Introduction

An alternative to the long-dominant hard disk drive has recently achieved global significance. Unlike hard drives, solid-state storage systems are made entirely of transistors, just like memory and processors. Unlike conventional random-access memory (RAM), a solid-state storage device (SSD) preserves data even when power is lost, making it an ideal contender for long-term data storage.

Flash has certain unique properties. For example, to write to a flash page, you must first erase a larger piece (a flash block), which might be costly. Writing on a page too often will wear it out. These two features make building a flash-based SSD a fun challenge:

HOW TO MAKE A FLASH-BASED SSD

How can we make a flash SSD?
How can we deal with erasing's high cost?
Given that repeated overwriting will wear down a device, how can we build one that lasts for a long time?

Storage: From Bits to Banks/Planes

Each transistor in a flash chip stores one or more binary values mapped to the level of charge trapped therein. The charge levels of 00, 01, 10, and 11 represent low, slightly low, moderately high, and high levels, respectively, in a single-level cell (SLC) flash. TLC flash encodes 3 bits each cell. SLC chips are faster and more expensive.

A single bit (or a few) does not a storage system create. Flash chips are therefore grouped into big banks or planes of cells.
In a bank, data is accessed in two sizes: blocks (sometimes called erase blocks) of 128 KB or more, and pages of a few KB (e.g., 4KB). Each bank has several blocks, and each block has many pages. Flash is a new term that is distinct from disk blocks, RAID blocks, and virtual memory pages.

INSERT IMAGE

Figure 44.1 depicts a flash plane with blocks and pages; there are three blocks, each with four pages. We'll learn why distinguishing between blocks and pages is important for flash activities like reading and writing, and much more so for overall device speed. What you'll learn is that to write to a page within a block, you must first erase the entire block; this tough aspect makes developing a flash-based SSD an enjoyable and worthy effort.

Basic Flash Operations

Flash chips are thought of as having a state associated with each page. Pages begin as INVALID. A page (and all pages within it) are erased when its block is erased, which resets their content but also makes them programmable. Its status turns to VALID when programmed, signifying its contents are set and may be read. State-independent reads (although you should only read from pages that have been programmed). Once a page is programmed, the only way to modify its content is to remove the entire block. In a 4-page block, the following states change after various erase and program operations:

INSERT IMAGE

With this flash arrangement, a flash chip can handle three low-level processes. The read command reads a page from the flash; the erase and program commands write.

- Read (a page): A flash chip client can read any page (e.g., 2KB or 4KB) by sending the read instruction and the appropriate page number. This action is often very quick, tens of microseconds or less, regardless of device location or previous request location (quite unlike a disk). A random access device can swiftly access any place.

- Erase (a block): Before writing to a page in a flash, you must first erase the block it is contained in. To avoid losing data, you must ensure that it has been moved to another location (memory or another flash block) before issuing the erase command. The wipe instruction takes a few milliseconds to complete. The block is then reset and each page is ready to program.

- Program (a page): After erasing a block, the program command can be used to convert some 1s to 0s and write the desired page contents to the flash. Programming a page is faster than deleting a block, but slower than reading a page on current flash devices.
  Programming a page is faster than deleting a block, but slower than reading a page on current flash devices.

Flash chips are thought of as having a state associated with each page. Pages begin as INVALID. A page (and all pages within it) are erased when its block is erased, which resets their content but also makes them programmable. Its status turns to VALID when programmed, signifying its contents are set and may be read. State-independent reads (although you should only read from pages that have been programmed). Once a page is

programmed, the only way to modify its content is to remove the entire block. In a 4-page block, the following states change after various erase and program operations:

INSERT IMAGE

Example

Because writing (i.e., erasing and programming) is so unusual, let's walk through an example. The following four 8-bit pages (both unreasonably short, but useful in this case) are VALID, having been previously programmed.

