**Chapter 44: Flash-based SSDs**

**Solid-state storage** does not have mechanical or moving parts like hard drives. It is simply built out of transistors, much like memory and processors. However, unlike typical random-access memory (e.g., DRAM), such a **solid-state storage device** (a.k.a., an **SSD**) retains information despite power loss, and thus is an ideal candidate for use in persistent storage of data.

The technology we’ll focus on is known as **flash**. This has some unique properties. For example, to write to a given chunk of it (i.e., a **flash page**), you first have to erase a bigger chunk (i.e., a **flash block**), which can be quite expensive. In addition, writing too often to a page will cause it to wear out.

**44.1 Storing a Single Bit**

Flash chips are designed to store one or more bits in a single transistor. The level of charge trapped within the transistor is mapped to a binary value. In a **single-level cell (SLC)** **flash**, only a single bit is stored within a transistor (i.e., 1 or 0); with a **multi-level cell (MLC) flash**, two bits are encoded into different levels of charge, e.g., 00, 01, 10, and 11 are represented by low, somewhat low, somewhat high, and high levels. There is even a **triple-level cell (TLC) flash** that encodes 3 bits per cell. Overall, SLC chips achieve higher performance and are more expensive.

**44.2 From Bits to Banks/Planes**

Flash chips are organized into **banks** or **planes** which consist of a large number of cells.

A Bank is accessed in two different sized unit: **blocks** (**erase** **blocks** that are typically 128KB or 256KB) and pages (4KB in size). Within each bank there are a large number of blocks and within each block, there are a large number of pages.

Chart, box and whisker chart

Description automatically generated

**44.3 Basic Flash Operations**

The **read** command is used to read a page from the flash; **erase** and **program** are used in tandem to write.

1. **Read (a page):** A client of the flash chip can read any page simply by specifying the read command and appropriate page number to the device. This operation is typically quite fast (10s of microseconds) regardless of the location of the device and location of previous request. Being able to access any location uniformly quickly means the device is a **random access** device.
2. **Erase (a block):** Before writing to a **page** within a flash, the nature of the device requires that you first erase the entire block the page lies within. Erase destroys the contents of the block (setting each bit to value 1). Thus, we must be sure that any data you care about in the block has been copied elsewhere (to memory, or perhaps to another flash block) before executing the erase. The erase command is quite expensive, taking a few milliseconds to complete. Once finished, the entire block is reset, and each page is ready to be programmed.
3. **Program (a page):** Once a block has been erased, the program command can be used to change some of the 1’s within a page to 0’s and write the desired contents of a page to the flash.

Each page has a state associate with it. Pages start in an INVALID state. By erasing the block that a page resides within, you set the state of the page (and all pages within that block) to ERASED which resets the content of each page in the block but also (importantly) makes them programmable. When you program a page, its state changes to VALID, meaning its contents have been set and can be read.

Reads do no affect these states. Once a page has been programmed, the only way to change it is to erase the entire block where the page resides.

Text

Description automatically generated

**A Detailed Example**

Imagine we have 4 8-byte pages, within a 4-page block. Each page is VLID as they has been programmed:

Table

Description automatically generated

If we want to write to page 0, filling it with new contents, we must first erase the entire block:

Table

Description automatically generated with medium confidence

We can then write the content we wished to write to page 0:

Table

Description automatically generated with medium confidence

However, the contents of other pages are gone. Thus, before overwriting any page within a block, we must save the data to another location.

In summary, reading a page is easy. However, for writing, we must first erase a block, and then the desired page programmed. Not only is this expensive, but frequent repetitions of this program/erase cycle can lead to the biggest reliability problem flash chips have: **wear out**.

**44.4 Flash Performance And Reliability**

Text, table

Description automatically generated

To get a good write performance, we will use the multiple flash chips in parallel.

Flash chips are pure silicon and in that sense have fewer reliability issues to worry about. The only concern is **wear out** as when a flash block is erased and programmed, it slowly accrues a little bit of extra charge. Over time, as that extra charge builds up, it becomes increasingly difficult to differentiate between a 0 and a 1. At the point where it becomes impossible, the block becomes unusable.

One other reliability problem is disturbance. When accessing a particular page within a flash, it is possible that some bits get flipped in neighboring pages; such bit flips are known as **read disturbs** or **program disturbs**, depending on whether the page is being read or programmed, respectively.

**44.5 From Raw Flash to Flash-Based SSDs**

Internally, an SSD consists of some number of flash chips, some amount of volatile memory for caching, buffering of data and for mapping tables and control logic to orchestrate device operation.

