Log-structured file systems: There's one in every SSD

September 18, 2009
This article was contributed by Valerie Aurora (formerly Henson)

When you say "log-structured file system," most storage developers will immediately think of Ousterhout and Rosenblum's classic paper, The Design and Implementation of a Log-structured File System - and the nearly two decades of subsequent work attempting to solve the nasty segment cleaner problem (see below) that came with it. Linux developers might think of JFFS2, NILFS, or LogFS, three of several modern log-structured file systems specialized for use with solid state devices (SSDs). Few people, however, will think of SSD firmware. The flash translation layer in a modern, full-featured SSD resembles a log-structured file system in several important ways. Extrapolating from log-structured file systems research lets us predict how to get the best performance out of an SSD. In particular, full support for the TRIM command, at both the SSD and file system levels, will be key for sustaining long-term peak performance for most SSDs.

What is a log-structured file system?

Log-structured file systems, oddly enough, evolved from logging file systems. A logging (or journaling) file system is a normal write-in-place file system in the style of ext2 or FFS, just with a log of write operations bolted on to the side of it. (We'll use the term "journaling file system" in the rest of the paper to avoid confusion between "logging" and "log-structured" file systems.) A journaling file system keeps the on-disk state of the file system consistent by writing a summary of each write operation to the log, stored somewhere non-volatile like disk (or NVRAM if you have the money), before writing the changes directly to their long-term place in the file system. This summary, or log record, contains enough information to repeat the entire operation if the direct write to the file system gets interrupted mid-way through (e.g., by a system crash). This operation is called replaying the log. So, in short, every change to the file system gets written to disk twice: once to the log, and once in the permanent location.

Around 1988, John K. Ousterhout and several collaborators realized that they could skip the second write entirely if they treated the entire file system as one enormous log. Instead of writing the operation to the log and then rewriting the changes in place somewhere else on the disk, it would just write it once to the end of the log (wherever that is) and be done with it. Writes to existing files and inodes are copy-on-write - the old version is marked as free space, and the new version is written at the end of the log. Conceptually, finding the current state of the file system is a matter of replaying the log from beginning to end. In practice, a log-structured file system writes checkpoints to disk periodically; these checkpoints describe the state of the file system at that point in time without requiring any log replay. Any changes to the file system after the checkpoint are recovered by replaying the relatively small number of log entries following the checkpoint.

One of the interesting benefits of the log-structured file system (LFS) structure is that most writes to the file system are sequential. The section describing the motivation for Sprite LFS, written nearly 20 years ago, demonstrates how little has changed in the storage world:

> Over the last decade CPU speeds have increased dramatically while disk access times have only improved slowly. This trend is likely to continue in the future and it will cause more and more applications to become disk-bound. [...] Log-structured file systems are based on the assumption that files are cached in main memory and that increasing memory sizes will make the caches more and more effective at satisfying read requests. As a result, disk traffic will

become dominated by writes.

But wait, why are we still talking about disk seeks? SSDs have totally changed the performance characteristics of storage! Disks are dead! Long live flash!

Surprisingly, log-structured file systems are more relevant than ever when it comes to SSDs. The founding assumption of log-structured file systems - that reads are cheap and writes are expensive - is emphatically true for the bare-metal building blocks of SSDs, NAND-based flash. (For the rest of this article, "flash" refers to NAND-based flash and SSD refers to a NAND-based flash device with a wear-leveling, write-gathering flash translation layer.) When it comes to flash, reads may be done at small granularities - a few hundreds of bytes - but writes must be done in large contiguous blocks - on the order of tens of thousands or hundreds of thousands of bytes. A write to flash takes two steps: First the entire block is cleared, setting all the bits to the same value (usually 1, counter-intuitively). Second, individual bits in the block are flipped back to 0 until you get the block you wanted.

Log-structured file systems turn out to be a natural fit for flash. One of the details of the log-structured design is that the log is written in large contiguous chunks, called "segments," on the order of several megabytes in size. To cut down on metadata overhead and get the best performance, log entries are gathered and written out sequentially to a completely free segment. Most segments are partially in use and partially free at any given time, so the file system has to collect all the in-use data from a segment and move it elsewhere before it can start writing to it. When the file system needs a fresh segment, it first cleans an existing partially-used segment by moving all the in-use, or live data to another free segment - basically, it garbage-collects. Now that everything is arranged properly, the file system can do one big streaming write to the empty segment. This system of segments and cleaning is exactly what is needed to efficiently write to a flash device, given the necessity to erase large contiguous blocks of flash before writing to them.

Sadly, many thousands of people probably now associate the Tux penguin bootup logo with the inability to watch TV on long distance flights.

The match between log-structured file systems and flash is obvious when you look at file systems written for the bare flash programming interface - that is, for devices without built-in wear-leveling or write-gathering. File systems that know about and have to manage erase blocks and other details of the flash hardware are almost invariably log-structured in design. The most widely used such file system for Linux is JFFS2, used in many embedded devices, such as ticket machines and seatback airline entertainment systems. More than once, I've boarded a plane and seen a JFFS2 error message reporting flash corruption on a hung seatback entertainment system. (Sadly, many thousands of people probably now associate the Tux penguin bootup logo with the inability to watch TV on long distance flights.)

For SSDs that export a disk-style block interface - most consumer-grade SSDs these days - the operating systems uses a regular file system to talk to the SSD via the block interface (that is, read block #37 into this buffer, write this buffer into block #42, etc.). However, this system still contains the logical equivalent of a log-structured file system; it's just hidden inside the SSD. The firmware that implements wear-leveling, write-gathering, and any other features has to solve the same problems as a log-structured file system.

Most SSD manufacturers refuse to reveal any details of their internal firmware, but we can be fairly confident that it has a lot in common with log-structured file systems. First, the only way to implement efficient random writes is to buffer them and write them out to a single erase block together. This requires clearing an erase block, moving all the in-use blocks to another area, and keeping a mapping between the logical location of blocks and their physical locations - exactly what a log-structured file system does. Second, when we do get SSD implementation details from research publications, they look like log-structured file systems. Third, when we look at long-term performance testing of SSDs, we see the same pattern of performance degradation over time that we do with log-structured file systems. We'll talk about this in detail in the next section.

Log-structured file system performance

Log-structured file systems are a natural fit for flash-based storage today, but back in 1990, they appeared to have great potential for disk-based file systems as well. Yet, as we all know, we're not using log-structured file systems on our disk-based laptops and servers. What happened?

In short, log-structured file systems performed relatively well as long as most of the segment cleaning - movement of live data out of a segment so it can be re-used - could be done in the background when the file system wasn't busy with "real" work. The first major follow-up paper on LFS [PDF] found performance of LFS degraded by up to 40% from the best case at real-world levels of disk utilization, memory-to-disk ratio, and file system traffic. In short, in the steady state the file system was spending a significant amount of disk access time cleaning segments - moving old data out of a segment so it could be used for new writes. This **segment cleaning problem** was the subject of active research for at least another decade, but none of the solutions could consistently beat state-of-the-art write-in-place file systems at practical levels of disk utilization. It's a little bit like comparing garbage collection to explicit reference counting for memory management; when memory usage is low and the occasional high latency hit is okay, the convenience of garbage collecting outweighs the performance benefits. But at "high" levels of disk utilization - as little as 50% - the cleaning cost and periodic high latencies waiting for space to be freed up become a problem.