INSERT IMAGE

Let us now write to page 0, adding fresh content. To start writing, we must first erase the block. Assume we do, leaving the block as follows:

INSERT IMAGE

News! We might now program page 0 with contents 00000011, overwriting the original page 0 (contents 00011000). Now our block looks like this:

INSERT IMAGE

Worse, all previous contents of pages 1, 2, and 3 are gone! So, before overwriting any page within a block, we must relocate any important data elsewhere (e.g., memory, or elsewhere on the flash). As we'll see, the nature of erase has a big impact on how we design flash-based SSDs.

Summary

To summarize, simply read the page. Flash chips accomplish this well and quickly, perhaps outperforming current disk drives, which are slow due to mechanical search and rotation costs.

Writing a page requires first erasing the entire block (moving any data we care about first), then programming the desired page. This is not only costly, but it can cause flash chips to wear out. Writing performance and reliability are critical when constructing a flash storage system. We'll soon see how contemporary SSDs overcome these constraints to deliver exceptional performance and reliability.

Flash Performance And Reliability

Because we want to develop a storage device from raw flash chips, we should know their basic performance characteristics. Figure 44.2 summarizes some numbers. On the other hand, SLC flash stores 1, 2, or 3 bits of data per cell and MLC flash stores 1, 2, or 3 bits of data per cell.

As shown in the table, read latencies are excellent, taking only a few microseconds. Program latency increases with bit density, from 200 microseconds for SLC to 700 microseconds for SLC+, thus you'll need to employ multiple flash chips in parallel to get acceptable write performance. Finally, erasing takes a few milliseconds. Modern flash storage architecture deals with this cost.

Now examine flash chip dependability. Comparatively, flash chips are pure silicon and hence have less reliability difficulties than mechanical disks, which can fail for a multitude of reasons (including the grisly and physical head crash). The main worry is wear; each time a flash block is wiped and programmed, it adds a little charge. With time, the added charge makes it impossible to distinguish between a 0 and a 1. The block becomes unusable when it becomes impossible.

The average block lifetime is unknown. A 10,000 P/E (Program/Erase) cycle lifetime is given to MLC-based blocks by manufacturers. SLC-based devices have a longer lifetime, usually 100,000 P/E cycles, because they only store one bit per transistor. Recent research, however, suggests that lifetimes are longer than expected.

Disturbance is another flash chip reliability issue. When reading or programming a page in a flash, it is conceivable that some bits in surrounding pages become flipped. These bit flips are known as read disturbs or program disturbs.

From Raw Flash to Flash-Based SSD

Now that we know what flash chips are, we need to figure out how to make them look like a storage device. The standard storage interface reads and writes blocks (sectors) of 512 bytes (or bigger) depending on a block address. The flash-based SSD's job is to provide a standard block interface on top of the raw flash chips.

An SSD is made up of flash chips (for persistent storage). An SSD also incorporates volatile memory (e.g., SRAM), which is used for data caching, buffering, and mapping tables (more on that later). It also comprises control logic for device operation. Figure 44.3 is a simplified block diagram (page 7).

This control mechanism converts client reads and writes into internal flash operations as needed. The Flash Translation Layer (FTL) does exactly that. The FTL converts logical block read and write requests into low-level read, erase, and program commands on underlying physical blocks and pages (that com- prise the actual flash device). The FTL should do this with great performance and reliability.

As we'll see, combining approaches results in excellent performance. One essential will be to employ numerous flash chips in tandem; we won't go into detail about this, but all modern SSDs use multiple chips inside to improve performance. Another performance target is to limit write amplification, which is defined as the FTL's total write traffic (in bytes) divided by the client's total write traffic (in bytes). As we'll see, naive FTL design leads to significant write amplification and low performance.

Several methods will be combined to attain high reliability. One major worry is wear out. The FTL should aim to spread writes throughout the blocks of the flash as evenly as possible, ensuring that all of the blocks wear out at around the same time. This is termed wear leveling and is a fundamental feature of any modern FTL.

