acmqueue

A File System All Its Own

Flash memory has come a long way. Now it's time for software to catch up.

Adam H. Leventhal

In the past five years, flash memory has progressed from a promising accelerator,⁷ whose place in the data center was still uncertain, to an established enterprise component for storing performance-critical data^{4,9}. It's rise to prominence followed its proliferation in the consumer world and the volume economics that followed (see figure 1). With SSDs (solid-state devices), flash arrived in a form optimized for compatibility—just replace a hard drive with an SSD for radically better performance. But the properties of the NAND flash memory used by SSDs differ significantly from those of the magnetic media in the hard drives they often displace.² While SSDs have become more pervasive in a variety of uses, the industry has only just started to design storage systems that embrace the nuances of flash memory. As it escapes the confines of compatibility, significant improvements in performance, reliability, and cost are possible.

The native operations of NAND flash memory are quite different from those required of a traditional block device. The FTL (flash translation layer), as the name suggests, translates the block-device commands into operations on flash memory. This translation is by no means trivial; both the granularity and the fundamental operations differ. SSD controllers compete in subspecialties such as garbage collection, write amplification, wear leveling, and error correction.² The algorithms used by modern SSDs are growing increasingly sophisticated despite the seemingly simple block-read and block-write operations that they must support. A very common use of a block device is to host a file system. File systems, of course, perform their own type of translation: from file creations, opens, reads, and writes within a directory hierarchy to block reads and writes. There's nothing innate about file-system operations that makes them well served by the block interface; it's just the dominant standard for persistent storage, and it has existed for decades.

Layering the file system translation on top of the flash translation is inefficient and impedes performance. Sophisticated applications such as databases have long circumvented the file system—again, layers upon layers—to attain optimal performance. The information lost between abstraction layers impedes performance, longevity, and capacity. A file system may “know” that a file is being copied, but the FTL sees each copied block as discrete and unique. File systems also optimize for the physical realities of a spinning disk, but placing data on the sectors that spin the fastest doesn't make sense when they don't spin at all. Volume managers, software that presents collections of disks as a block device, led to similar inefficiencies in disk-based storage, obscuring information from the file system.

Modern file systems such as WAFL (Write Anywhere File Layout)⁵ ZFS, and Btrfs (B-tree file system)¹ integrated the responsibilities previously assigned to volume managers and reorganized the layers of abstraction. The resulting systems were more efficient and easier to manage. Poorly optimized software mattered when operations were measured in milliseconds; it matters much more on flash devices whose operations are measured in microseconds. To take full advantage of flash, users need software expressly designed for the native operations and capabilities of NAND flash.

THE STATE OF SSDS

For many years SSDs were almost exclusively built to seamlessly replace hard drives; they not only supported the same block-device interface, but also had the same form factor (e.g., a 2.5- or 3.5-inch hard drive) and communicated using the same protocols (e.g., SATA, SAS, or FC). This is a bit like connecting an iPod to a car stereo using a tape adapter; now it seems that 30-pin iPod connectors are more common in new cars than tape decks are. Recently SSDs have started to break away from the old constraints on compatibility: some laptops now use a custom form-factor SSD for compactness, and many vendors produce PCI-attached SSDs for lower latency.

The majority of SSDs still emulate the block interface of hard drives: reading and writing an arbitrary series of sectors (512-byte or 4-KB regions). The native operations of NAND flash memory are different enough to create some substantial challenges. Reads and writes happen at the granularity of a page (usually around 8 KB) with the significant caveats that writes can occur only to erased pages, and pages are erased exclusively in blocks of 32-64 (256-512 KB). While a detailed description of how an FTL presents a block interface from flash primitives is beyond the scope of this article, it's easy to get a sense of its complexity. Consider the case of a block in which all pages have been written, and the device receives an operation to logically overwrite the contents of one page.

The FTL could copy the block into memory, modify the page, erase the block, and rewrite it in its entirety, but this would be very slow—slower even than a hard drive! In addition, each write or erase operation wears out NAND flash. Chips are rated for a certain number of such operations—anywhere from 500-50,000 cycles today depending on the type and quality, and those numbers are shrinking as the chips themselves shrink. A naive approach to block management would quickly wear down the media; and to compound the problem, a frequently overwritten region would wear out before other regions. For these reasons, FTLs use an indirection layer that allows data to be written at arbitrary locations and implements wear leveling, the process of distributing writes uniformly across the media.²

BRIDGING THE GAP

The algorithms that make up an FTL are highly complex but no more than those of a modern file system. Indeed, the FTL and the file system have much in common. Both track allocated versus free regions, both implement a logical-to-physical mapping, and both translate one operation set to another. Newer FTLs even include facilities such as compression and deduplication—still marquee features for modern file systems. FTLs and file systems are usually built in isolation. The idea of a dramatic integration and reorganization of the responsibilities of the FTL and file system represents a classic conundrum: who will write software for nonexistent hardware, and who will build hardware to enable heretofore-unwritten software?