As the first LFS paper showed, the key to good performance in a log-structured file system is to place data such that nearly empty segments are created about as quickly as they are used. The file system write bandwidth is limited by the rate at which it can produce clean segments. The worst case happens when, in a file system that is X% full, every segment is also X% full. Producing one clean segment requires collecting the live data from:

```
N = ceiling(1/(1 - X))
```

segments and writing out the old data to `N - 1` of those segments. For a disk utilization of 80%, we get:

```
N = ceiling(1/(1 - .80)) = 1/.20 = 5
```

segments to clean. If segments were 1MB in size, we'd have to read

```
5 * 800KB = 4MB
```

of data seekily and write 4MB sequentially before we could write 1MB of new data. (Note to pedants: I'm using MB/KB in powers of 10, not 2).

The best case, instead, is a file system with two kinds of segments, completely full and completely empty. The best case write pattern is one that changes all of the metadata and data in a single segment, so that when the new versions are written out, the old versions are freed and the entire segment becomes free again. Reality lies somewhere between these two cases. The goal for a log-structured file system is to create a bimodal segment usage distribution: Most segments are either very full or very empty, and full segments tend to be unchanged. This turns out to be difficult to achieve.

SSDs have an extra interesting constraint: wear-leveling. Even in the best case in which most segments are 100% full and no writes ever change the data in them, the SSD must still move those segments around occasionally because it has to spread writes out over every available flash block. This adds an extra segment move in some cases and makes achieving good performance even harder than in a disk-based log-structured file system.

Lessons - learned?

It's great that SSD manufacturers can learn from two decades of prior work on log-structured file systems. What's not clear is whether they are doing so. Most manufacturers take a very closed approach to SSD firmware development - it's the secret sauce that turns cheap commodity flash with very low margins into extremely expensive, reliable, high-performance storage devices with high margins. Some manufacturers are clearly better at this task than others. Currently, manufacturers are taking the trade secret strategy for maintaining their competitive advantage - apply for patents on individual elements of the design, but keep

maintaining their competitive advantage - apply for patents on individual elements of the design, but keep the overall implementation a secret. The message to file systems developers is "Just trust us" and "Don't worry your pretty little systems programmers' heads about it" whenever we ask for more information on SSD implementation. You can't particularly argue with this strategy at present, but it tends to come from (and reinforce) the mindset that not only refuses to share information with the outside, but also ignores information from the outside, such as previously published academic work.

One of the greatest missed opportunities for optimization based on lessons learned from log-structured file systems is the slow adoption of TRIM support for SSDs. TRIM is a command to a block device informing it that a certain range of blocks is no longer in use by the file system - basically a `free()` call for blocks. As described earlier, the best performance comes when empty segments are created as a side effect of ongoing writes. As a simple example, imagine a segment that contains only a single inode and all of its file data. If the next set of writes to the file system overwrites all of the file data (and the inode as a side effect), then that segment becomes completely free and the file system doesn't have to move any live data around before it uses that segment again. The equivalent action for an SSD is to write to a block that has already been written in the past. Internally, the SSD knows that the old copy of that block is now free, and it can reuse it without copying its data elsewhere.

But log-structured file systems have a distinct advantage over pre-TRIM SSDs (basically all commercially available SSDs as of now, September 2009). Log-structured file systems know when on-disk data has been freed even when it isn't overwritten. Consider the case of deleting the one-segment file: the entire segment is freed, but no overwrite occurred. A log-structured file system knows that this happened and now has a free segment to work with. All the SSD sees is a couple of tiny writes to other blocks on the disk. As far as it's concerned, the blocks used by the now-deleted file are still precious data in-use by the file system and it must continue to move that data around forever. Once every block in the device has been written at least once, the SSD is doomed to a worst case performance state in which its spare blocks are at a minimum and data must be moved each time a new block is rotated into use.

As we've seen, the key to good performance in a log-structured file system is the availability of free or nearly-free segments. An SSD without TRIM support does not know about many free segments and accrues an immense performance disadvantage, which make it somewhat shocking that any SSD ever shipped without the TRIM feature. My guess is that SSDs were initially performance tested only with write-in-place file systems (cough, cough, NTFS) and low total file system usage (say, 70% or less).

Unfortunately, TRIM in its current form is both designed and implemented to perform incredibly poorly: TRIM commands aren't tagged and at least one SSD takes hundreds of milliseconds to process a TRIM command. Kernel developers have debated exactly how to implement TRIM support at the Linux Plumbers Conference, at the Linux Storage and File System Workshop, and on mailing lists: what the performance cost of each TRIM is, what granularity TRIMs should have, how often they should be issued, and whether it's okay to forget or miss TRIM commands. In my opinion, the in-use/free state of a block on a TRIM-enabled device should be tracked as carefully as that of a page of memory. The file system implementation can take the form of explicit synchronous `alloc()/free()` calls, or else asynchronous garbage collection (during a file system check or scrubbing run), but we shouldn't "leak" in-use blocks for all the same reasons we don't leak memory. Additionally, in an ideal world, TRIM would be redesigned or replaced by a command that is a full-featured, well-designed first-class citizen in the ATA spec, rather than a hack bolted on after the fact.

Of course, all this is speculation in the absence of implementation details from the SSD manufacturers. Perhaps some SSD firmware programmers have come up with entirely new algorithms for remapping and write-gathering that don't resemble log-structured file systems at all, and the performance characteristics and optimizations we have seen so far just happen to match those for log-structured file systems. However, so far it appears that treating an SSD as though it were backed by a log-structured file system is a good rule of thumb for getting good performance. Full TRIM support by both SSDs and file systems will be key to long-term good performance.

(Log in to post comments)

Log-structured file systems: There's one in every SSD
Posted Sep 18, 2009 19:50 UTC (Fri) by **SEJeff** (subscriber, #51588) [Link]

Keep up the excellent articles Val.

Log-structured file systems: There's one in every SSD
Posted Sep 18, 2009 20:57 UTC (Fri) by **smoogen** (subscriber, #97) [Link]

I wonder how much work it would take to build a SSD from a bunch of flash that would be open for

kernel people to 'see how it could be done.' An open SSD would probably be much more expensive than commercial items but might allow for TRIM and other items to be done 'in the open' to experimentally test whether they work well and how they work on flash.

Log-structured file systems: There's one in every SSD
Posted Sep 18, 2009 21:09 UTC (Fri) by **flewellyn** (subscriber, #5047) [Link]

Flash itself is dirt cheap. Making an SSD just requires a controller to handle the I/O, the wear levelling, and other functions. I imagine you could do it with a small, low-power CPU, a small amount of firmware space (on a separate, smaller flash unit?), some DRAM for caching, and energy storage to let the device flush the cache in the event of a power drop.

Fitting all of that into a standard hard disk form factor would probably be doable. I imagine the high costs on SSDs are mostly due to the development costs.

Log-structured file systems: There's one in every SSD
Posted Sep 18, 2009 21:19 UTC (Fri) by **drag** (subscriber, #31333) [Link]

Well flash is probably cheap, but they are sticking a hell of a lot of flash on those devices.

Log-structured file systems: There's one in every SSD
Posted Sep 18, 2009 21:58 UTC (Fri) by **flewellyn** (subscriber, #5047) [Link]

This is true. Even so, I imagine the overhead would be lower for a "DIY" device.