Program disruption is another issue. To avoid disruption, FTLs generally program pages within an erased block in ascending sequence. This approach reduces disruption and is extensively used.

FTL Organization

An FTL can be di-rect mapped for simplicity. A read to logical page N corresponds to a read to physical page N. A write to logical page N requires the FTL to first read in the full block containing page N, then erase the block, and then program both the old and new pages.

As you would imagine, direct-mapped FTL has significant performance and reliability issues. Each write causes performance issues since the device must read in the complete block, erase it, and then program it (costly). With mechanical searches and rotational delays, this causes significant write amplification (proportional to the number of pages in a block) and poor write performance.

Worse, this method is unreliable. The same block is erased and programmed again, rapidly wearing it out and potentially losing data. It gives the client workload too much control over wear out; if the workload doesn't distribute write load fairly among its logical blocks, the underlying physical blocks with popular data quickly wear out. A direct-mapped FTL is unreliable and slow.

Log-Structured FTL

The principle is useful in storage devices (as we'll see) and file systems above them (as we'll see in the log-structured files chapter). A write to logical block N is appended to the next free location in the currently-written-to block; this is called logging. That is, the device preserves a mapping table (in memory, and persistent in some manner) that stores the physical addresses of each logical block in the system.
Let's look at an example to see how the fundamental log-based technique works. Clients perceive it as a normal disk that can read and write 512-byte sectors (or groups of sectors). Assume the client is reading or writing 4-KB chunks. Assume the SSD has a huge number of 16-KB blocks, each divided into four 4-KB pages; these specifications are impractical (flash blocks frequently have more pages), but they will enough for our purposes. Assume the customer performs the following steps:

• Write(100)withcontentsa1
• Write(101)withcontentsa2
• Write(2000)withcontentsb1
• Write(2001)withcontentsb2

The SSD client (e.g., a file system) uses these logical block addresses (e.g., 100) to remember information.
Internally, the device must convert these block writes into raw hardware erase and program operations, and record which physical page of the SSD each logical block address corresponds to. Take the SSD as a blank slate, requiring all blocks to be wiped before programming. Its initial state is seen here, with all pages marked INVALID (i)

INSERT IMAGE

The FTL decides to write the first write (to logical block 100) to physical block 0, which comprises four physical pages: 0, 1, 2, and 3. We can't write to it yet because it's not erased; the device must first delete block 0. As a result,

INSERT IMAGE

Block 0 is now programmed. Most SSDs write pages in order (low to high), decreasing program disruption difficulties. The SSD writes logical block 100 to physical page 0:

INSERT IMAGE

Suppose the client requests logical block 100. How can it locate itself? The SSD must convert a logical block 100 read to a physical page 0 read. To support this, the FTL writes logical block 100 to physical page 0 and stores the information in an in-memory mapping table. The diagrams will also show the state of the mapping table:

INSERT IMAGE

Now you can see the client writing to the SSD. That page is then loaded with the block's contents and the logical-to-physical mapping is stored in the SSD's mapping table. Reading data requires a physical page number, which is determined by the logical block address given by the client.
Examine now writes 101, 2000, and 2001 in our example write stream. After writing these blocks, the device states:

INSERT IMAGE

The log-based technique increases performance (rarely requiring erasures, and avoiding the costly read-modify-write of direct-mapped), and considerably improves reliability. The FTL can now spread writes across all pages, enhancing device lifetime and reducing wear.
Sadly, this fundamental log structuring strategy has several drawbacks. The first is that overwriting logical blocks causes garbage, i.e. old versions of data cluttering up the drive. To discover said blocks and free space for future writes, the device must periodically perform garbage collection (GC). The second is the high expense of in-memory mapping tables; larger devices require more memory. Let's go over each one.

