44

Flash-based SSDs

After decades of hard-disk drive dominance, a new form of persistent storage device has recently gained significance in the world. Generically referred to as **solid-state storage**, such devices have no mechanical or moving parts like hard drives; rather, they are 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** (more specifically, **NAND-based flash**), which was created by Fujio Masuoka in the 1980s [M+14]. Flash, as we'll see, 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**. These two properties make construction of a flash-based SSD an interesting challenge:

> CRUX: HOW TO BUILD A FLASH-BASED SSD
> How can we build a flash-based SSD? How can we handle the expensive nature of erasing? How can we build a device that lasts a long time, given that repeated overwrite will wear the device out? Will the march of progress in technology ever cease? Or cease to amaze?

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 **triple-level cell (TLC)** flash, which encodes 3 bits per cell. Overall, SLC chips achieve higher performance and are more expensive.

TIP: BE CAREFUL WITH TERMINOLOGY
You may have noticed that some terms we have used many times before
(blocks, pages) are being used within the context of a flash, but in slightly
different ways than before. New terms are not created to make your life
harder (although they may be doing just that), but arise because there is
no central authority where terminology decisions are made. What is a
block to you may be a page to someone else, and vice versa, depending
on the context. Your job is simple: to know the appropriate terms within
each domain, and use them such that people well-versed in the discipline
can understand what you are talking about. It's one of those times where
the only solution is simple but sometimes painful: use your memory.

Of course, there are many details as to exactly how such bit-level stor-
age operates, down at the level of device physics. While beyond the scope
of this book, you can read more about it on your own [J10].

44.2 From Bits to Banks/Planes

As they say in ancient Greece, storing a single bit (or a few) does not
a storage system make. Hence, flash chips are organized into **banks** or
planes which consist of a large number of cells.

A bank is accessed in two different sized units: **blocks** (sometimes
called **erase blocks**), which are typically of size 128 KB or 256 KB, and
pages, which are a few KB in size (e.g., 4KB). Within each bank there are
a large number of blocks; within each block, there are a large number of
pages. When thinking about flash, you must remember this new termi-
nology, which is different than the blocks we refer to in disks and RAIDs
and the pages we refer to in virtual memory.

Figure 44.1 shows an example of a flash plane with blocks and pages;
there are three blocks, each containing four pages, in this simple exam-
ple. We'll see below why we distinguish between blocks and pages; it
turns out this distinction is critical for flash operations such as reading
and writing, and even more so for the overall performance of the device.
The most important (and weird) thing you will learn is that to write to
a page within a block, you first have to erase the entire block; this tricky
detail makes building a flash-based SSD an interesting and worthwhile
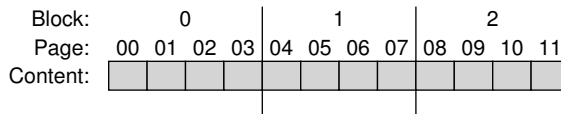challenge, and the subject of the second-half of the chapter.

Block:		0				1				2	
Page:	00	01	02	03	04	05	06	07	08	09	10	11
Content:												

Figure 44.1: **A Simple Flash Chip: Pages Within Blocks**

44.3 Basic Flash Operations

Given this flash organization, there are three low-level operations that a flash chip supports. The **read** command is used to read a page from the flash; **erase** and **program** are used in tandem to write. The details:

- **Read (a page)**: A client of the flash chip can read any page (e.g., 2KB or 4KB), simply by specifying the read command and appropriate page number to the device. This operation is typically quite fast, 10s of microseconds or so, regardless of location on the device, and (more or less) regardless of the location of the previous request (quite unlike a disk). Being able to access any location uniformly quickly means the device is a **random access** device.
- **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, importantly, destroys the contents of the block (by setting each bit to the value 1); therefore, you must be sure that any data you care about in the block has been copied elsewhere *WAF* (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.
- **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. Programming a page is less expensive than erasing a block, but more costly than reading a page, usually taking around 100s of microseconds on modern flash chips.

One way to think about flash chips is that each page has a state associated 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 not affect these states (although you should only read from pages that have been programmed). Once a page has been programmed, the only way to change its contents is to erase the entire block within which the page resides. Here is an example of states transition after various erase and program operations within a 4-page block:

	iiii	*Initial: pages in block are invalid* (i)
Erase()	→	EEEE	*State of pages in block set to erased* (E)
Program(0)	→	VEEE	*Program page 0; state set to valid* (V)
Program(0)	→	**error**	*Cannot re-program page after programming*
Program(1)	→	VVEE	*Program page 1*
Erase()	→	EEEE	*Contents erased; all pages programmable*

A Detailed Example

Because the process of writing (i.e., erasing and programming) is so unusual, let's go through a detailed example to make sure it makes sense. In this example, imagine we have the following four 8-bit pages, within a 4-page block (both unrealistically small sizes, but useful within this example); each page is VALID as each has been previously programmed.

Page 0	Page 1	Page 2	Page 3
00011000	11001110	00000001	00111111
VALID	VALID	VALID	VALID

Now say we wish to write to page 0, filling it with new contents. To write any page, we must first erase the entire block. Let's assume we do so, thus leaving the block in this state:

Page 0	Page 1	Page 2	Page 3
11111111	11111111	11111111	11111111
ERASED	ERASED	ERASED	ERASED

Good news! We could now go ahead and program page 0, for example with the contents 00000011, overwriting the old page 0 (contents 00011000) as desired. After doing so, our block looks like this:

Page 0	Page 1	Page 2	Page 3
00000011	11111111	11111111	11111111
VALID	ERASED	ERASED	ERASED

And now the bad news: the previous contents of pages 1, 2, and 3 are all gone! Thus, before overwriting any page *within* a block, we must first move any data we care about to another location (e.g., memory, or elsewhere on the flash). The nature of erase will have a strong impact on how we design flash-based SSDs, as we'll soon learn about.