Log-structured file systems: There's one in every SSD
Posted Sep 21, 2009 9:09 UTC (Mon) by **michaeljt** (subscriber, #39183) [Link]

And if a DIY device could be made to work reasonably well, then some company would be sure to start using the software in a commercial device, which would have a price advantage on the market. Then just make sure that some of the critical components are GPLv3, and you have a load of devices that will let you update their firmware for further experiments.

Log-structured file systems: There's one in every SSD
Posted Sep 19, 2009 6:57 UTC (Sat) by **butlerm** (subscriber, #13312) [Link]

If you are making an open SSD solution, the last thing you want to do is make the basic hardware device a "disk", or contain a controller, a cpu, wear leveling algorithms, intelligence, or any standard disk interface.

What you want is something like a PCIe or perhaps Firewire interface that lets software running on the central CPU (or perhaps a peripheral card) read and write the flash with software that is open, customizable, and upgradeable.

That would make Solid State "Disk" storage much cheaper, much more

reliable, and much more customizable at a cost in hardware compatibility of course. SATA is all dead weight - other than the serial interface, it seems a gigantic step backward in the state of storage I/O technology.

SAS/SCSI is similarly over burdened, if not quite so backward as SATA. SATA is one of those "make the simple things simple, and the hard things impossible" sort of technologies".

Log-structured file systems: There's one in every SSD
Posted Sep 19, 2009 7:14 UTC (Sat) by **dlang** (★ **supporter** ★, #313) [Link]

one huge advantage of the SATA interface is that it works in any machine, it also doesn't require any special drivers for the OS.

it's also surprisingly complicated to make a PCI interface correctly. in many ways it's far easier to make a SATA device.

Log-structured file systems: There's one in every SSD
Posted Sep 19, 2009 15:23 UTC (Sat) by **butlerm** (subscriber, #13312) [Link]

The problem is that the SATA interface is technically defective in a number of respects, one of which is particularly bad for SSDs. The other problems are well described in the original article - a black box interface to obscure what is definitely not a black box. Switch manufacturers, and you could have entirely different performance characteristics in a manner that could take to months to evaluate.

The only way SATA/SAS physical interface would work well for SSDs is to develop an entirely new command set that allowed the off-loading of virtually all the intelligence to the host, i.e. presenting a flash memory interface rather than a disk interface over the serial bus. At that point you really couldn't call it a SATA "disk" any more, it would be more like a large capacity SATA "memory stick". That is the way it should be.

Log-structured file systems: There's one in every SSD
Posted Sep 19, 2009 19:55 UTC (Sat) by **dlang** (★ **supporter** ★, #313) [Link]

the only problem I have heard of is the fact that trim is not a NCQ command. what are the other problems?

the black box argument doesn't apply to a open DIY product like was being proposed.

I don't see why the host needs to address the raw flash. the problem is that there is currently zero visability to how the flash is being managed. if you had access to the source running on the device, why would you have to push all those details back to the OS?

Log-structured file systems: There's one in every SSD
Posted Sep 20, 2009 6:55 UTC (Sun) by **butlerm** (subscriber, #13312) [Link]

The other major problem is effective control over when blocks physically hit the platter, and which ones. There are no write barriers, the command to flush the cache can't be queued, and Force Unit Access write commands generally flush all other dirty data as well, making them slow to the degree that no one uses them. All this stuff is important for reliable operation of modern filesystems and databases, especially in portable devices.

I don't think do-it-yourself has anything to do with it - the question is

whether the person(s) concerned want to re-implement what other companies are already doing with no obvious advantage, or do something that could potentially run circles around current devices, if only due to the flexibility and performance characteristics of the interface. A single level filesystem ought to outperform one filesystem on top of another filesystem every time.

Log-structured file systems: There's one in every SSD
Posted Sep 20, 2009 8:48 UTC (Sun) by **nhippi** (subscriber, #34640) [Link]

It is just incredibly stupid to program flash memory with a very low-level interface designed for controlling spinning disks. You could extend ATA enough to give a raw flash (erase this block, then write with this data) or semi-raw (distribute write this block to nand chips 1,3,5,7, ...), but it still doesn't change the fact that the bus and protocol are engineered around spinning disks and overcoming its latency problems.

Other problems with sata (and pci) is that they are incredibly power-hungry busses.

While we have some interesting NAND flash-optimized code in linux kernel (UBIFS), microsofts dominant position in desktop market means they only appear in embedded systems. Instead of using flash-optimized free software filesystems, we attach a cpu on the flash chips to emulate the behaviour of a hard drive - just to keep microsoft operating systmems happy.

How much more evidence do people need that microsofts monopoly stiffles innovation?

Log-structured file systems: There's one in every SSD
Posted Sep 20, 2009 9:19 UTC (Sun) by **dlang** (★ **supporter** ★, #313) [Link]

keep in mind that the same hardware standardization that you are blaming microsoft for is the same thing that let linux run on standard hardware.

think about the problems that we have in the cases where hardware doesn't use standard interfaces, and the only code that is around to make things work is the closed-source windows drivers.

examples of this are the wifi and video drivers right now (just about everything else has standardized, and I believe that wifi is in the process)

another example s sound hardware, some of it is open, but some of it is not.

if you want other historic examples, go take a look at the nightmare that was CD interfaces before they standardized on IDE.

it's not just microsoft that benifits from this standardization, it's other opensource operating systems (eventually including whatever is going to end up replacing linux someday ;-)

as for your comments about the horrible storage interface designed for rotating disk. drives export their rotating disks as if they were a single logical array of sectors. they didn't use to do this, but nowdays that's what's needed. the cylinder/head/sector counts that they report are pure fiction for the convenience of the OS. what the interface does is lets you store and retrieve blocks of data. it doesn't really matter of those blocks are stored on a single drive, a raid array, ram, flash, or anything else.

Log-structured file systems: There's one in every SSD
Posted Sep 20, 2009 18:59 UTC (Sun) by **drag** (subscriber, #31333) [Link]

The problem is that memory devices are NOT block devices and you can't treat them

like block devices. Raw flash has "Erase Blocks", but that is just about were the similarities ends.

And I suspect, even though I have very little knowledge of SATA, that the ATA protocols used are going to be a very poor match for dealing with non-block devices.

Also on top of that people are talking about handling things that are now currently reserved to drive firmware... For example current harddrives reserve sectors to replace back sectors, right? Well in flash your going to have to record drive memory usage someplace on the device so that if you plug the drive into another OS or reinstall you don't loose your wear leveling information.

So probably the best thing to do is forgo the entire SATA interface and create a PCI Express card so you can create a new interface.

This should allow a much quicker and low-level interface to bypass any and all "black boxes".

The trick is if you want to have low-level access you need to be able to map portions of your flash memory to the machine's memory map so you can access the flash _as_memory_ and not as block devices.

And obviously if your dealing with a 250GB drive you probably don't want to map the entire drive to your system's memory (which I doubt is even possible to have a PCI Express thingy do that and still be able to take advantage of things like DMA). So you'll need some sort of sliding window mechanism in PCI configuration space so that the kernel can say "I want to look at memory addresses 0xX to memory 0xY relative to the flash drive"

Then your probably going to want to have multiple "sliding windows".. for example on a 4-lane PCI E device I would think its possible to be reading/writing to 4 different portions of the flash memory simultaneously. Sort of like hyperthreading, but with flash memory.