Most SSD vendors are focused on a volume market where requiring a new file system on the host would be an impediment rather than an advantage. SSD vendors could enable the broader file-system developer community by providing different interfaces or opening up their firmware, but again—and without an obvious and compelling file system—there's little incentive. The exception was Indilinx's participation in the OpenSSD¹⁰ project, but the primary focus was FTL development and experimentation within conventional bounds. OpenSSD became effectively defunct when OCZ acquired Indilinx. There seems to be no momentum and only vague incentive for vendors to give developers the level of visibility and control that they most want. Mainstream efforts to build flash awareness into file systems have led to more modest modifications to the interface between file system and SSD.

The most publicized interface between the file system and SSD is the ATA TRIM command or its counterpart, the SCSI UNMAP command. TRIM and UNMAP convey the same meaning to a device: the given region is no longer in use. One of the challenges with an FTL is efficient space management; and the more space that's available, the easier it is to perform that task. As free space is exhausted, FTLs have less latitude to migrate data, and they need to keep data in an increasingly compact form; with lots of free space, FTLs can be far sloppier.

For both performance and redundancy, almost all SSDs “overprovision.” They include more flash memory capacity than the advertised capacity of the SSD by anywhere from 10 to 100 percent. File systems have the notion of allocated and free blocks, but there isn't a means—or a reason—to communicate that information to a hard drive. To let SSDs reap the benefits of free storage, modern file systems use the TRIM or UNMAP commands to indicate that logical regions are no longer in use. Some SSDs—particularly those designed for the consumer market—greatly benefit from file systems that support TRIM and UNMAP. Of course, for a file system whose steady state is close to full, TRIM and UNMAP have very little impact because there aren't many free blocks.

INCREMENTAL REVOLUTION

While many companies participate in incremental improvements, the most likely candidates to create a flash-optimized file system are those that build both SSDs and software that runs on the host. The most popularized example thus far is DirectFS⁶ from FusionIO. Here, the flash storage provides more expressive operations for the file system. Rather than solely using the legacy block interface, DirectFS interacts with a virtualized flash storage layer. That layer manages the flash media much like a traditional FTL but offers greater visibility and an expanded set of operations to the file system above it.

DirectFS achieves significant performance improvements not by supplanting intelligence in the hardware controller, but by reorganizing responsibilities between the file system and flash controller. For example, FusionIO has proposed extensions to the SCSI standard that perform scattered reads and writes atomically.³ These are easily supported by the FTL, but dramatically simplify the logic required in a file system to ensure metadata consistency in the face of a power failure. DirectFS also relies on storage that provides a “sparse address space”, which effectively transfers allocation and block mapping responsibilities from the file system to the FTL, a task the FTL already must do. A 2010 article by William Josephson et al. states that “novel layers of abstraction specifically for flash memory can yield substantial benefits in software simplicity and system performance.”⁶

As with TRIM, incrementally adding expressiveness and functionality to the existing storage interfaces allows file systems to take advantage of new facilities on devices that provide them. Storage system designers can choose whether to require devices that provide those interfaces or to implement a work-alike facility that they disable when it's not needed. Device vendors can decide whether supporting a richer interface represents a sufficient competitive advantage. Though this approach may never lead to an optimal state, it may allow the industry to navigate monotonically to a sufficient local maximum.

THE CHICKEN AND THE EGG

There are still other ways to construct a storage system around flash. A more radical approach is to go further than DirectFS, assigning additional high-level responsibilities to the file system such as block management, wear leveling, read-disturb awareness, and error correction. This would allow for a complete reorganization of the software abstractions in the storage system, ensuring sufficient information for proper optimization where today's layers must cope with suboptimal information and communication. Again, this approach today requires a vendor that can assert broad control over the whole system—from the file system to the interface, controller, and flash media. It is certainly tenable for closed proprietary systems—indeed, several vendors are pursuing this approach—but for it to gain traction as a new open standard would be difficult.

SSD ALCHEMY

The SSDs that exist today for the volume market are cheap and fast, but they exhibit performance that's inconsistent and reliability that's insufficient. Higher-level software designed with full awareness of those shortcomings could turn that commodity iron into gold. Without redesigning part or all of the I/O interface, those same SSDs could form the basis of a high-performing and highly reliable storage system.