Summary

To summarize, reading a page is easy: just read the page. Flash chips do this quite well, and quickly; in terms of performance, they offer the potential to greatly exceed the random read performance of modern disk drives, which are slow due to mechanical seek and rotation costs.

Writing a page is trickier; the entire block must first be erased (taking care to first move any data we care about to another location), 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**. When designing a storage system with flash, the performance and reliability of writing is a central focus. We'll soon learn more about how modern SSDs attack these issues, delivering excellent performance and reliability despite these limitations.

Device	Read (μs)	Program (μs)	Erase (μs)
SLC	25	200-300	1500-2000
MLC	50	600-900	~3000
TLC	~75	~900-1350	~4500

Figure 44.2: **Raw Flash Performance Characteristics**

44.4 Flash Performance And Reliability

Because we're interested in building a storage device out of raw flash chips, it is worthwhile to understand their basic performance characteristics. Figure 44.2 presents a rough summary of some numbers found in the popular press [V12]. Therein, the author presents the basic operation latency of reads, programs, and erases across SLC, MLC, and TLC flash, which store 1, 2, and 3 bits of information per cell, respectively.

As we can see from the table, read latencies are quite good, taking just 10s of microseconds to complete. Program latency is higher and more variable, as low as 200 microseconds for SLC, but higher as you pack more bits into each cell; to get good write performance, you will have to make use of multiple flash chips in parallel. Finally, erases are quite expensive, taking a few milliseconds typically. Dealing with this cost is central to modern flash storage design.

Let's now consider reliability of flash chips. Unlike mechanical disks, which can fail for a wide variety of reasons (including the gruesome and quite physical **head crash**, where the drive head actually makes contact with the recording surface), flash chips are pure silicon and in that sense have fewer reliability issues to worry about. The primary concern is **wear out**; 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.

The typical lifetime of a block is currently not well known. Manufacturers rate MLC-based blocks as having a 10,000 P/E (Program/Erase) cycle lifetime; that is, each block can be erased and programmed 10,000 times before failing. SLC-based chips, because they store only a single bit per transistor, are rated with a longer lifetime, usually 100,000 P/E cycles. However, recent research has shown that lifetimes are much longer than expected [BD10].

One other reliability problem within flash chips is known as **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.

TIP: THE IMPORTANCE OF BACKWARDS COMPATIBILITY
Backwards compatibility is always a concern in layered systems. By
defining a stable interface between two systems, one enables innovation
on each side of the interface while ensuring continued interoperability.
Such an approach has been quite successful in many domains: operating
systems have relatively stable APIs for applications, disks provide the
same block-based interface to file systems, and each layer in the IP net-
working stack provides a fixed unchanging interface to the layer above.

Not surprisingly, there can be a downside to such rigidity, as interfaces
defined in one generation may not be appropriate in the next. In some
cases, it may be useful to think about redesigning the entire system en-
tirely. An excellent example is found in the Sun ZFS file system [B07];
by reconsidering the interaction of file systems and RAID, the creators of
ZFS envisioned (and then realized) a more effective integrated whole.

44.5 From Raw Flash to Flash-Based SSDs

Given our basic understanding of flash chips, we now face our next
task: how to turn a basic set of flash chips into something that looks like
a typical storage device. The standard storage interface is a simple block-
based one, where blocks (sectors) of size 512 bytes (or larger) can be read
or written, given a block address. The task of the flash-based SSD is to
provide that standard block interface atop the raw flash chips inside it.

Internally, an SSD consists of some number of flash chips (for persis-
tent storage). An SSD also contains some amount of volatile (i.e., non-
persistent) memory (e.g., SRAM); such memory is useful for caching and
buffering of data as well as for mapping tables, which we'll learn about
below. Finally, an SSD contains control logic to orchestrate device opera-
tion. See Agrawal et. al for details [A+08]; a simplified block diagram is
seen in Figure 44.3 (page 7).

One of the essential functions of this control logic is to satisfy client
reads and writes, turning them into internal flash operations as need be.
The **flash translation layer**, or **FTL**, provides exactly this functionality.
The FTL takes read and write requests on *logical blocks* (that comprise the
device interface) and turns them into low-level read, erase, and program
commands on the underlying *physical blocks* and *physical pages* (that com-
prise the actual flash device). The FTL should accomplish this task with
the goal of delivering excellent performance and high reliability.

Excellent performance, as we'll see, can be realized through a com-
bination of techniques. One key will be to utilize multiple flash chips
in **parallel**; although we won't discuss this technique much further, suf-
fice it to say that all modern SSDs use multiple chips internally to obtain
higher performance. Another performance goal will be to reduce **write
amplification**, which is defined as the total write traffic (in bytes) issued
to the flash chips by the FTL divided by the total write traffic (in bytes) is-
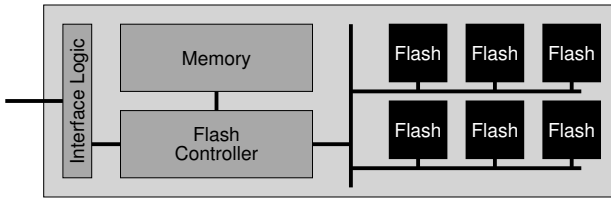
Figure 44.3: **A Flash-based SSD: Logical Diagram**

sued by the client to the SSD. As we'll see below, naive approaches to FTL construction will lead to high write amplification and low performance.

High reliability will be achieved through the combination of a few different approaches. One main concern, as discussed above, is **wear out**. If a single block is erased and programmed too often, it will become unusable; as a result, the FTL should try to spread writes 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; doing so is called **wear leveling** and is an essential part of any modern FTL.

Another reliability concern is program disturbance. To minimize such disturbance, FTLs will commonly program pages within an erased block *in order*, from low page to high page. This sequential-programming approach minimizes disturbance and is widely utilized.