So for example if you can write to the drive at 200MB/s then having a 8-lane PCI Express memory device means that you can have a total write performance of 1600MB/s.

Then on top of that you'll want to have a special area of flash memory, with very small erase blocks, that you can store statistical information about each erase block of flash memory as well as real-to-virtual block mappings for wear leveling algorithms to use, and then extra space for block flags for future proofing.. probably in more expensive SLC flash memory so you don't have to worry about wear leveling and whatnot compared to the regular MLC-style flash that you'll use for your actual mass storage.

And of course you'll have to realize that if you want low-level access you can't use existing file systems for Linux.

So no BTRFS, no Ext3, no Ext4 or anything like that. The flash memory file systems for Linux are pretty worthless for large amounts of storage. You could use software memory-to-block translation to run BTRFS on top of it, but you lose the advantage of "one file system", although you still retain the ability to actually know what is going on and be able to do layer violations so that BTRFS's behavior can be modified to optimize flash memory access.

And then on top of that you have no Windows compatibility. So no formatting it as vfat or anything like that. Sure you could still use Linux's memory-to-block translation stuff, but that won't be compatible for Windows.

Log-structured file systems: There's one in every SSD
Posted Sep 20, 2009 19:43 UTC (Sun) by **cmccabe** (guest, #60281) [Link]

The problem is that once something becomes a standard, it's very hard to change it. It usually requires the major players to hold long meetings in standards organizations. A lot of horse trading can go on in these meetings. Sometimes companies even try to sabotage the process because they don't think that the proposed standard would help them.

For example, Intel is feeling good about NAND-over-SATA right now because they have one of the most advanced block emulation layers. They have a competitive advantage and they want to make the most of it. I would be surprised if they made any moves at all towards exposing raw flash to the operating system. It would not be in their best interest.

The big sticking point with any raw-flash interface is Windows support. Pretty much all

Windows users use NTFS, which works on traditional block devices. Any company hoping to replace NAND-over-SATA would have to supply a filesystem for Windows to use with their product of equivalent quality. Filesystems can take years to become truly stable. In the meantime all those angry users will be banging on *your* door.

Microsoft might create a log-structured filesystem for Windows UltraVista Aquamarine Edition (or whatever their next release will be called), but I doubt it. I just don't see what's in it for them. It's more likely that we'll see another one of those awkward compromises that the PC industry is famous for. Probably the next version of SATA will include some improvements to make NAND-over-SATA more bearable. And filesystem developers will just have to learn to live with having another layer of abstraction between them and the real hardware. Perhaps we'll finally get bigger blocks (512 is way too small for flash).

C.

Log-structured file systems: There's one in every SSD
Posted Sep 20, 2009 23:36 UTC (Sun) by **dlang** (★ **supporter** ★, #313) [Link]

it is very expensive to run all the wires to connect things via a parallel bus, that is why drive interfaces have moved to serial busses

trying to map your flash storage into your address space is going to be expensive, and it also isnt very portable from system to system.

it's already hard to get enough slots for everything needed in a server, dedicating one to flash is a hard decision, and low-end systems have even fewer slots.

there is one commercial company making PCI-E based flash drives, their cost per drive is an order of magnitude higher than the companies making SATA based devices, they also haven't been able to get their device drivers upstream into the kernel so users are forced into binary-only drivers in their main kernel. this is significantly worse than the SATA interface version because now mistakes in the driver can clobber the entire system, not just the storage device.

sorry, I don't buy into the 'just connect the raw flash and everything will be good'

sorry, I don't buy into the 'just connect the raw flash and everything will be good' reasoning.

Log-structured file systems: There's one in every SSD
Posted Sep 21, 2009 0:38 UTC (Mon) by **drag** (subscriber, #31333) [Link]

Ya. I was just trying to show what it would be like to try to access a large amount of flash memory in a 'raw mode'.

People are kinda confused, I think, about the whole block vs flash thing.

The way I see it Flash memory, in a lot of ways, is much more like memory then

block devices. Instead of thinking it as block devices with no seek time... think of it more like memory with pecular requirements that makes writes very expensive.

Log-structured file systems: There's one in every SSD
Posted Sep 21, 2009 1:03 UTC (Mon) by **dlang** (★ **supporter** ★, #313) [Link]

the thing is that even main memory access isn't really 'memory like' anymore.

on modern systems, it's impossible to directly access a particular byte of ram. the memory chips actually act more like tape drives, it takes a significant amount of time to get to the start position, then it's cheap to do sequential read/writes from that point forward.

your cpu uses this to treat your ram as if it was a tape device with blocks the size of your cpu cache lines (64-256 bytes each)

In addition, if you want to have checksums to detect problems you need to define what size the chunks of data are that you are doing the checksum over.

Log-structured file systems: There's one in every SSD
Posted Sep 21, 2009 9:10 UTC (Mon) by **nix** (subscriber, #2304) [Link]

> on modern systems, it's impossible to directly access a particular byte of ram. the memory chips actually act more like tape drives, it takes a significant amount of time to get to the start position, then it's cheap to do sequential read/writes from that point forward. your cpu uses this to treat your ram as if it was a tape device with blocks the size of your cpu cache lines (64-256 bytes each)

That's almost entirely inaccurate, I'm afraid. Ulrich Drepper's article on memory puts it better, in section 2.2.1.

The memory *is* necessarily read in units of cachelines, and it takes a significant amount of time to load uncached data from main memory, and of course it takes time to latch RAS and CAS, but main memory itself has a jagged access pattern, with additional delays from precharging and so on whenever RAS has to change.

But that doesn't make it like a tape drive, it's still random-access: it takes the same time to jump one row forwards as to jump fifty backwards. It's just that the units of this random access are very strange, given that they're dependent on the physical layout of memory in the machine (not machine words and possibly not cachelines), and are shielded from you by multiple layers of

> caching.

Log-structured file systems: There's one in every SSD
Posted Sep 21, 2009 8:30 UTC (Mon) by **butlerm** (subscriber, #13312) [Link]

PCI-E really isn't a parallel bus - it is multi-lane differential serial bus.
You can use a single lane if you want to - 500 MB/sec per lane today, 1 GB/s soon.

The good thing about PCI-E for this application is that you can use simple external cables, so you can easily locate your flash units in a different chassis than the CPU. External PCI-E connections are being used for external disk arrays already. *Much* faster than SAS with more than one lane.

Flash is not exactly a byte addressable memory technology, btw, so you still need to DMA to and from host memory.

Log-structured file systems: There's one in every SSD
Posted Sep 25, 2009 22:32 UTC (Fri) by **giraffedata** (subscriber, #1954) [Link]

> it is very expensive to run all the wires to connect things via a parallel bus, that is why drive interfaces have moved to serial busses

That's not why drive interfaces (and every other data communication interface in existence) have moved to serial. They did it to get faster data transfer.

But I'm confused as to the context anyway, because the ancestor posts don't mention parallel busses.