Garbage Collection

Using a log-structured technique like this creates garbage, which means dead-block reclamation is required. Let us take our previous example to clarify. Remember that the device has logical blocks 100, 101, 2000, and 2001.
Assume now that blocks 100 and 101 have been rewritten with c1 and c2. When writing to free pages (in this case, 4 and 5), the mapping table is updated. To program the device, the device must first delete block 1:

The issue now is obvious: although being designated VALID, physical pages 0 and 1 contain junk, i.e. old copies of blocks 100 and 101. Because the device is log-structured, overwrites create garbage blocks that must be reclaimed to allow new writes.
Garbage collection is the process of discovering and reclaiming dead blocks for future usage. The basic approach is to discover a block with garbage pages, read in the live (non-junk) pages, log them, and then recover the full block for writing.
Let us now use an example. It decides to reclaim any dead pages in block 0. Pages 0 and 1 are dead blocks in block 0. (pages 2 and 3, which contain blocks 2000 and 2001, respectively). To accomplish so, it will:

• Read live data (pages 2 and 3) from block 0
• Write live data to end of the log
• Erase block 0 (freeing it for later usage)

For the garbage collector to work, each block must have enough data to detect whether a page is alive or dead. One natural approach to achieve this is to store information about which logical blocks are contained within each page within each block. The device can then utilize the mapping table to identify whether or not each page contains live data.
In our previous example, block 0 contained logical blocks 100, 101, 2000, and 2001. The device can quickly detect whether each page in the SSD block contains live data by checking the mapping table (which contained 100->4, 101->5, 2000->2, 2001->3). For example, the map clearly shows 2000 and 2001, but not 100 and 101, which are possibilities for rubbish pickup.
In our case, the gadget is in the state:

To collect garbage requires reading and rewriting of live data. A block of solely dead pages is perfect for reclamation since it may be wiped and reused for fresh data without costly data migration.

Because certain SSDs overprovision the device, cleaning might be postponed and done in the background, when the device is less busy. Increasing capacity increases internal bandwidth, which can be used for

cleaning without affecting customer perception. Modern drives overprovision to achieve excellent overall performance.

Block-Based Mapping

One entry for each 4-KB page of the device is the second cost of log-structuring. For a 1-TB SSD, a single 4-byte entry per 4-KB page requires 1 GB of memory just for mappings! So this page-level FTL technique is useless.

Keeping a pointer per block of the device instead of every page can reduce mapping expenses by a factor of Sizeblock. Larger block-level FTLs are similar to greater virtual memory page sizes, where you need less bits for the VPN and a larger offset in each virtual address.

Unfortunately, employing a block-based mapping inside a log-based FTL is slow. The main issue is when a “small write” occurs (i.e., one that is less than the size of a physical block). In this situation, the FTL must read and copy significant amounts of live data from the old block (along with the data from the small write). As a result, data copying reduces performance. Let’s look at an example to better understand. Take logical blocks 2000, 2001, 2002, and 2003 (with contents a,b,c,d) and place them in physical block 1 at physical pages 4, 5, 6, and 7. With per-page mappings, the translation table would have to record four mappings for these logical blocks: 2000→4, 2001→5, 2002→6, 2003→7.
Using block-level mapping reduces the number of address translations required by the FTL. However, the address mapping is slightly different. We think of the device’s logical address space as being divided into parts the size of flash blocks. It has two parts: a chunk number and an offset. Because each physical block contains four logical blocks, the offset portion of the logical addresses requires two bits; the remaining bits create the chunk number.

They all have the same chunk number (500) but different offsets (0, 1, 2, and 3, respectively). As indicated in the picture, chunk 500 maps to block 1 (beginning at physical page 4).

INSERT IMAGE

Reading is simple in FTL. First, the FTL gets the chunk number from the client’s logical block address by extracting the top bits. The FTL then looks up the chunk-to-page mapping in the table. Finally, the FTL computes the desired flash page address by adding the logical address offset to the block physical address.
This is done by looking up the translation in the mapping table (finding 4), and then adding an offset from the client’s logical address (2002, for example) to the translation (4). The data is located at the resulting physical-page address (6), which the FTL can read to acquire the needed data (c).

