Interview with Steve Swanson

RIK FARROW

Steven Swanson is an associate professor in the Department of Computer Science and Engineering at the University of California, San Diego and the director of the Non-Volatile Systems Laboratory. His research interests include the systems, architecture, security, and reliability issues surrounding non-volatile, solid-state memories. He also co-leads projects to develop low-power co-processors for irregular applications and to devise software techniques for using multiple processors to speed up single-threaded computations. In previous lives he has worked on scalable dataflow architectures, ubiquitous computing, and simultaneous multithreading. He received his PhD from the University of Washington in 2006 and his undergraduate degree from the University of Puget Sound.
swanson@eng.ucsd.edu

Rik is the editor of *;login:*.
rik@usenix.org

While walking the poster session at OSDI '14, I saw Steve Swanson. I wanted to talk to him about the past and future of non-volatile memory (NVM), since he and his students had produced many papers about the topic that I'd heard presented. I also had some vague ideas about a class for FAST '15 on the history of NVM. Alas, Steve was moving so quickly through the crowd that I never caught up to him in Broomfield.

However, Steve later agreed to an email interview.

Rik: How did you get interested in non-volatile storage?

Steve: I was a new professor and was talking to a colleague from industry who was working on storage systems. He mentioned that flash memory was really beginning to shake things up, so I took a look at it. It turned out to be a perfect research problem for me: It involved hardware and system design (which is, broadly, what I did my PhD on), and it centered around a huge disruption in the performance of a particular piece of the system. Those kinds of changes always result in big challenges and open up all kinds of new areas for research. My students and I dove in, and it's been really exciting and has allowed us to do interesting work on both the hardware and software side as well as at the application level.

Rik: I wanted to start off with some history, or at least try to better understand how we got to where we are today with flash-based storage devices. Having heard many paper presentations, it seems like there have been, and will continue to be, two big issues, both of them interrelated.

These are the flash translation layer (FTL) and the disk-style interface for flash-based storage. Can you explain why vendors adopted these interfaces?

Steve: FTLs arose because SSD vendors needed to make it as easy as possible for customers to use their new drives. It's a much simpler proposition to sell something as a faster, drop-in replacement for a hard drive. If you can make your flash drive look like a hard drive, you immediately have support from all major operating systems, you can use existing file systems, etc. The alternative is to tell a customer that you have a new, fast storage device, but it will require them to completely change the way their software interacts with storage. That's just a non-starter.

The disk-based interface that FTLs emulate emerged because it is a natural and reasonably efficient interface for talking to a disk drive. Indeed, just about everything about how software interacts with storage has been built up around disk-based storage. It shows up throughout the standard file-based interfaces that programmers use all the time.

The problem is that flash memory looks nothing like a disk. The most problematic difference is that flash memory does not support in-place update. Inside an SSD, there are several flash chips. Each flash chip is broken up into 1000s of "blocks" that are a few hundred kilobytes in size. The blocks are, in turn, broken into pages that are between 2 and 16 KB.

Flash supports three main operations. First, you can "erase" a block, that is, set it to all 1s. It seems like "erased" should be all 0s but the convention is that it's all 1s. Erasing a block takes a few milliseconds. Second, you can "program" a page in an erased block, which means you can change some of the 1s to 0s. You have to program the whole page at once, and you must program the pages within a block in order. Programming takes hundreds of microseconds. Third, you can read a page, and reading takes tens of microseconds. The result of this is that if you want to change a value in a particular page, you need to first erase the entire block and then reprogram the entire block. This is enormously inefficient.

The final wrinkle is that you can only erase each block a relatively small number of times before it will become unreliable—between 500 and 100,000 depending on the type of flash chip. This means that even if erasing and reprogramming a block were an efficient way to modify flash, performing an erase on every modification of data would quickly wear out your flash.

So the FTL's job is pretty daunting: It has to hide the asymmetry between programs and erasures, ensure that erasures are spread out relatively evenly across all the flash in the system so that "hot spots" don't cause a portion of the flash to wear out too soon, present a disk-like interface, and do all this efficiently and quickly. Meeting these challenges has turned out to be pretty difficult, but SSD manufacturers have done a remarkably good job of producing fast, reliable SSDs.

The first SSDs looked exactly like small hard drives. They were the same shape, and they connected to the computer via standard hard drive interface protocols (i.e., SATA or SAS). But those protocols were built for disks. Flash memory provided the possibility of building much faster (in terms of both bandwidth and latency) storage devices than SATA or SAS could support. Importantly, SSD could also support much more concurrency than hard drives, and they supported vastly more efficient random accesses than hard drives.

The first company to take a crack at something better was FusionIO. They announced and demonstrated their ioDrive product in September 2007. Instead of using a conventional form factor and protocol, the ioDrive was a PCIe card (like a graphics card) and used a customized interface that was tuned for flash-based storage rather than disk-based storage. FusionIO also began to experiment with new interfaces for storage, making it look quite different from a disk. It's not clear how successful this has been. The disk-like interface has a heavy incumbent advantage.