Log-structured file systems: There's one in every SSD
Posted Sep 25, 2009 22:57 UTC (Fri) by **dlang** (★ **supporter** ★, #313) [Link]

they described handling flash as if it was just ram with special write requirements.

that (at least to me) implied the need for a full memory bus (thus the lots of wires)

by the way, parallel buses are inherently faster than serial buses, all else being equal.

if 1 wire lets you transmit data at speed H, N wires will let you transmit data at a speed of NxH.

the problem with parallel buses at high speeds is that we have gotten fast enough that the timing has gotten short enough that the variation in the length of the wires (and therefor the speed-of-light time for signals to get to the other end) and the speed of individual transistors varies enough to run up against the timing limits.

Log-structured file systems: There's one in every SSD
Posted Sep 28, 2009 15:31 UTC (Mon) by **giraffedata** (subscriber, #1954)

[Link]

> They described handling flash as if it was just ram with special write requirements.
>
> That (at least to me) implied the need for a full memory bus (thus the lots of wires)

the lots of wires)

But the post described doing that with a PCI Express card. PCI-E is as serial as SATA.

> by the way, parallel buses are inherently faster than serial buses, all else being equal.
>
> if 1 wire lets you transmit data at speed H, N wires will let you transmit data at a speed of NxH.

It sounds like you consider any gathering of multiple wires to be a parallel bus. That's not how people normally use the word; for example, when you run a trunk of 8 Ethernet cables between two switches, that's not a parallel bus. A parallel bus is where the bits of a byte travel on separate wires at the same time, as opposed to one wire at different times. Skew is an inherent part of it.

crosstalk, not wire length
Posted Oct 2, 2009 0:21 UTC (Fri) by **gus3** (guest, #61103) [Link]

> if 1 wire lets you transmit data at speed H, N wires will let you transmit data at a speed of NxH.

That is true, when the bus clock speed is slow enough to allow voltages and crosstalk between the wires to settle. However, as clock speeds approach 1GHz, crosstalk becomes a big problem.

> the problem with parallel buses at high speeds is that we have gotten fast enough that the timing has gotten short enough that the variation in the length of the wires ... and the speed of individual transistors varies enough to run up against the timing limits.

Wire length on a matched set of wires (whether it's cat5 with RJ-45 plugs, or a USB cable, IDE, SCSI, or even a VGA cable) has nothing to do with it. The switching speed on the transmission end can accomplish only so much, but there has to be some delay to allow the signal to settle onto the line. The culprit is the impedance present in even a single wire, that resists changes in current. The more wires there are in a bundle, the longer it takes the transmitted signal to settle across all the wires. By reducing the number of wires, the settling time goes down as well.

Related anecdote/urban legend: On the first day of a new incoming class, RAdm Grace Hopper would hold up a length of wire and ask how long it was. Most of the students would say "one foot", but the correct answer was "one nanosecond."

crosstalk, not wire length
Posted Oct 2, 2009 17:15 UTC (Fri) by **giraffedata** (subscriber, #1954) [Link]

That's good information about transmitting signals on electrical wires, but it doesn't distinguish between parallel and serial protocols.

Crosstalk is a phenomenon on bundled wires, which exist in serial protocols too: each wire carries one serial stream. This wire configuration is common and affords faster total data transmission than parallel with the same number of wires

of wires.

Signals having to settle onto the line also happens in serial protocols as well as parallel.

Is there some way that crosstalk and settling affect skew between wires but not the basic signal rate capacity of each individual wire?

crosstalk, not wire length
Posted Oct 7, 2009 5:47 UTC (Wed) by **gus3** (guest, #61103) [Link]

I had to think about the crosstalk vs. skew issue for a bit, but I think I can explain it. (N.B.: IANAEE; I Am Not An Electronics Engineer. But I did work with one for a couple years, and he explained this behavior to me.)

Take an old 40-conductor IDE cable, for example. Typically, it's flat; maybe it's bundled. Each type creates its own issues.

A flat cable, with full 40-bit skew, basically means that the bit transmitted on pin 1, can't be considered valid until the bit on pin 40 is transmitted, AND its signal settles. Or, with an 8-bit skew, bits 1, 9, 17, 25, and 33 aren't valid until bits 8, 16, 24, 32, and 40 are transmitted.

(IIRC, an 80-conductor cable compensated for this, using differential signaling, transmitting opposite signals on a pin pair, using lower voltages to do so. This permitted less crosstalk between bits, while speeding the signal detection at the other end. But I could be wrong on this.)

A bundled 40-conductor cable is a little better. Think about an ideal compaction: 1 wire in the center, 6 wires around it, 12 around those, 18

around those, and 3 more strung along somewhere. From an engineering view, this could mean bit 1, then bits 2-7 plus settling time, then bits 8-19, plus settling time, then bits 20-37 plus settling time, then bits 38-40 plus settling time. (This from an iterative programmer's mind-set. A scrambled bundle might be better, if an EE person takes up the puzzle.)

Now, consider a SATA bus. Eight wires: ground, 2 differential for data out, ground, 2 differential for data in, ground, and reference notch. Three ground lines, with the center ground isolating the input and output lines. Add to this the mirror-image polarity between input and output; the positive wires are most isolated from each other, while the negative wires are each next to the middle ground wire. The crosstalk between the positive input and positive output drops to a negligible level, and the negative lines, near the center ground, serve primarily for error checking (per my best guess).

I hope my visualization efforts have paid off for you. Corrections are welcome from anyone. Remember, IANAEE, and my info is worth what you pay for it. This stuff has been a hobby of mine for over 30 years now, but alas, it's only a hobby.

Log-structured file systems: There's one in every SSD
Posted Sep 22, 2009 18:30 UTC (Tue) by **ttonino** (subscriber, #4073) [Link]

I's rather see the intelligence and the file system in the drive exploited to make an object level store out of the drive. In essence it would present a (possibly very limited) file system to build real file systems on top of.

The real file system layer can then more easily handle things like striping and mirroring, which would involve writing the same block with the same identifier to multiple drives.

Maybe the object level store could support the use of directories. These could be applied in lieu of partitions.

Deleting an object would obviously free the used flash.

One advantage could be that the size of each object is preallocated, and that data that belongs in an object can be kept together by the drive. The current situation is that a large write gets spread over multiple free logical areas, and the drive may have problems to guess that these will be later deleted as a single entity.

Log-structured file systems: There's one in every SSD
Posted Sep 20, 2009 20:48 UTC (Sun) by **dwmw2** (subscriber, #2063) [Link]

> *"keep in mind that the same hardware standardization that you are blaming microsoft for is the same thing that let linux run on standard hardware."*

Nonsense. Disk controllers aren't standardised — we have a multitude of different SCSI/IDE/ATA controllers, and we need drivers for each of them. You can't boot Linux on your 'standard' hardware unless you have a driver for its disk controller, and that's one of the main reasons why ancient kernels can't be used on today's hardware. Everything else would limp along just fine.

The *disks*, or at least the block device interface, might be fairly standard but that makes no real difference to whether you can run Linux on the system or not. It does help you share file systems between Linux and other operating systems, perhaps — but that's a long way from what you were saying.

NAND flash *is* fairly standard, although as with disks we have a multitude of different controllers to provide access to it. And even the bizarre proprietary things like the M-Systems DiskOnChip devices, with their "speshul" formats to pretend to be disks and provide INT 13h services to DOS, can be used under Linux quite happily. You don't need to implement their translation layer unless you want to dual-boot (although we *have* implemented it). You can use the raw flash just fine with flash file systems like JFFS2.