Suppose the client writes to logical block 2002 (contents ‘). In this example, the FTL must read in 2000, 2001, and 2003 before updating the mapping table. Block 1 can then be wiped and reused, as demonstrated above.

As this example shows, while block level mappings considerably reduce translation memory requirements, they cause significant performance issues when writes are smaller than the device’s physical block size. So we need a better solution. Can you tell that this is the chapter when we reveal the solution? Can you figure it out before reading on?

Hybrid Mapping

Many recent FTLs use hybrid mapping to allow flexible writing while reducing mapping costs. Using this method, the FTL retains a few blocks erased and directs all writes to them. Because the FTL wishes to avoid all the copying necessary by a pure block-based mapping, it saves per-page mappings for these log blocks.

For each page, the FTL keeps a small collection of per-page mappings in the log table, and a larger set of per-block mappings in the data table. If the FTL cannot identify a logical block in the log table, it consults the data table to discover it and then accesses the necessary data.

The number of log blocks is limited in hybrid mapping. To reduce the number of log blocks, the FTL must periodically inspect log blocks (one per page) and convert them into blocks with only one block pointer. There are three ways to achieve this switch depending on the contents of the block. So, logical pages 1000, 1001, 1002, and 1003 are written to physical block 2 (physical pages 8, 9, 10, 11), and their contents are a, b, c, and d.

So, let's say the client overwrites each of these blocks (in the exact same order) in one of the available log blocks, say physical block 0. (physical pages 0, 1, 2, and 3). In this situation, the FTL will be:

Because these blocks are identical, the FTL can conduct a switch merging. The former log block (2) is now erased and utilized as a log block. In this situation, a single block pointer replaces all per-page pointers.

This switch merge is a hybrid FTL best case. Sadly, the FTL is not always so lucky. For example, suppose we start with the same conditions (logical blocks 1000–1003 in physical block 2), but the client overwrites 1000 and 1001.
What do you think happens? Why is it harder to handle? (pause before viewing the following page's result)

The FTL does a partial merge to recombine the remaining pages of this physical block so that only one block pointer can refer to them. Read from physical block 2, logical blocks 1002 and 1003 are appended to the log. The SSD now has the same state as the switch merge, but the FTL has to do more I/O to meet its goals, raising write amplification.

A full merging is the FTL's last case and requires much more labor. In this scenario, the FTL must combine pages from many blocks to clean. Imagine writing logical blocks 0, 4, 8, and 12 to log block A. First, the FTL must build a data block containing logical blocks 0, 1, 2, and 3, then read 1, 2, and 3 from elsewhere and write them together. The merging must then discover

logical blocks 5, 6, and 7 and reassemble them into a single physical block. Then (eventually) log block A can be freed. Frequent full merges can substantially impair performance and should be avoided whenever possible.

Page Mapping with Caching

Others have suggested simpler solutions to lessen the memory consumption of page-mapped FTLs. The easiest is to store only the active bits of the FTL in memory, lowering memory requirements.

This strategy can work. For example, if a task only accesses a small number of pages, the in-memory FTL will provide good speed without consuming much memory. Of course, it can also fail. If memory doesn't have the appropriate translations, each access will require an extra flash read to bring in the missing mapping before accessing the data. Worse, if the old mapping is unclean (i.e., not yet written to the flash persistently), the FTL may have to evict it, causing an additional write. In many cases, the workload is local, therefore caching reduces memory overheads while maintaining performance.

Wear Leveling

Finally, as mentioned previously, modern FTLs must do wear leveling. Because many erase/program cycles wear out a flash block, the FTL should strive to distribute the work uniformly among all the blocks of the de- vice. Instead of a few “popular” blocks immediately becoming worthless, all blocks will wear out at around the same period.