44.6 FTL Organization: A Bad Approach

The simplest organization of an FTL would be something we call **direct mapped**. In 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; the FTL first has to read in the entire block that page N is contained within; it then has to erase the block; finally, the FTL programs the old pages as well as the new one.

As you can probably guess, the direct-mapped FTL has many problems, both in terms of performance as well as reliability. The performance problems come on each write: the device has to read in the entire block (costly), erase it (quite costly), and then program it (costly). The end result is severe write amplification (proportional to the number of pages in a block) and as a result, terrible write performance, even slower than typical hard drives with their mechanical seeks and rotational delays.

Even worse is the reliability of this approach. 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. The direct mapped approach simply gives too much control over wear out to the client workload; if the workload does not spread write load evenly across its logical blocks, the underlying physical blocks containing popular data will quickly wear out. For both reliability and performance reasons, a direct-mapped FTL is a bad idea.

44.7 A Log-Structured FTL

For these reasons, most FTLs today are **log structured**, an idea useful in both storage devices (as we'll see now) and file systems above them (as we'll see in the chapter on **log-structured file systems**). 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**. To allow for subsequent reads of block N, the device keeps a **mapping table** (in its memory, and persistent, in some form, on the device); this table stores the physical address of each logical block in the system.

pohent-free - page.

Let's go through an example to make sure we understand how the basic log-based approach works. To the client, the device looks like a typical disk, in which it can read and write 512-byte sectors (or groups of sectors). For simplicity, assume that the client is reading or writing 4-KB sized chunks. Let us further assume that the SSD contains some large number of 16-KB sized blocks, each divided into four 4-KB pages; these parameters are unrealistic (flash blocks usually consist of more pages) but will serve our didactic purposes quite well.

Assume the client issues the following sequence of operations:

- Write(100) with contents a1
- Write(101) with contents a2
- Write(2000) with contents b1
- Write(2001) with contents b2

These **logical block addresses** (e.g., 100) are used by the client of the SSD (e.g., a file system) to remember where information is located.

Internally, the device must transform these block writes into the erase and program operations supported by the raw hardware, and somehow record, for each logical block address, which **physical page** of the SSD stores its data. Assume that all blocks of the SSD are currently not valid, and must be erased before any page can be programmed. Here we show the initial state of our SSD, with all pages marked INVALID (i):

Block:	0				1				2			
Page:	00	01	02	03	04	05	06	07	08	09	10	11
Content:												
State:	i	i	i	i	i	i	i	i	i	i	i	i

When the first write is received by the SSD (to logical block 100), the FTL decides to write it to physical block 0, which contains four physical pages: 0, 1, 2, and 3. Because the block is not erased, we cannot write to it yet; the device must first issue an erase command to block 0. Doing so leads to the following state:

Block:	0				1				2			
Page:	00	01	02	03	04	05	06	07	08	09	10	11
Content:												
State:	E	E	E	E	i	i	i	i	i	i	i	i

Block 0 is now ready to be programmed. Most SSDs will write pages in order (i.e., low to high), reducing reliability problems related to **program disturbance**. The SSD then directs the write of logical block 100 into physical page 0:

Block:	0				1				2			
Page:	00	01	02	03	04	05	06	07	08	09	10	11
Content:	a1											
State:	V	E	E	E	i	i	i	i	i	i	i	i

But what if the client wants to *read* logical block 100? How can it find where it is? 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**. We will track the state of this mapping table in the diagrams as well:

Table:	100 ➔ 0											Memory

| Block:	0				1				2				
Page:	00	01	02	03	04	05	06	07	08	09	10	11	Flash
Content:	a1												Chip
State:	V	E	E	E	i	i	i	i	i	i	i	i	

Now you can see what happens when the client writes 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.

Let's now examine the rest of the writes in our example write stream: 101, 2000, and 2001. After writing these blocks, the state of the device is:

Table:	100 ➔ 0	101 ➔ 1	2000 ➔ 2	2001 ➔ 3								Memory

Block:	0				1				2				
Page:	00	01	02	03	04	05	06	07	08	09	10	11	Flash
Content:	a1	a2	b1	b2									Chip
State:	V	V	V	V	i	i	i	i	i	i	i	i	

The log-based approach by its nature improves performance (erases only being required once in a while, and the costly read-modify-write of the direct-mapped approach avoided altogether), and greatly enhances reliability. The FTL can now spread writes across all pages, performing what is called **wear leveling** and increasing the lifetime of the device; we'll discuss wear leveling further below.
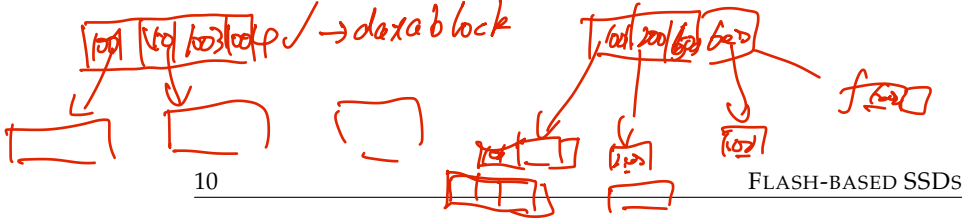
> ASIDE: FTL MAPPING INFORMATION PERSISTENCE
> You might be wondering: what happens if the device loses power? Does
> the in-memory mapping table disappear? Clearly, such information can-
> not truly be lost, because otherwise the device would not function as a
> persistent storage device. An SSD must have some means of recovering
> mapping information.
>
> The simplest thing to do is to record some mapping information with
> each page, in what is called an **out-of-band** (**OOB**) area. When the device
> loses power and is restarted, it must reconstruct its mapping table by
> scanning the OOB areas and reconstructing the mapping table in mem-
> ory. This basic approach has its problems; scanning a large SSD to find
> all necessary mapping information is slow. To overcome this limitation,
> some higher-end devices use more complex **logging** and **checkpointing**
> techniques to speed up recovery; learn more about logging by reading
> chapters on crash consistency and log-structured file systems [AD14].