Log-structured file systems: There's one in every SSD
Posted Sep 20, 2009 23:28 UTC (Sun) by **dlang** (★ **supporter** ★, #313) [Link]

now linux has support for a huge number of devices.

but think about when linux started. at that point in time it started out with ST506 MFM support and vga video support.

that wasn't enough to drive the hardware optimally, but it was enough to talk to it.

similarly with the IDE/SATA controllers, most of them work with generic settings, but to get the most out of them you want to do the per-controller tweaks.

even in video, the nvidia cards can be used as simple VGA cards and get a display.

the harder you make it to get any functionality the harder it is to get to the point where the system works well enough to be used and start being tuned

the system works well enough to be used and start being tuned.

DIY SSD form factor
Posted Sep 19, 2009 13:11 UTC (Sat) by **pflugstad** (subscriber, #224) [Link]

For a dev system, you probably wouldn't need to conform to the hard drive form factor. Most kernel devs probably wouldn't care if it was a (small) pile of (quiet) equipment on their desk. Or even a DVD/CD-ROM sized form factor.

Log-structured file systems: There's one in every SSD
Posted Sep 19, 2009 0:22 UTC (Sat) by **quozl** (guest, #18798) [Link]

... build an SSD from a bunch of flash ... how big would it have to be to

be useful to 'see how it could be done?' ... is a gigabyte enough? you could take an OLPC XO-1, use the NAND flash therein, with jffs2 ... and 1.3 million of this configuration have been deployed. ;-)

Log-structured file systems: There's one in every SSD
Posted Sep 19, 2009 19:53 UTC (Sat) by **dwmw2** (subscriber, #2063) [Link]

Indeed. And the kernel already has more than one FTL implementation, which lets you do this silly fake-disk thing on top of raw flash and then use a 'standard' file system on top of that, just like an SSD does.

Log-structured file systems: There's one in every SSD
Posted Sep 19, 2009 20:35 UTC (Sat) by **dlang** (★ **supporter** ★, #313) [Link]

part of the reason for doing a open/DIY SSD would be to shame the manufacturers of the commercial ones into improving them.

creating a linux specific software only item won't do that (and because the hardware would then be linux-only it would be more expensive due to the limited production run)

it's also hard to test how a linux software solution would work for a windows or mac access pattern.

Log-structured file systems: There's one in every SSD
Posted Sep 18, 2009 21:02 UTC (Fri) by **Yorick** (subscriber, #19241) [Link]

Does this mean that file systems that avoid overwriting live data ZFS and the like are at a disadvantage with SSDs, assuming the firmware algorithms were tuned for NTFS and with overwriting in mind? If the file system prefers writing updates elsewhere as long as there is unused space, I can imagine that will cause trouble for the poor FTL.

Log-structured file systems: There's one in every SSD
Posted Sep 18, 2009 21:36 UTC (Fri) by **ikm** (subscriber, #493) [Link]

Yes -- but that stops to matter the moment you fill up your NTFS with data completely just once. After that there would be no difference whatsoever -- all the blocks are marked as used anyway.

I guess there are some people out there who never ever happen to use their drive to full capacity (imagine a person using Word to type documents and IE to surf the web, and doing nothing more than that) -- I guess that's what Valerie has been referring to. In-place filesystems should be beneficial to such kind of people, of course.

Btw, I strongly happen to believe that most SSDs would mark blocks as free if they are written as zeroes -- just because full NTFS format would mark all blocks as used otherwise

zeroes -- just because full NTFS format would mark all blocks as used otherwise.

Log-structured file systems: There's one in every SSD
Posted Sep 21, 2009 16:14 UTC (Mon) by **nye** (guest, #51576) [Link]

I wondered about this, but how would it know that you didn't really want to store all zeroes in that block? When you read back from it you might find that it's been re-used in the meantime and filled with something else, or more likely the SSD firmware would complain that the (logical) block you are requesting doesn't currently map to any (physical) block.

I'm getting pretty tired this afternoon - am I missing something obvious?

Log-structured file systems: There's one in every SSD
Posted Sep 21, 2009 16:35 UTC (Mon) by **farnz** (guest, #17727) [Link]

It takes a little intelligence on the part of the SSD designer. Reads from a logical block not currently mapped to any physical flash must be defined as returning all zeros; once you've done that, you can treat writing a logical block with all zeros as a "free" operation, not a write. The "free" operation just marks that logical block as not mapped to any physical flash.

You then know you don't need to hang onto the data that used to be important to that logical block; initially, it's still there, but when you garbage collect the physical block that used to contain the bytes from that logical block, you don't bother copying the data.

Log-structured file systems: There's one in every SSD
Posted Sep 21, 2009 20:22 UTC (Mon) by **ikm** (subscriber, #493) [Link]

> I wondered about this, but how would it know that you didn't really want to store all zeroes in that block?

Because it's the same thing. There's no point in actually storing zeroes -- you could just mark the block as unused instead, so if later read, it would read back as zeroes.

> When you read back from it you might find that it's been re-used in the meantime and filled with something else

SSD maintains a mapping between logical disk blocks and physical flash blocks. If a logical block is marked as unmapped, it just doesn't use any physical space at all. SSD can't reuse your logical block since it doesn't "use" logical blocks -- it uses physical flash blocks.

> the SSD firmware would complain that the (logical) block you are requesting doesn't currently map to any (physical) block.

It would return zeroes, since the block is empty (unmapped). And that's the intent of it.

Log-structured file systems: There's one in every SSD
Posted Sep 22, 2009 11:08 UTC (Tue) by **nye** (guest, #51576) [Link]

This is indeed obvious now that I'm awake (somewhat embarrassingly so :P).
I think the bit that I was somehow missing was that 'reading' from an unmapped block should just return all zeroes.

But thanks to both for making it explicit.

Log-structured file systems: There's one in every SSD
Posted Oct 6, 2009 16:45 UTC (Tue) by **sethml** (subscriber, #8471) [Link]

The problem with this approach is that transferring zillions of zeros over the disk interface is slow. Imagine if deleting a 10gb file took several minutes - that would be rather annoying.

Too bad SATA doesn't have a "write a million zeros" command. Of course, that's effectively what a proper TRIM would do.

Log-structured file systems: There's one in every SSD
Posted Oct 6, 2009 19:59 UTC (Tue) by **ikm** (subscriber, #493) [Link]

True. But it still might be better than the current TRIM implementation -- if not performance-wise, then at least compatibility-wise. The blocks could be zeroed in background when the I/O is otherwise idle.

Log-structured file systems: There's one in every SSD
Posted Sep 18, 2009 22:02 UTC (Fri) by **paravoid** (subscriber, #32869) [Link]

That's kind of interesting, considering that Sun's OpenStorage solution (which uses ZFS) uses SSDs for caches -- which I presume has them at 100% full capacity.

Log-structured file systems: There's one in every SSD
Posted Sep 19, 2009 6:44 UTC (Sat) by **butlerm** (subscriber, #13312) [Link]

If want to use SSDs for caching the last thing you want is to keep them at 100% capacity - it takes too long to kick things out of the cache, just when you need to put something new in.

Log-structured file systems: There's one in every SSD
Posted Sep 22, 2009 15:15 UTC (Tue) by **k8to** (subscriber, #15413) [Link]

I think the idea is they are physically at 100% capacity, even if not logically so. That is, they've probably hit full at some point.