Diagram

Description automatically generated

The **flash translation layer**, or **FTL**, takes read and write requests on logical blocks and turns them into low-level read, erase, and program commands on the underlying physical blocks and physical pages that comprise the actual flash device.

Excellent performance is due to utilizing multiple flash chips in **parallel** and reducing **write amplification** (the total write traffic (in bytes) issued to the flash chips by the FTL divided by the total write traffic (in bytes) issued by the client to the SSD).

High reliability is also achieved by reducing wear out by spreading write across the blocks of the flash as evenly as possible, ensuring that all of the blocks of the device wear out at roughly the same time (called **wear leveling**). Another reason is that it minimizes program disturbance by programming pages within an erased block in order, from low page to high page.

**44.6 FTL Organization: A Bad Approach**

The simplest organization of an FTL is **direct mapped**. For this approach, a read to logical page N is mapped directly to a read of physical page N. A write to logical page N is more complicated as FTL first has to read in the entire block that page N is contained within. It then has to erase the block and programs the old pages as well as the new one.

The problem occurs in performance as on each write, the device has to read in the entire block, erase it and then program it. This results in severe write amplification. Regarding reliability, if file system metadata or user file data is repeatedly overwritten, the same block is erased and programmed, over and over, rapidly wearing it out and potentially losing data. This simply gives too much control over wear out to the client workload.

**44.7 A Log-Structured FTL**

Upon a write to logical block N, the device appends the write to the next free spot in the currently-being-written-to block. We call this style of writing **logging**. The device keeps a **mapping table**, that stores the physical address of each logical block in the system, to allow subsequent reads of block N.

Let’s assume that the client is reading or writing 4-KB sized chunks. Our SSD contains a large number of 16-KB sized blocks, each divided into 4 4-KB pages. The sequence of operation is described as follows:

• Write(100) with contents a1

• Write(101) with contents a2

• Write(2000) with contents b1

• Write(2001) with contents b2

Assume that all blocks of the SSD are currently not valid, and must be erased before any page can be programmed. The initial state is defined as follows:

Chart, box and whisker chart

Description automatically generated

When the first write is received by the SSD, the FTL decides to write it to physical block 0, which contains four physical pages: 0, 1, 2, and 3. The device must first erase block 0, leading to:

A screenshot of a computer

Description automatically generated with medium confidence

The SSD then directs the write of logical block 100 into physical page 0:

Chart, box and whisker chart

Description automatically generated

If the client wants to read logical block 100, the SSD must transform a read issued to logical block 100 into a read of physical page 0. To accommodate such functionality, when the FTL writes logical block 100 to physical page 0, it records this fact in an **in-memory mapping table**:

Chart

Description automatically generated

When the client want to write to the SSD, the SSD finds a location for the write, usually just picking the next free page; it then programs that page with the block’s contents, and records the logical-to-physical mapping in its mapping table. Subsequent reads simply use the table to translate the logical block address presented by the client into the physical page number required to read the data. For subsequence write 101, 2000 and 2001, the state of the device is.

Chart, timeline

Description automatically generated

The log-based approach improves performance by requiring erases once in a while and greatly enhances reliability by wear leveling.

However, there are some downsides to this. The first is that overwrites of logical blocks lead to something we call **garbage**. Thus, the device has to periodically perform **garbage collection (GC)** to find said blocks and free space for future writes. Excessive garbage collection drives up write amplification.

The second is high cost of in-memory mapping tables; the larger the device, the more memory such tables need.

**44.8 Garbage Collection**

Let’s use the same example where we write logical blocks 100, 101, 2000 and 2001 to the device. Assume the blocks 100 and 101 are written to again, with contents c1 and c2. Thus, the writes are written to the next free page (physical pages 4 and 5), and the mapping table is updated accordingly:

Table

Description automatically generated with medium confidence

The problem is now that physical pages 0 and 1 have **garbage** in them (older version of blocks 100 and 101) although they are VALID.

Thus, the **garbage collection** finds **garbage blocks** (**dead blocks**) and reclaim them in the future. The process is that it finds a block that contains one or more garbage pages, read in the live (non-garbage) pages from that block, write out those live pages to the log, and (finally) reclaim the entire block for use in writing. Consider the previous example, since block 0 has two dead blocks (00 and 01) and two live blocks (02 and 03), the device will read live data, write live data to end of the log and erase block 0 (freeing it for later usage).