Unfortunately, this basic approach to log structuring has some down-
sides. The first is that overwrites of logical blocks lead to something we
call **garbage**, i.e., old versions of data around the drive and taking up
space. The device has to periodically perform **garbage collection** (**GC**) to
find said blocks and free space for future writes; excessive garbage collec-
tion drives up write amplification and lowers performance. The second
is high cost of in-memory mapping tables; the larger the device, the more
memory such tables need. We now discuss each in turn.

44.8 Garbage Collection

The first cost of any log-structured approach such as this one is that
garbage is created, and therefore **garbage collection** (i.e., dead-block recla-
mation) must be performed. Let's use our continued example to make
sense of this. Recall that logical blocks 100, 101, 2000, and 2001 have been
written to the device.

Now, let's assume that blocks 100 and 101 are written to again, with
contents c1 and c2. The writes are written to the next free pages (in this
case, physical pages 4 and 5), and the mapping table is updated accord-
ingly. Note that the device must have first erased block 1 to make such
programming possible:

Table:	100 → 4	101 → 5	2000 → 2	2001 → 3	Memory

	Block:		0				1				2			
Page:	00	01	02	03	04	05	06	07	08	09	10	11	Flash
Content:	a1	a2	b1	b2	c1	c2							Chip
State:	V	V	V	V	V	V	E	E	i	i	i	i	

The problem we have now should be obvious: physical pages 0 and 1, although marked VALID, have **garbage** in them, i.e., the old versions of blocks 100 and 101. Because of the log-structured nature of the device, overwrites create garbage blocks, which the device must reclaim to provide free space for new writes to take place.

The process of finding garbage blocks (also called **dead blocks**) and reclaiming them for future use is called **garbage collection**, and it is an important component of any modern SSD. The basic process is simple: find 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.

Let's now illustrate with an example. The device decides it wants to reclaim any dead pages within block 0 above. Block 0 has two dead blocks (pages 0 and 1) and two live blocks (pages 2 and 3, which contain blocks 2000 and 2001, respectively). To do so, the device 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 function, there must be enough information within each block to enable the SSD to determine whether each page is live or dead. One natural way to achieve this end is to store, at some location within each block, information about which logical blocks are stored within each page. The device can then use the mapping table to determine whether each page within the block holds live data or not.

From our example above (before the garbage collection has taken place), block 0 held logical blocks 100, 101, 2000, 2001. By checking the mapping table (which, before garbage collection, contained 100->4, 101->5, 2000->2, 2001->3), the device can readily determine whether each of the pages within the SSD block holds live information. For example, 2000 and 2001 clearly are still pointed to by the map; 100 and 101 are not and therefore are candidates for garbage collection.

When this garbage collection process is complete in our example, the state of the device is:

Table:	100 ➜ 4	101 ➜ 5	2000 ➜ 6	2001 ➜ 7	Memory

Block:		0				1				2			
Page:	00	01	02	03	04	05	06	07	08	09	10	11	Flash
Content:					c1	c2	b1	b2					Chip
State:	E	E	E	E	V	V	V	V	i	i	i	i	

As you can see, 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.

ASIDE: A NEW STORAGE API KNOWN AS TRIM
When we think of hard drives, we usually just think of the most basic interface to read and write them: read and write (there is also usually some kind of **cache flush** command, ensuring that writes have actually been persisted, but sometimes we omit that for simplicity). With log-structured SSDs, and indeed, any device that keeps a flexible and changing mapping of logical-to-physical blocks, a new interface is useful, known as the **trim** operation.
The trim operation takes an address (and possibly a length) and simply informs the device that the block(s) specified by the address (and length) have been deleted; the device thus no longer has to track any information about the given address range. For a standard hard drive, trim isn't particularly useful, because the drive has a static mapping of block addresses to specific platter, track, and sector(s). For a log-structured SSD, however, it is highly useful to know that a block is no longer needed, as the SSD can then remove this information from the FTL and later reclaim the physical space during garbage collection.
Although we sometimes think of interface and implementation as separate entities, in this case, we see that the implementation shapes the interface. With complex mappings, knowledge of which blocks are no longer needed makes for a more effective implementation.

To reduce GC costs, some SSDs **overprovision** the device [A+08]; 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. Adding more capacity also increases internal bandwidth, which can be used for cleaning and thus not harm perceived bandwidth to the client. Many modern drives overprovision in this manner, one key to achieving excellent overall performance.

44.9 Mapping Table Size

The second cost of log-structuring is the potential for extremely large mapping tables, with one entry for each 4-KB page of the device. With a large 1-TB SSD, for example, a single 4-byte entry per 4-KB page results in 1 GB of memory needed by the device, just for these mappings! Thus, this **page-level** FTL scheme is impractical.

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 $\frac{Size_{block}}{Size_{page}}$. 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.

Unfortunately, using a block-based mapping inside a log-based FTL does not work very well for performance reasons. The biggest problem arises when a "small write" occurs (i.e., one that is less than the size of a physical block). In this case, the FTL must read 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 data copying increases write amplification greatly and thus decreases performance.

To make this issue more clear, let's look at an example. Assume the client previously wrote out logical blocks 2000, 2001, 2002, and 2003 (with contents, a, b, c, d), and that they are located within 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.

If, instead, we use block-level mapping, the FTL only needs to record a single address translation for all of this data. The address mapping, however, is slightly different than our previous examples. Specifically, we think of the logical address space of the device as being chopped into chunks that are the size of the physical blocks within the flash. Thus, the logical block address consists of two portions: a chunk number and an offset. Because we are assuming four logical blocks fit within each physical block, the offset portion of the logical addresses requires 2 bits; the remaining (most significant) bits form the chunk number.