Well, that's my guess, since it makes the most sense.

Log-structured file systems: There's one in every SSD
Posted Sep 25, 2009 14:38 UTC (Fri) by **knweiss** (subscriber, #52912) [Link]

FWIW they have separate read and write caches (both SSD).

Log-structured file systems: There's one in every SSD
Posted Sep 20, 2009 8:17 UTC (Sun) by **jzbiciak** (★ **supporter** ★, #5246) [Link]

Nitpicking on an otherwise excellent article:

> *This requires clearing an erase block, moving all the in-use blocks to another area,*
>
> *and keeping a mapping between the logical location of blocks and their physical locations - exactly what a log-structured file system does.*

I really do hope that the in-use blocks get moved before the erase block gets cleared. ;-) The only other detail I'd nitpick about is that an erase block can generally get filled incrementally. At least, that's been my experience with embedded flash.

Where that last detail's important: One doesn't need to gather many kilobytes of writes *a priori* before scheduling an erase block for erasure. Rather, once an erase block is erased, all of the next several kilobytes of writes should go to the block. You're committing the direction of future writes.

I took particular interest in this article since I'm designing a special purpose "file system" for a flash-based microcontroller. The flash in the microcontroller can't tolerate very many rewrite cycles (100 worst case, 1000 nominal), so I'm relying heavily on wear leveling, error correction and having way more physical

cells than data stored to make my application work. (I'm storing around 1.5K bytes of dynamic data in about 100K of space, which helps. And yes, my approach looks like a giant log, and if the most recent save ends up corrupt when I eventually read it, I plan on rolling back to the previous, since its right there in the log.)

And on a different note: I personally think that the absurd performance of TRIM (10s or 100s of milliseconds? Really?) on some devices is just criminal. Better to just not support TRIM if it's going to perform so badly. I can't wait until manufacturers get it right. I really want to go SSD on my laptop, but only once the technology gets sane.

As far as how TRIM should be treated: I agree filesystems should treat it like alloc()/free(). At the same time, we should also allow for some free() calls that didn't get committed in the case of system failure. A trim request doesn't have the same correctness requirements as, say, a journal update. So, in some sense, it makes sense to allow them to be lossy near a crash boundary.

I'm up a little too late to think this through completely, but I think there are cases where you can't be 100% sure that you've executed all of your TRIMs or recorded all of your intents to TRIM before you've committed all the necessary actions to free a write block. If I'm correct (and I'd lay even odds that I'm not, at this point), that means that you'll need some mechanism to scan and TRIM in the background on an unclean unmount, even with an otherwise airtight filesystem.

> **Log-structured file systems: There's one in every SSD**
> Posted Sep 22, 2009 5:22 UTC (Tue) by **butlerm** (subscriber, #13312) [Link]
>
> Apparently Micron's flash chips have the ability to internally move data around without having it leave the chip. No doubt very useful in this application.
>
> > **Log-structured file systems: There's one in every SSD**
> > Posted Sep 23, 2009 0:48 UTC (Wed) by **dwmw2** (subscriber, #2063) [Link]
> >
> > Not really.
> >
> > The 'read and then reprogram elsewhere from internal buffer' facility is all very well in theory, but your ECC is *off-chip*. So if you want to be able to detect and correct ECC errors as you're moving the data, rather than allowing them to propagate, then you need to do a proper read and write instead.
> >
> > Linux has never bothered to use the 'copy page' operation on NAND chips which support it.

Log-structured file systems: There's one in every SSD
Posted Sep 20, 2009 19:07 UTC (Sun) by **cmccabe** (guest, #60281) [Link]

Excellent article, Val!

> Once every block in the device has been written at least once,
> the SSD is doomed to a worst case performance state in which its
> spare blocks are at a minimum and data must be moved each time a
> new block is rotated into use.

I wonder if the manufacturers include "extra" flash chips to get around this problem. For example, if the device had 1.1 GB of flash, but reported 1 GB, it would never run into the worst-case scenario.

Colin

> **Log-structured file systems: There's one in every SSD**
> Posted Sep 20, 2009 23:37 UTC (Sun) by **dlang** (★ supporter ★, #313) [Link]

Posted Sep 20, 2009 23:37 UTC (Sun) by **dlang** (★ **supporter** ★, #313) [Link]

we know that they do include more flash than they report, but it still takes a significant amount of time to erase it, so the problem doesn't quite hit the worst-case scenerio, but it's still not pretty.

Is there a better way to use flash memory?
Posted Sep 21, 2009 6:19 UTC (Mon) by **PaulWay** (★ **supporter** ★, #45600) [Link]

One thing I've wondered with flash memory is whether there's a better way of using it than 'blank the entire block when you write it'. If one imagines every bit in a byte being represented by the parity on a byte of flash - so that 11111111 has parity 0, 11111110 has parity 1, etc. - then toggling that bit is equivalent to zeroing out one more bit from the flash byte. There's probably other, much more intelligent algorithms for spreading one source byte out over multiple target bytes such that a change of value in the source is simply a process of zeroing out (or one-ing out) certain bits in the target.

Another approach might be to treat each block as a miniature log - a megabyte block on flash equals 64K (call it a 'chunk') of filesystem. Each time you write to that 64K chunk you copy the chunk into the next available 64K of space on the block. When the block is full, you flush it and rewrite the chunk at the start of the block. In this way, sixteen writes to the chunk can occur before you have to rewrite the whole block, reducing the wear on that block (and thus the chance that that chunk of filesystem will fail). An optimal size might be 1KB chunks per 1MB block, where the first 1KB is used in the 'parity' fashion above to notate where the current 1KB is in that block, giving you 1023 possible rewrites before you have to flush the block and rewrite it from scratch.

As many people in this article's comments and elsewhere have said, flash memory is incredibly cheap. I think there's a body of people who would pay more for solid state disk space for it to behave with exactly the same MTBF characteristics as a regular spinning-rust hard disk.

Have fun,

Paul

Is there a better way to use flash memory?
Posted Sep 21, 2009 6:40 UTC (Mon) by **dlang** (★ **supporter** ★, #313) [Link]

I've had (and posted) thoughts along the same lines. the last time I posted them one person responded that checksums and ecc codes could prevent writing to small portions of flash

I think it's an idea worth investigating (it could trade some space for reduced erase cycles, being especially effective in metadata), but it would require either a smart drive (that doesn't move a block if it can be modified in place from the current to the desired value) or raw access to the flash.

Is there a better way to use flash memory?
Posted Sep 22, 2009 15:55 UTC (Tue) by **jzbiciak** (★ **supporter** ★, #5246) [Link]

My understanding and experience is that flash is rather similar to EPROM. You erase the entire erase block, sending it to all 1s. This is an indivisible operation—the whole block gets clobbered, and there's no way to clobber only a section of it. Then, over whatever period of time is convenient to you, you fill in sections of that erase block with live data. The size of the section you have to fill in at a time is governed by the width of the memory, since a programming pulse has to be applied for all of the bits across the width of the memory, but you only have to program one row. So, erasure erases a group of rows, and then you can fill the rows in at your leisure.

