are others (for example, the essentially flat name space supported by the IBM zOS catalog).

A less visible, though no less important, function provided by most file systems is that of in-memory caching. File data is is cached in main memory by the file system for two complementary reasons, both of them related to the huge disparity in access times between current disk storage and main memory systems (approximately a factor of 30,000). For input operations, data is cached in memory in the hope that future reads will be satisfied from the cached data, thus avoiding reads from disk. The data cached in this way may have been read or written by an application some time earlier, or it may have been been read by the file system in anticipation of future access, using any number of predictive techniques.

For output operations, data is cached in order to isolate applications from the slow write times of external devices; data is written into memory, control is returned to the application requesting the write, and the data is "flushed" to disk some time later. Write caching can be problematic to applications due to the lack of guaranteed completion of the write to disk; a system failure shortly after caching a write request can easily lead to data loss. Both read and write caching become problematic when data is shared across systems, as may be done in distributed or clustered file systems. The desire to guarantee cache (and therefore data) consistency between systems leads to complex, and often slow or error-prone locking techniques.

Because of the tight interaction between the file system and its cache, file systems are often closely tied to the memory management component of their host operating system. While this integration provides some interesting capabilities, such as memory-mapped file I/O, it also creates a great deal of complexity. For instance, the interaction between the Windows file system and Virtual Memory Management (VMM) subsystems is extremely complex and difficult to implement correctly.

The final major function provided by the file system is actual I/O device access. At its most basic, this might consist only of converting a file offset into a location on disk, then calling the disk device driver to perform the I/O operation. Even in this simple case, the file system must deal predictably and "correctly" with error conditions returned by the device driver. Depending on the file system and type of request, however, the I/O operation might be considerably more complex, perhaps involving read-modify-write operations, network access, reads or writes to multiple devices, parity computation or validation, or other operations.

**Storage Class Memory-based File Systems**

The storage-class memories that will become available in the next decade will be within a factor of 2-10 times the speed of DRAM memory. They will be large and inexpensive enough to replace today's disk drives. The availability of such fast, affordable, non-volatile memory will drive major changes in file system architecture and implementation.

The externals of the file system, that is the name space and the programming API, will be unlikely to see immediate change as a direct result of such a shift in the underlying storage. Changes to the file system at the levels below the external interfaces will be the most drastic and perhaps the most beneficial, and we will consider those first.

The use of any sufficiently high-speed storage class memory will obviate the need for in-memory data caching. Access to data storage will be fast enough to allow each I/O operation to reference the primary copy of the data. This will eliminate the problem of unprocessed writes from applications (and the associated need for applications to force data to disk at points in their processing). If the memory is available to multiple systems concurrently, a much simpler locking protocol than those common today could suffice to allow direct distributed access to data, without concern for cache consistency.

A storage-class memory such as we envision could be treated as main memory, at least in terms of access and addressing. If the virtual memory capability of the computing system is used to map ranges of the storage-class memory into an application's address space, then an input operation might do nothing more than map a block range of a file into the application space as read-only memory, allowing data accesses to be satisfied directly from the primary copy of the data without any other data transfer. For applications using memory-mapped I/O mode (mmap) the storage could be mapped as writable, allowing this approach to be extended to output operations, as well. If the file were always mapped to

the same memory address, then useful intra- and inter-file pointers could be implemented, allowing for efficent data embedded structures in files.

Unlike disk drives, which can only write in multiples of the physical block (sector) size, storage-class memory can be written at arbitrary byte boundaries. The use of this ability will eliminate the need for the read-modify-write processing necessary today in order to update data within a block when the entire block is not being replaced.

Perhaps more importantly, byte-level addressing will give us the opportunity to rethink the data structures used for implementing the file system. B-tree based structures have been the norm for many years; however, when storage is memory, structures with very different access patterns and behavior such as skip lists or even Bloom filters may be worth exploring.

With the ability to map primary storage into an address space, the sometimes lengthy process of loading applications could be eliminated. Instead, the executable file would be mapped into the address space as read-only memory segments. This is a logical extension of the shared, read-only mapping of applications in most modern operating systems. Although execution from storage-class memory will be somewhat slower than execution from today's main memory, once the program's working set is in the processor's cache this difference will be hidden.

In an environment such as we describe, the file system actually becomes an interface and name space manager for the memory subsystem. Block extent lists will be replaced by memory address ranges, and at the lowest level all files will be processed through memory-mapped I/O (as is done in several modern Unixes today). If the storage class memory is sufficiently large, data could be organized so that files are always in a contiguous address range, making both file management and address space mapping very simple. (In order to avoid costly reorganizations of huge data sets, this requirement might apply only to files that are within a predefined size.)

Though not specifically related to file systems, the blurring of the line between external storage and main memory raises the question of how paging and swap spaces would be implemented and used. The idea of moving data from memory to external storage, when external storage is (storage class) memory, suggests that paging as we know it would disappear. While the virtual memory mechanism is still important for mapping data into address spaces, most pages would never be moved.

Storage-class memories will probably have a very different failure model from current disk drives. While the space efficiency and protection afforded by today's RAID systems will still be desirable, implementation details will need to be completely rethought. It is unlikely that large amounts of data (corresponding to a disk drive) will become suddenly unavailable in a storage-class memory; it is more likely that individual bits, or small ranges of bytes will fail together. Erasure codes matched to the importance of individual data files might replace the indiscriminate use of RAID protection for all files on a particular disk set.

So far we have focused on changes to file system architecture that are essentially invisible to the file system's users. Are there opportunities for creating new storage paradigms that are enabled by the use of storage-class memories? Perhaps a change in the underlying storage architecture will provide opportunities for expanding the abilities of file systems based on storage class memories.

One such opportunity is semantic file system access; that is, the ability to find and access data based on the contents or attributes of the data itself, rather than by using the file's directory location and name.

Semantic access to data is not a new idea, but it seems to be slow in gaining momentum. Is this because users are essentially content with the current paradigm, or is it because the technology to implement useful semantic access is lagging?

We do not believe that semantic acces to data will replace today's tree-structured organization as the primary way to organize files; the idea of organizing data hierarchically predates computers, and may be as efficient a way to keep data for personal use as has been invented. However, when there are very large numbers of files or when data is shared by many people the hierarchical approach begins to break down, and other approaches are needed. Search engines on the Web demonstrate how useful the ability to find documents by content and/or metadata is. The ability to read and index data much more quickly from storage-class memories than from today's disks may make both indexing and searching more practical and thus accelerate the availability and use of semantic access. Certain forms of storage class memory allow the creation of low cost but huge content-addressable memories, which could greatly simplify the construction of semantic file sytems.