Logical blocks 2000, 2001, 2002, and 2003 all have the same chunk number (500), and have different offsets (0, 1, 2, and 3, respectively). Thus, with a block-level mapping, the FTL records that chunk 500 maps to block 1 (starting at physical page 4), as shown in this diagram:

Table:	500 → 4											Memory

Block:		0				1				2		
Page:	00	01	02	03	04	05	06	07	08	09	10	11	Flash
Content:					a	b	c	d					Chip
State:	i	i	i	i	V	V	V	V	i	i	i	i	

In a block-based FTL, reading is easy. First, the FTL 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.

For example, if the client issues a read to logical address 2002, the device extracts the logical chunk number (500), looks up the translation in the mapping table (finding 4), and adds the offset from the logical address (2) to the translation (4). The resulting physical-page address (6) is

where the data is located; the FTL can then issue the read to that physical address and obtain the desired data (c).

But what if the client writes to logical block 2002 (with contents c')? In this case, the FTL must read in 2000, 2001, and 2003, and then write out all four logical blocks in a new location, updating the mapping table accordingly. Block 1 (where the data used to reside) can then be erased and reused, as shown here.

正是因为这种
block-level mapping
存在问题，导致修改"动"
要很高 block.

然后就使用 page-level
mapping，则直接把整页
page 写入新页面.

Table:	500 ➝ 8											Memory
Block:	0				1				2				
Page:	00	01	02	03	04	05	06	07	08	09	10	11	Flash
Content:									a	b	c'	d	Chip
State:	i	i	i	i	E	E	E	E	V	V	V	V	

As you can see from this example, while block level mappings greatly reduce the amount of memory needed for translations, they cause significant performance problems when writes are smaller than the physical block size of the device; as real physical blocks can be 256KB or larger, such writes are likely to happen quite often. Thus, a better solution is needed. Can you sense that this is the part of the chapter where we tell you what that solution is? Better yet, can you figure it out yourself, before reading on?

Hybrid Mapping

To enable flexible writing but also reduce mapping costs, many modern FTLs employ a **hybrid mapping** technique. With this approach, the FTL keeps a few blocks erased and directs all writes to them; these are called **log blocks**. Because the FTL wants to be able to write any page to any location within the log block without all the copying required by a pure block-based mapping, it keeps *per-page* mappings for these log blocks.

page-map. ←

The FTL thus logically 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. This switch is accomplished by one of three main techniques, based on the contents of the block [KK+02].

For example, let's say the FTL had previously written out logical pages 1000, 1001, 1002, and 1003, and placed them in physical block 2 (physical

pages 8, 9, 10, 11); assume the contents of the writes to 1000, 1001, 1002, and 1003 are `a`, `b`, `c`, and `d`, respectively.

Log Table:
Data Table: 250 → 8   Memory

Block:	0				1				2				
Page:	00	01	02	03	04	05	06	07	08	09	10	11	Flash
Content:									a	b	c	d	Chip
State:	i	i	i	i	i	i	i	i	V	V	V	V	

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:

Log Table: 1000→0   1001→1   1002→2   1003→3
Data Table: 250 → 8   Memory

Block:	0				1				2				
Page:	00	01	02	03	04	05	06	07	08	09	10	11	Flash
Content:	a'	b'	c'	d'					a	b	c	d	Chip
State:	V	V	V	V	i	i	i	i	V	V	V	V	

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.

Log Table:
Data Table: 250 → 0   Memory

Block:	0				1				2				
Page:	00	01	02	03	04	05	06	07	08	09	10	11	Flash
Content:	a'	b'	c'	d'									Chip
State:	V	V	V	V	i	i	i	i	i	i	i	i	

This switch merge is the best case for a hybrid FTL. Unfortunately, sometimes the FTL is not so lucky. Imagine the case where we have the same initial conditions (logical blocks 1000 ... 1003 stored in physical block 2) but then the client overwrites logical blocks 1000 and 1001.

What do you think happens in this case? Why is it more challenging to handle? *(think before looking at the result on the next page)*

| Log Table: | 1000 → 0 | 1001 → 1 | | | | | | | | | |
| Data Table: | 250 → 8 | | | | | | | | | | Memory |

Block:		0				1				2		
Page:	00	01	02	03	04	05	06	07	08	09	10	11	Flash
Content:	a'	b'							a	b	c	d	Chip
State:	V	V	i	i	i	i	i	i	V	V	V	V	

To reunite the other pages of this physical block, and thus be able to re-
fer 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.   *read*.

The final case encountered by the FTL known as a **full merge**, and re-
quires even more work. In this case, the FTL must pull together pages
from many other blocks to perform cleaning. For example, imagine that
logical blocks 0, 4, 8, and 12 are written to log block A. To switch this log
block into a block-mapped page, the FTL must first create a data block
containing logical blocks 0, 1, 2, and 3, and thus the FTL must read 1, 2,
and 3 from elsewhere and then write out 0, 1, 2, and 3 together. Next, the
merge must do the same for logical block 4, finding 5, 6, and 7 and recon-
ciling them into a single physical block. The same must be done for logi-
cal blocks 8 and 12, and then (finally), the log block A can be freed. Fre-
quent full merges, as is not surprising, can seriously harm performance
and thus should be avoided when at all possible [GY+09].

Page Mapping Plus Caching

Given the complexity of the hybrid approach above, others have sug-
gested simpler ways to reduce the memory load of page-mapped FTLs.
Probably the simplest is just to cache only the active parts of the FTL in
memory, thus reducing the amount of memory needed [GY+09].

This approach can work well. For example, if a given workload only
accesses a small set of pages, the translations of those pages will be stored
in the in-memory FTL, and performance will be excellent without high
memory cost. Of course, the approach can also perform poorly. If mem-
ory cannot contain the **working set** of necessary translations, each access
will minimally require an extra flash read to first bring in the missing
mapping before being able to access the data itself. Even worse, to make
room for the new mapping, the FTL might have to **evict** an old map-
ping, and if that mapping is **dirty** (i.e., not yet written to the flash per-
sistently), an extra write will also be incurred. However, in many cases,
the workload will display locality, and this caching approach will both
reduce memory overheads and keep performance high.