If your ECC lives within the the row as your data, then your ECC encoding doesn't really matter. Since row writes are atomic, the fact that ECC bits toggle back and forth as you monotonically clear 1s to 0s in your data bits doesn't matter. You have to present your data and ECC in parallel when you write the row. Typical ECCs such as Reed-Solomon are built around this block principle.

(Now here's where I don't know how similar EPROMs and flash are: You could keep reprogramming the same row as long as you only flip 1s to 0s, which is where your initial idea becomes relevant. At least one flash-based embedded device I've used tells me to never program a row more than twice without an intervening erase, which suggests there may be an issue with storing too much charge on the floating gate, which in turn could physically damage the gate. That charge is what makes a 1 turn into a 0. Old school EPROMs were a bit more durable in this regards. But, then, you also blast them with bright UV for 15-30 minutes to erase them.)

If the rows are fine enough granularity, you could in theory encode the data, a version number and an ECC in that row, and do some sort of delta-update. If only a few bytes in a block changed, there's no reason to store an entire new copy of the whole block. Only store the changed rows. This would provide great compression for certain types of updates, such as appending to a file or doing filesystem metadata updates (ie. ext2 block-bitmap updates, where only some of the bits in the bitmap flip).

If you also included an internal map that hashed all the data rows into a reverse map database, you could use that to quickly collapse all of the identical rows across the entire drive into a single row. That is, whenever you decide to go store a particular row of data, find out of that row already exists on the physical media and instead point to that. For typical storage patterns (ie. lots of similar text across many files due to duplicated files, lots of end-of-block empty fill, etc.), this could result in a huge on-disk savings. That savings would then directly translate to a larger erase block pool for the same apparent loading vs. advertised capacity.

Is there a better way to use flash memory?
Posted Sep 22, 2009 16:00 UTC (Tue) by **jzbiciak** (★ **supporter** ★, #5246) [Link]

Oh, and I forgot to mention: The width of a row could actually be pretty wide for a large storage array in a single chip. One embedded microcontroller I use has a row width of 192 bytes. (It's a 24-bit wide memory, hence the weird number.) I could imagine the row being much, much wider in a higher density flash such as what SSDs are made of.

Still, redundant row compression seems like an interesting idea to me. I'm not sure what you'd store the reverse map in, though, to make it effective, since that too needs to be stored somewhere non-volatile. This is where having a multi-tiered storage setup (volatile RAM + non-volatile RAM + flash) could be really interesting as compared to just having a PC + flash.

Log-structured file systems: There's one in every SSD
Posted Oct 1, 2009 11:44 UTC (Thu) by **victusfate** (guest, #61091) [Link]

So I joined LWN.net just so I could comment on this post. First off as a new timer to the hardware end of things, this post was not only informative but available to new readers. I'm more curious now about memory handling than ever before.

As a long time coder, my memory concerns ended with mallocs/frees/news/deletes and eventually I just forgot them inside of other high data structures that cleaned themselves up when out of scope.

Do you think it would be possible to write something similar up for NTFS or other formats? Is this article strictly unix centric so that windows SSD formats have their own particulars. I'd love to enhance a goofy squidoo lens I wrote up about SSDs a while back with REAL detailed information. (here's the link for the curious http://www.squidoo.com/KingstonSSDNowSolidStateHardDrive)

I keep thinking about the guy that had his OS running on two dozen or so SSD raid 0 (youtube video: http://www.youtube.com/watch?v=96dWOEa4Dis)

SSD raid 0 (youtube video: http://www.youtube.com/watch?v=96dWOEa4Dj8)

Flash Translation Layer
Posted Oct 1, 2009 21:50 UTC (Thu) by **flash-translation-layer** (guest, #61101) [Link]

Zeeis **Flash Translation Layer** have integrated in over 160 million flash based devices (SSDs, TransFlash Cards, SD cards, CF Cards, USB flash drives, MP3 players and mobile phones) and over 62% market share in China, 2008.

> **Flash Translation Layer**
> Posted Oct 5, 2009 14:10 UTC (Mon) by **tmassey** (guest, #52228) [Link]
>
> Did someone just create an account to spam LWN with? I don't think I've ever seen this before. Given the name of the account and the content of the message, it sure seems so...
>
> Wow, that's dedication to marketing your product! At least it's on-topic--better than V1@GR@ blog spamming! :)
>
> > **Flash Translation Layer**
> > Posted Oct 5, 2009 23:39 UTC (Mon) by **flash-translation-layer** (guest, #61101) [Link]
> >
> > I was developing SSDs and flash-based file system, so I am very interested in this article.
> >
> > SSD is currently a small but growing industry, there are many flash-based devices (SSDs, SD cards, USB drives) are designed & manufactured in China, also contains the IC controllers.
> >
> > Therefore I want this article reader to know the flash-based file system usage situation.
> >
> > > **Flash Translation Layer**
> > > Posted Oct 6, 2009 0:13 UTC (Tue) by **nix** (subscriber, #2304) [Link]
> > >
> > > Your "use our product" might have worked better if you were providing raw flash for the semi-mass-market (i.e. LWN readers, who are willing to pay over the odds, but not hugely).
> > >
> > > However, you're providing... an apparently closed-source FTL. The very thing that the article you followed up to is (rightly) criticising. (Of course if it *is* open source, well, that's nicer but we still need raw flash to use it with.)
> > >
> > > Pardon me for thinking that you didn't read it very carefully.

Log-structured file systems: There's one in every SSD
Posted Oct 2, 2009 17:58 UTC (Fri) by **djcapelis** (subscriber, #53964) [Link]

IIRC some of the next generation SSDs based on phase-change memory and some other things don't require any of this madness.

Samsung is doing phase-change memory at scale now. The capacity is smaller than we'd like, but it's actually here and in production finally.

(To be fair write cycles on PCM aren't infinite, but at 100mil cycles and writes on the bit level instead of the block level, PCM is a good deal that likely to make SSDs less annoying in the future. PCM isn't the only type of new SSD that's coming out with this stuff.)

> **Log-structured file systems: There's one in every SSD**
> Posted Oct 5, 2009 20:48 UTC (Mon) by **joern** (subscriber, #22392) [Link]
>
> When talking to hardware people who want to market PCM, you may notice that it suffers a similar

When talking to hardware people who want to market PCM, you may notice that it suffers a similar problem as flash SSDs do. Since there is no PCM software stack, they want to plug into an existing software stack by pretending to be something else. And from what I've heard so far, that something else will be NOR flash.

Which isn't too bad an idea, honestly. 100M may seem big, but if you are ignorant enough and write your filesystem superblock on every sync, you can have that worn out in just 24h. So you still need some amount of wear leveling.

Plus, the programming of PCM is asynchronous. Flipping a bit one direction is about 6x slower than flipping it the other way. Which means that by treating your random-writeable PCM as block-eraseable flash you gain a speedup that can more than counter the slowdowns from garbage collection under fairly realistic conditions.

> **Log-structured file systems: There's one in every SSD**
> Posted Oct 7, 2009 7:17 UTC (Wed) by **mcortese** (guest, #52099) [Link]
>
> > *Plus, the programming of PCM is asynchronous.*
>
> I guess you meant asymmetric?
>
> > **Log-structured file systems: There's one in every SSD**
> > Posted Oct 7, 2009 11:27 UTC (Wed) by **joern** (subscriber, #22392) [Link]
> >
> > Of course I did. Thanks.