The basic log-structuring strategy spreads out the write load initially, and trash collection also helps. However, if a block contains long-lived data that is never overwritten, garbage collection will never reclaim it, and it will not receive its fair share of write load.

So the FTL can write again, it must regularly read all the live data out of such blocks and rewrite it elsewhere. The extra I/O required to ensure that all blocks wear at roughly the same pace reduces performance.

SSD Performance

Before concluding, let's look at modern SSD performance and pricing to see how they might be used in persistent storage systems. In both cases, we'll compare to traditional hard drives (HDDs) and emphasize the key differences.

Flash-based SSDs, unlike hard disk drives, are "random access" devices that are similar to DRAM in many aspects. While a normal disk drive can only execute a few hundred random I/Os per second, SSDs can do far better. Here, we analyze data from contemporary SSDs to see how well FTLs disguise the performance concerns of the raw chips.

Table 44.4 presents performance data for three SSDs and one high-end hard disk from several internet sources. The first three rows show data for three different SSDs (from Samsung, Seagate, and Intel), and the last row displays performance for a hard disk drive (HDD), in this case a Seagate high-end drive.

The table reveals some fascinating information. The biggest difference is in random I/O speed between SSDs and a single hard drive. When compared to SSDs, this "high performance" hard drive has a peak of just a couple MB/s (we rounded up to 2MB/s). Second, while SSDs perform better, a hard drive is still an excellent choice if sequential performance is all you need. Third, SSD random read performance is inferior to SSD random write performance. The log-structured design of many SSDs converts random writes into sequential ones, improving speed. Since sequential and random I/Os perform differently on SSDs, many of the techniques we'll learn in later chapters about how to build file systems for hard drives still apply to SSDs. Although the performance difference between sequential and random I/Os is smaller, it's enough to consider how to design file systems that reduce random I/Os.

SSD Cost

As shown above, SSDs outperform current hard drives even while conducting sequential I/O. So why haven't SSDs completely replaced hard drives? This is simple: cost, or cost per unit of capability. Currently, a 250GB SSD costs around $150, or 60 cents per GB. A typical 1-TB hard disk costs around $50, or 5 cents per GB. The cost gap between these two storage medium is still over ten.

Large-scale storage systems are created based on performance and cost disparities. SSDs are a great choice for performance, especially for random reads. If you're building a big data center and need to store a lot of data, the pricing differential will push you towards hard drives. The usage of SSDs for popular "hot" data and high performance can be justified, while storing less used "colder" data on hard drives can save money. While the pricing difference continues, hard drives will remain.

Summary

Flash-based SSDs are becoming prevalent in laptops, desktops, and servers in the world's datacenters.

- A flash chip has multiple banks, each structured into erase blocks (sometimes just called blocks). Each block is broken into pages.
- Blocks are large (128KB–2MB) and contain several smaller pages (1KB–8KB).
- To read from flash, send a read command with an address and length.
- Flash is more difficult. Initially, the client must erase the entire block (which deletes all information within it). Once programmed, the client can complete the write.
- A new trim operation is important for informing the device that a certain block (or set of blocks) is no longer required.
- Reliability of flash is largely controlled by wear out; too much erasing and programming destroys flash.
- A flash-based SSD works like a regular block-based read/write disk, transforming client reads and writes into reads, erases, and programs to underlying flash chips.
- Most FTLs are log-structured, reducing writing costs by reducing erase/program cycles. An in-memory translation layer tracks physical writes.
- The cost of trash collection leads to write amplification in log-structured FTLs.
- Another issue is the mapping table's size, which can be rather extensive. Remedies include hybrid mapping or only caching hot FTL parts.
- Last but not least, wear leveling requires the FTL to migrate data from blocks that are mostly read to guarantee they get their fair portion of erase/program load.