44.10 Wear Leveling

Finally, a related background activity that modern FTLs must implement is wear leveling, as introduced above. The basic idea is simple: 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. In this manner, all blocks will wear out at roughly the same time, instead of a few "popular" blocks quickly becoming unusable.

The basic log-structuring approach does a good initial job of spreading out write load, and garbage collection helps as well. 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 remedy 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. Many different algorithms exist in the literature [A+08, M+14]; read more if you are interested.

44.11 SSD Performance And Cost

Before closing, let's examine the performance and cost of modern SSDs, to better understand how they will likely be used in persistent storage systems. In both cases, we'll compare to classic hard-disk drives (HDDs), and highlight the biggest differences between the two.

Performance

Unlike hard disk drives, flash-based SSDs have no mechanical components, and in fact are in many ways more similar to DRAM, in that they are "random access" devices. The biggest difference in performance, as compared to disk drives, is realized when performing random reads and writes; while a typical disk drive can only perform a few hundred random I/Os per second, SSDs can do much better. Here, we use some data from modern SSDs to see just how much better SSDs perform; we're particularly interested in how well the FTLs hide the performance issues of the raw chips.

Table 44.4 shows some performance data for three different SSDs and one top-of-the-line hard drive; the data was taken from a few different online sources [S13, T15]. The left two columns show random I/O performance, and the right two columns sequential; the first three rows show data for three different SSDs (from Samsung, Seagate, and Intel), and the last row shows performance for a **hard disk drive** (or **HDD**), in this case a Seagate high-end drive.

We can learn a few interesting facts from the table. First, and most dramatic, is the difference in random I/O performance between the SSDs

	Random		Sequential	
Device	Reads (MB/s)	Writes (MB/s)	Reads (MB/s)	Writes (MB/s)
Samsung 840 Pro SSD	103	287	421	384
Seagate 600 SSD	84	252	424	374
Intel SSD 335 SSD	39	222	344	354
Seagate Savvio 15K.3 HDD	2	2	223	223

Figure 44.4: **SSDs And Hard Drives: Performance Comparison**

and the lone hard drive. While the SSDs obtain tens or even hundreds of MB/s in random I/Os, this "high performance" hard drive has a peak of just a couple MB/s (in fact, we rounded up to get to 2 MB/s). Second, you can see that in terms of sequential performance, there is much less of a difference; while the SSDs perform better, a hard drive is still a good choice if sequential performance is all you need. Third, you can see that SSD random read performance is not as good as SSD random write performance. The reason for such unexpectedly good random-write performance is due to the log-structured design of many SSDs, which transforms random writes into sequential ones and improves performance. Finally, because SSDs exhibit some performance difference between sequential and random I/Os, many of the techniques we will learn in subsequent chapters about how to build file systems for hard drives are still applicable to SSDs; although the magnitude of difference between sequential and random I/Os is smaller, there is enough of a gap to carefully consider how to design file systems to reduce random I/Os.

Cost

As we saw above, the performance of SSDs greatly outstrips modern hard drives, even when performing sequential I/O. So why haven't SSDs completely replaced hard drives as the storage medium of choice? The answer is simple: cost, or more specifically, cost per unit of capacity. Currently [A15], an SSD costs something like $150 for a 250-GB drive; such an SSD costs 60 cents per GB. A typical hard drive costs roughly $50 for 1-TB of storage, which means it costs 5 cents per GB. There is still more than a $10\times$ difference in cost between these two storage media.

These performance and cost differences dictate how large-scale storage systems are built. If performance is the main concern, SSDs are a terrific choice, particularly if random read performance is important. If, on the other hand, you are assembling a large data center and wish to store massive amounts of information, the large cost difference will drive you towards hard drives. Of course, a hybrid approach can make sense – some storage systems are being assembled with both SSDs and hard drives, using a smaller number of SSDs for more popular "hot" data and delivering high performance, while storing the rest of the "colder" (less used) data on hard drives to save on cost. As long as the price gap exists, hard drives are here to stay.

44.12 Summary

Flash-based SSDs are becoming a common presence in laptops, desktops, and servers inside the datacenters that power the world's economy. Thus, you should probably know something about them, right?

Here's the bad news: this chapter (like many in this book) is just the first step in understanding the state of the art. Some places to get some more information about the raw technology include research on actual device performance (such as that by Chen et al. [CK+09] and Grupp et al. [GC+09]), issues in FTL design (including works by Agrawal et al. [A+08], Gupta et al. [GY+09], Huang et al. [H+14], Kim et al. [KK+02], Lee et al. [L+07], and Zhang et al. [Z+12]), and even distributed systems comprised of flash (including Gordon [CG+09] and CORFU [B+12]). And, if we may say so, a really good overview of all the things you need to do to extract high performance from an SSD can be found in a paper on the "unwritten contract" [HK+17].

Don't just read academic papers; also read about recent advances in the popular press (e.g., [V12]). Therein you'll learn more practical (but still useful) information, such as Samsung's use of both TLC and SLC cells within the same SSD to maximize performance (SLC can buffer writes quickly) as well as capacity (TLC can store more bits per cell). And this is, as they say, just the tip of the iceberg. Dive in and learn more about this "iceberg" of research on your own, perhaps starting with Ma et al.'s excellent (and recent) survey [M+14]. Be careful though; icebergs can sink even the mightiest of ships [W15].

ASIDE: KEY SSD TERMS

- A **flash chip** consists of many banks, each of which is organized into **erase blocks** (sometimes just called **blocks**). Each block is further subdivided into some number of **pages**.

- Blocks are large (128KB–2MB) and contain many pages, which are relatively small (1KB–8KB).