To enable SSD to determine whether each page is live or dead, store information about which logical blocks are stored within each page within each block. The device can then use the mapping table to determine whether each page within the block holds live data or not.

When this garbage collection process is complete in our example, the state of the device isGraphical user interface, text, table

Description automatically generated

Garbage collection can be expensive, requiring reading and rewriting of live data. The ideal candidate for reclamation is a block that consists of only dead pages; in this case, the block can immediately be erased and used for new data, without expensive data migration.

To reduce GC costs, some SSDs overprovision the device by adding extra flash capacity, cleaning can be delayed and pushed to the background, perhaps done at a time when the device is less busy.

**44.9 Mapping Table Size**

**Block-Based Mapping**

One approach to reduce the costs of mapping is to only keep a pointer per block of the device, instead of per page, reducing the amount of mapping information by a factor of (size block)/(size page).

This **block-level** FTL is akin to having bigger page sizes in a virtual memory system. In that case, you use fewer bits for the VPN and have a larger offset in each virtual address.

The problem occurs when a small write occurs, FTL still reads a large amount of live data from the old block and copy it into a new one along with the data from the small write. This will increase write amplification.

To read in block-based FTL, the FTL first extracts the chunk number from the logical block address presented by the client, by taking the topmost bits out of the address. Then, the FTL looks up the chunk- number to physical-page mapping in the table. Finally, the FTL computes the address of the desired flash page by adding the offset from the logical address to the physical address of the block.

The write of FTL might cause some trouble, and in this case, we use:

**Hybrid Mapping**

With this approach, the FTL keeps a few blocks erased and directs all writes to them; these are called **log blocks**.

FTL has two types of mapping table in its memory: a small set of per-page mappings in what we’ll call the **log table**, and a larger set of per-block mappings in the **data table**.

When looking for a particular logical block, the FTL will first consult the log table; if the logical block’s location is not found there, the FTL will then consult the data table to find its location and then access the requested data.

The key to the hybrid mapping strategy is keeping the number of log blocks small. To keep the number of log blocks small, the FTL has to periodically examine log blocks (which have a pointer per page) and switch them into blocks that can be pointed to by only a single block pointer.

For example, FTL has previously written out logical pages 1000 to 1003 and placed them in physical block 2:

Chart

Description automatically generated

Now assume that the client overwrites each of these blocks (with data a’, b’, c’, and d’), in the exact same order, in one of the currently available log blocks, say physical block 0 (physical pages 0, 1, 2, and 3). In this case, the FTL will have the following state:

Chart, timeline

Description automatically generated

Because these blocks have been written exactly in the same manner as before, the FTL can perform what is known as a **switch merge**. In this case, the log block (0) now becomes the storage location for blocks 0, 1, 2, and 3, and is pointed to by a single block pointer; the old block (2) is now erased and used as a log block. In this best case, all the per-page pointers required replaced by a single block pointer:

Timeline

Description automatically generated

To reunite the other pages of this physical block, and thus be able to refer to them by only a single block pointer, the FTL performs what is called a **partial merge**. In this operation, logical blocks 1002 and 1003 are read from physical block 2, and then appended to the log. The resulting state of the SSD is the same as the switch merge above; however, in this case, the FTL had to perform extra I/O to achieve its goals, thus increasing write amplification:

Chart, timeline

Description automatically generated with medium confidence

The final case encountered by the FTL known as a **full merge** and requires even more work. In this case, the FTL must pull together pages from many other blocks to perform cleaning.

**Page Mapping Plus Caching**

We cache only the active parts of the FTL in memory to reduce the amount of memory needed.

**44.10 Wear Leveling**

The Idea is that because multiple erase/program cycles will wear out a flash block, the FTL should try its best to spread that work across all the blocks of the device evenly. Thus, all blocks will wear out at roughly the same time.

However, sometimes a block will be filled with long-lived data that does not get over-written; in this case, garbage collection will never reclaim the block, and thus it does not receive its fair share of the write load.

To deal with this problem, the FTL must periodically read all the live data out of such blocks and re-write it elsewhere, thus making the block available for writing again. This process of wear leveling increases the write amplification of the SSD, and thus decreases performance as extra I/O is required to ensure that all blocks wear at roughly the same rate.

**44.11 SSD Performance And Cost**

Performance: Flash-based SSD has no mechanical components, and it is **random access** device, so it performs well for random workload.

SSD’s random reads performance is not as good as random write performance because it transforms random writes into sequential ones.

Cost: SSD is ten time more expensive than HDD.