Rather than designing a file system around the properties of NAND flash, this approach would

treat the commodity SSDs themselves as the elementary unit of raw storage. NAND flash memory already has complicated intrinsic properties; the emergent properties of an SSD are even more obscure and varied. A common pathology with SSDs, for example, is variable performance when servicing concurrent or interleaved read and write operations. Understanding these pathologies sufficiently and creating higher-level software to accommodate them would represent the flash version of an existential software parable: enterprise quality from commodity components. It's a phenomenon that the storage world has seen before with disks; software such as ZFS from Sun has produced fast, reliable systems from cheap components.

The only easy part of this transmutation is finding the base material. Building such a software system given a single, unchanging SSD would already be complicated; doing it amid the changing diversity of the SSD market further complicates the task. The properties of flash differ between types and fabrication processes, but change happens at the rate of hardware evolution. SSDs change not only to accommodate the underlying media and controller hardware, but also at the speed of software, fixing bugs and improving algorithms. Still, some vendors are pursuing this approach¹¹ because, while it is more complex than designing for purpose-built hardware, it has the potential to produce superlative systems that ride the economic curve of volume SSDs.

NEXT FOR FLASH

The life span of flash as a relevant technology is a topic of vigorous debate. The cost of flash memory has yet to catch up with that of hard disk drives, but prices per gigabyte are approaching those of HDDs from less than a decade ago, as shown in figure 1. While flash has ridden its price and density trends to a position of relevance, some experts anticipate fast-approaching limits to the physics of scaling NAND flash memory. Others foresee several decades of flash innovation. Whether it is flash or some other technology, nonvolatile solid-state memory will be a permanent part of the storage hierarchy, having filled the yawning gap between hard-drive and CPU speeds.⁸

The next evolutionary stage should see file systems designed explicitly for the properties of solid-state media rather than relying on an intermediate layer to translate. The various approaches are each imperfect. Incremental changes to the storage interface may never reach the true acme. Creating a new interface for flash might be untenable in the market. Treating SSDs as the atomic unit of storage may be just another half-measure, and a technically difficult one at that.

Some companies today are betting on the relevance of flash at least in the near term—some working within the confines of today's devices, others building, augmenting, or replacing the existing interfaces. The performance of flash memory has whetted the computer industry's appetite for faster and cheaper persistent storage. The experimentation phase is long over; it's time to build software for flash memory and embrace the specialization needed to realize its full potential.

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

ADAM H. LEVENTHAL is the CTO at Delphix, a database virtualization company. Previously he served as Lead Flash Engineer for Sun and then Oracle where he designed flash integration in the ZFS Storage Appliance, Exadata, and other products. For over a decade, Adam has been involved in storage system design at Sun, Oracle, and now Delphix.

REFERENCES

1. Btrfs wiki. https://btrfs.wiki.kernel.org/index.php/Main_Page
2. Cornwell, M. 2012. Anatomy of a solid-state drive. *ACM Queue* 10(10); <http://queue.acm.org/detail.cfm?id=2385276>
3. Elliott, R., Batwara, A. 2012. Notes to T10 Technical Committee. 11-229r4 SBC-4 SPC-5 Atomic writes and reads <http://www.t10.org/cgi-bin/ac.pl?t=d&f=11-229r4.pdf>; 12-086r2 SBC-4 SPC-5 Scattered writes, optionally atomic <http://www.t10.org/cgi-bin/ac.pl?t=d&f=12-086r2.pdf>; 12-087r2 SBC-4 SPC-5 Gathered reads - optionally atomic <http://www.t10.org/cgi-bin/ac.pl?t=d&f=12-087r2.pdf>
4. Gray, J., Fitzgerald, B. 2008. Flash disk opportunity for server applications. *ACM Queue* 06(04); <http://queue.acm.org/detail.cfm?id=1413261>
5. Hitz, D., Lau, J.; Malcolm, M. 1994. File system design for an NFS file server appliance. WTEC'94 USENIX Winter 1994 Technical Conference: 19-19. <http://dl.acm.org/citation.cfm?id=1267093>
6. Josephson, W. K., Bongo, L. A., Li, K., Flynn, D. 2010. DFS: A file system for virtualized flash storage. *ACM Transactions on Storage (TOS)*; 6(3). <http://dl.acm.org/citation.cfm?id=1837922>
7. Leventhal, Adam. 2008. Flash storage today. *ACM Queue* 6(4); <http://queue.acm.org/detail.cfm?id=1413262>
8. Leventhal, Adam. 2009. Triple-parity RAID and beyond. *ACM Queue* 7(11); <http://queue.acm.org/detail.cfm?id=1670144>
9. Moshayedi, M., Wilkison, P. 2008. Enterprise SSDs. *ACM Queue* 06(04); <http://queue.acm.org/detail.cfm?id=1413263>
10. The OpenSSD project. http://www.openssd-project.org/wiki/The_OpenSSD_Project
11. PureStorage FlashArray. <http://www.purestorage.com/flash-array/purity.html>