- To read from flash, issue a read command with an address and length; this allows a client to read one or more pages.

- Writing flash is more complex. First, the client must **erase** the entire block (which deletes all information within the block). Then, the client can **program** each page exactly once, thus completing the write.

- A new **trim** operation is useful to tell the device when a particular block (or range of blocks) is no longer needed.

- Flash reliability is mostly determined by **wear out**; if a block is erased and programmed too often, it will become unusable.

- A flash-based **solid-state storage device** (**SSD**) behaves as if it were a normal block-based read/write disk; by using a **flash translation layer** (**FTL**), it transforms reads and writes from a client into reads, erases, and programs to underlying flash chips.

- Most FTLs are **log-structured**, which reduces the cost of writing by minimizing erase/program cycles. An in-memory translation layer tracks where logical writes were located within the physical medium.

- One key problem with log-structured FTLs is the cost of **garbage collection**, which leads to **write amplification**.

- Another problem is the size of the mapping table, which can become quite large. Using a **hybrid mapping** or just **caching** hot pieces of the FTL are possible remedies.

- One last problem is **wear leveling**; the FTL must occasionally migrate data from blocks that are mostly read in order to ensure said blocks also receive their share of the erase/program load.

References

[A+08] "Design Tradeoffs for SSD Performance" by N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, R. Panigrahy. USENIX '08, San Diego California, June 2008. *An excellent overview of what goes into SSD design.*

[AD14] "Operating Systems: Three Easy Pieces" by *Chapters: Crash Consistency: FSCK and Journaling and Log-Structured File Systems*. Remzi Arpaci-Dusseau and Andrea Arpaci-Dusseau. *A lot more detail here about how logging can be used in file systems; some of the same ideas can be applied inside devices too as need be.*

[A15] "Amazon Pricing Study" by Remzi Arpaci-Dusseau. February, 2015. *This is not an actual paper, but rather one of the authors going to Amazon and looking at current prices of hard drives and SSDs. You too can repeat this study, and see what the costs are today. Do it!*

[B+12] "CORFU: A Shared Log Design for Flash Clusters" by M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobber, M. Wei, J. D. Davis. NSDI '12, San Jose, California, April 2012. *A new way to think about designing a high-performance replicated log for clusters using Flash.*

[BD10] "Write Endurance in Flash Drives: Measurements and Analysis" by Simona Boboila, Peter Desnoyers. FAST '10, San Jose, California, February 2010. *A cool paper that reverse engineers flash-device lifetimes. Endurance sometimes far exceeds manufacturer predictions, by up to $100\times$.*

[B07] "ZFS: The Last Word in File Systems" by Jeff Bonwick and Bill Moore. Available here: `http://www.ostep.org/Citations/zfs_last.pdf`. *Was this the last word in file systems? No, but maybe it's close.*

[CG+09] "Gordon: Using Flash Memory to Build Fast, Power-efficient Clusters for Data-intensive Applications" by Adrian M. Caulfield, Laura M. Grupp, Steven Swanson. ASPLOS '09, Washington, D.C., March 2009. *Early research on assembling flash into larger-scale clusters; definitely worth a read.*

[CK+09] "Understanding Intrinsic Characteristics and System Implications of Flash Memory based Solid State Drives" by Feng Chen, David A. Koufaty, and Xiaodong Zhang. SIGMETRICS/Performance '09, Seattle, Washington, June 2009. *An excellent overview of SSD performance problems circa 2009 (though now a little dated).*

[G14] "The SSD Endurance Experiment" by Geoff Gasior. The Tech Report, September 19, 2014. Available: `http://techreport.com/review/27062`. *A nice set of simple experiments measuring performance of SSDs over time. There are many other similar studies; use google to find more.*

[GC+09] "Characterizing Flash Memory: Anomalies, Observations, and Applications" by L. M. Grupp, A. M. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P. H. Siegel, J. K. Wolf. IEEE MICRO '09, New York, New York, December 2009. *Another excellent characterization of flash performance.*

[GY+09] "DFTL: a Flash Translation Layer Employing Demand-Based Selective Caching of Page-Level Address Mappings" by Aayush Gupta, Youngjae Kim, Bhuvan Urgaonkar. ASPLOS '09, Washington, D.C., March 2009. *This paper gives an excellent overview of different strategies for cleaning within hybrid SSDs as well as a new scheme which saves mapping table space and improves performance under many workloads.*

[HK+17] "The Unwritten Contract of Solid State Drives" by Jun He, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. EuroSys '17, Belgrade, Serbia, April 2017. *Our own paper which lays out five rules clients should follow in order to get the best performance out of modern SSDs. The rules are request scale, locality, aligned sequentiality, grouping by death time, and uniform lifetime. Read the paper for details!*

[H+14] "An Aggressive Worn-out Flash Block Management Scheme To Alleviate SSD Performance Degradation" by Ping Huang, Guanying Wu, Xubin He, Weijun Xiao. EuroSys '14, 2014. *Recent work showing how to really get the most out of worn-out flash blocks; neat!*

THREE
EASY
PIECES

[J10] "Failure Mechanisms and Models for Semiconductor Devices" by Unknown author. Report JEP122F, November 2010. Available on the internet at this exciting so-called web site: `http://www.jedec.org/sites/default/files/docs/JEP122F.pdf`. *A highly detailed discussion of what is going on at the device level and how such devices fail. Only for those not faint of heart. Or physicists. Or both.*

[KK+02] "A Space-Efficient Flash Translation Layer For Compact Flash Systems" by Jesung Kim, Jong Min Kim, Sam H. Noh, Sang Lyul Min, Yookun Cho. IEEE Transactions on Consumer Electronics, Volume 48, Number 2, May 2002. *One of the earliest proposals to suggest hybrid mappings.*