More recently, NVM Express has emerged as a standard for communicating with the PCIe-attached SSDs. It supports lots of concurrency and is built for low-latency, high-bandwidth drives. Many vendors sell (or will sell shortly) NVMe drives.

Another set of systems has taken a different approach. Rather than use NVMe to connect an SSD to a single system, they build large boxes full of flash and expose them over a network-like interconnect (usually Fibre Channel or iSCSI) to many servers. These network-attached storage (NAS) SSDs must solve all the same problems NVMe or SATA SSDs must solve, but they do address one puzzle that faces companies building high-end SSDs: These new drives can deliver so much storage performance that it's hard for a single server to keep up. By exposing one drive to many machines, NAS SSDs don't have that problem. Violin and Texas Memory Systems fall into this camp.

Rik: If vendors have done such a great job with flash, why has there been so much academic research on it?

Steve: I think the main problem here is that most researchers don't know what industry is actually doing. The inner workings of a company's FTL are their crown jewels. Physically building an SSD (i.e., assembling some flash chips next to a flash controller on PCB) is not technically challenging. The real challenge is in managing the flash so that performance is consistent and managing errors so that they can meet or exceed the endurance ratings provided by flash chip manufacturers. As a result, the research community has very little visibility into what's actually going on inside companies. Some researchers may know, but the information is hidden behind NDAs.

Of course, designing a good FTL is an interesting problem, and there are many different approaches to take, so researchers write papers about them. However, it's not clear how much impact they will have. Maybe the techniques they are proposing are cutting edge, extend the state of the art, and/or are adopted by companies. Or maybe they aren't. It's hard to tell, since companies don't disclose how their FTLs work.

My personal opinion is that, on the basic nuts and bolts of managing flash, the companies are probably ahead of the researchers, since that technology is directly marketable, the companies are better funded, and they have better access to proprietary information about flash chips, etc.

I think researchers have the upper hand in terms of rethinking how SSD should appear to the rest of the system—for example, adding programmability or getting away from the legacy block-based interface, since this kind of fundamental rethinking of how storage should work is more challenging in the commercial environment. However, I think it's probably the more interesting part of SSD research and, in the long term, will have more impact than, for example, a new proprietary wear-leveling scheme.

Rik: I've heard several paper presentations that cover aspects of NVM when it has become byte addressable, instead of block addressable, as it is today. That's assuming, of course, that the promises come true. Can you talk about future directions for research?

Steve: I think the most pressing questions going forward lie along four different lines:

Byte-addressable memories will probably first appear in small quantities in flash-based SSD. One important question is how can we use small amounts of byte-addressable NVM to improve the performance of flash-based SSDs. This is the nearest-term question, and there are already some answers out there. For instance, it's widely known that FusionIO (now SanDisk) uses a form of byte-addressable NVM in its SSDs.

A second likely early application for NVM is in smartphones and other mobile devices. You can imagine a system with a single kind of memory that would serve the role of modern DRAM and also serve as persistent data storage. Since it would have the performance of DRAM, it could alter the programming model for apps: Rather than interacting with key-value stores and other rather clumsy interfaces to persistent storage, they could just create data structures in persistent memory. This would, I think, be a nice fit for lots of small, one-off apps. The main challenge here is in making it easy for programmers to get the persistent data structures right. It's very hard to program a linked list or tree so that, if power fails at an inopportune moment, you can ensure that the data structure remains in a usable state. We have done some work in this area recently as has Mike Swift's group at the University of Wisconsin in Madison, but there's much left to do.

If we solve the next problem, then many of the techniques that we could use in mobile systems would be applicable in larger systems too.

Third, if byte-addressable memories are going to be useful in datacenter-scale storage systems, the data they hold must be replicated, so that if the server hosting one copy goes down, the data is still available. This is a challenge because the memories have access times on the order of tens of nanoseconds, while network latencies are on the order of (at least) a few microseconds. How can we transmit updates to the backup copy without squandering the performance of these new, fast, byte-addressable memories? There are many possible solutions. We've done work on a software-based solution, but it's also possible that we should integrate the network directly into the memory system. This also raises the question of how to reconcile the large body of work from the distributed systems community on distributed replication with the equally large body of work from the architecture community on how to build scalable memory systems. Both fields deal with issues of data consistency and how updates at different nodes should interact with one another, but they do so in very different ways.

The fourth area of interest is in how we can integrate I/O more deeply into programming languages. In modern languages, I/O is an afterthought, so the compiler really has no idea I/O is going on and can't do anything to optimize it. This was not a big deal for disk-based systems, since disk I/O operates on time scales so large (that is, they are so slow) that the compiler could not hope to do anything to improve I/O performance. As storage performance increases, it becomes very feasible that a compiler could, for example, identify I/O operations and execute them a few microseconds early so that the code that needs the results would not have to wait for them. Doing this means we need to formalize the semantics of I/O in a precise way that a compiler could deal with.