[L+07] "A Log Buffer-Based Flash Translation Layer by Using Fully-Associative Sector Translation. " Sang-won Lee, Tae-Sun Chung, Dong-Ho Lee, Sangwon Park, Ha-Joo Song. ACM Transactions on Embedded Computing Systems, Volume 6, Number 3, July 2007 *A terrific paper about how to build hybrid log/block mappings.*

[M+14] "A Survey of Address Translation Technologies for Flash Memories" by Dongzhe Ma, Jianhua Feng, Guoliang Li. ACM Computing Surveys, Volume 46, Number 3, January 2014. *Probably the best recent survey of flash and related technologies.*

[S13] "The Seagate 600 and 600 Pro SSD Review" by Anand Lal Shimpi. AnandTech, May 7, 2013. Available: `http://www.anandtech.com/show/6935/seagate-600-ssd-review`. *One of many SSD performance measurements available on the internet. Haven't heard of the internet? No problem. Just go to your web browser and type "internet" into the search tool. You'll be amazed at what you can learn.*

[T15] "Performance Charts Hard Drives" by Tom's Hardware. January 2015. Available here: `http://www.tomshardware.com/charts/enterprise-hdd-charts`. *Yet another site with performance data, this time focusing on hard drives.*

[V12] "Understanding TLC Flash" by Kristian Vatto. AnandTech, September, 2012. Available: `http://www.anandtech.com/show/5067/understanding-tlc-nand`. *A short description about TLC flash and its characteristics.*

[W15] "List of Ships Sunk by Icebergs" by Many authors. Available at this location on the "web": `http://en.wikipedia.org/wiki/List_of_ships_sunk_by_icebergs`. *Yes, there is a wikipedia page about ships sunk by icebergs. It is a really boring page and basically everyone knows the only ship the iceberg-sinking-mafia cares about is the Titanic.*

[Z+12] "De-indirection for Flash-based SSDs with Nameless Writes" by Yiying Zhang, Leo Prasath Arulraj, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. FAST '13, San Jose, California, February 2013. *Our research on a new idea to reduce mapping table space; the key is to re-use the pointers in the file system above to store locations of blocks, instead of adding another level of indirection.*

Homework (Simulation)

This section introduces `ssd.py`, a simple SSD simulator you can use to understand better how SSDs work. Read the README for details on how to run the simulator. It is a long README, so boil a cup of tea (caffeinated likely necessary), put on your reading glasses, let the cat curl up on your lap[1], and get to work.

Questions

1. The homework will mostly focus on the log-structured SSD, which is simulated with the "-T log" flag. We'll use the other types of SSDs for comparison. First, run with flags `-T log -s 1 -n 10 -q`. Can you figure out which operations took place? Use `-c` to check your answers (or just use `-C` instead of `-q -c`). Use different values of `-s` to generate different random workloads.

2. Now just show the commands and see if you can figure out the intermediate states of the Flash. Run with flags `-T log -s 2 -n 10 -C` to show each command. Now, determine the state of the Flash between each command; use `-F` to show the states and see if you were right. Use different random seeds to test your burgeoning expertise.

3. Let's make this problem ever so slightly more interesting by adding the `-r 20` flag. What differences does this cause in the commands? Use `-c` again to check your answers.

4. Performance is determined by the number of erases, programs, and reads (we assume here that trims are free). Run same workload again as above, but without showing any intermediate states (e.g., `-T log -s 1 -n 10`). Can you estimate how long this workload will take to complete? (default erase time is 1000 microseconds, program time is 40, and read time is 10) Use the `-S` flag to check your answer. You can also change the erase, program, and read times with the `-E, -W, -R` flags.

5. Now, compare performance of the log-structured approach and the (very bad) direct approach (`-T direct` instead of `-T log`). First, estimate how you think the direct approach will perform, then check your answer with the `-S` flag. In general, how much better will the log-structured approach perform than the direct one?

6. Let us next explore the behavior of the garbage collector. To do so, we have to set the high (`-G`) and low (`-g`) watermarks appropriately. First, let's observe what happens when you run a larger workload to the log-structured SSD but without any garbage collection. To do this, run with flags `-T log -n 1000` (the high wa-

---

[1]Now you might complain, "But I'm a dog person!" To this, we say, too bad! Get a cat, put it on your lap, and do the homework! How else will you learn, if you can't even follow the most basic of instructions?

termark default is 10, so the GC won't run in this configuration).
What do you think will happen? Use `-C` and perhaps `-F` to see.

7. To turn on the garbage collector, use lower values. The high water-
mark (`-G N`) tells the system to start collecting once `N` blocks have
been used; the low watermark (`-G M`) tells the system to stop col-
lecting once there are only `M` blocks in use. What watermark values
do you think will make for a working system? Use `-C` and `-F` to
show the commands and intermediate device states and see.

8. One other useful flag is `-J`, which shows what the collector is doing
when it runs. Run with flags `-T log -n 1000 -C -J` to see both
the commands and the GC behavior. What do you notice about the
GC? The final effect of GC, of course, is performance. Use `-S` to
look at final statistics; how many extra reads and writes occur due
to garbage collection? Compare this to the ideal SSD (`-T ideal`);
how much extra reading, writing, and erasing is there due to the
nature of Flash? Compare it also to the `direct` approach; in what
way (erases, reads, programs) is the log-structured approach supe-
rior?

9. One last aspect to explore is **workload skew**. Adding skew to the
workload changes writes such that more writes occur to some smaller
fraction of the logical block space. For example, running with `-K
80/20` makes 80% of the writes go to 20% of the blocks. Pick some
different skews and perform many randomly-chosen operations (e.g.,
`-n 1000`), using first `-T direct` to understand the skew, and then
`-T log` to see the impact on a log-structured device. What do you
expect will happen? One other small skew control to explore is `-k
100`; by adding this flag to a skewed workload, the first 100 writes
are not skewed. The idea is to first create a lot of data, but then only
update some of it. What impact might that have upon a garbage
collector?