File storage on NAND-Flash storage devices (aka SSDs)

Vinod Ganapathy

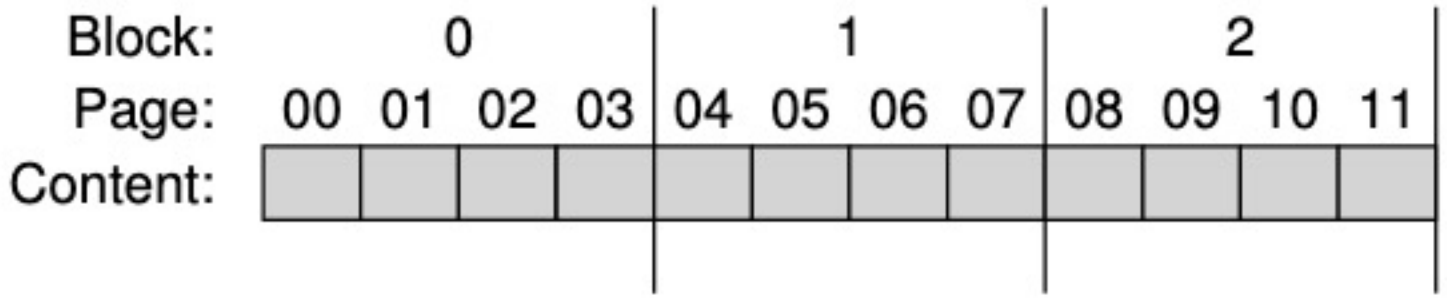
Material drawn from the OSTEP book

What are flash chips, aka SSDs?

- Flash chips are designed to store one or more bits in a single transistor
- single-level cell (SLC) flash, only a single bit is stored within a transistor (i.e., 1 or 0)
- 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.
- no mechanical or moving parts like hard drives; rather, they are simply built out of transistors, much like memory
- retains information despite power loss, and thus is an ideal candidate for use in persistent storage of data

Some terminology

- 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
    - Within each bank there are a large number of blocks
  - **Pages**, which are a few KB in size (e.g., 4KB)
    - Within each block, there are a large number of pages



The API to a Flash chip: Three operations

- **Read (a page)**:
  - 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
  - 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).
  - able to access any location uniformly quickly means the device is a **random access device**

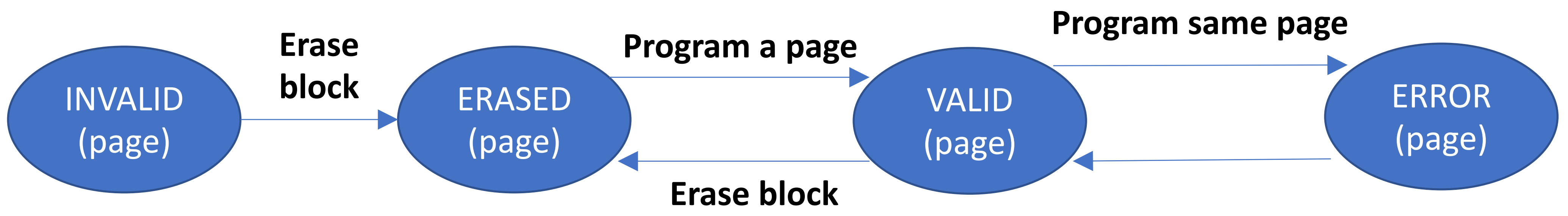
The API to a Flash chip: Three operations

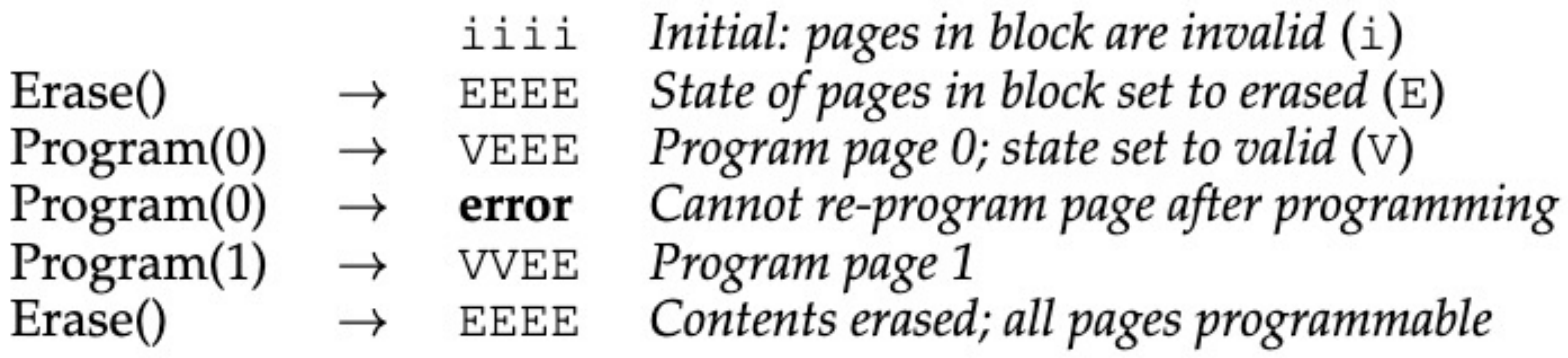
- **Erase (a block):**
  - Before writing to a page within a flash, the nature of the device requires that you first erase the entire block the page lies within.
  - Erase, importantly, destroys the contents of the block (by setting each bit to the value 1);
  - therefore, you must be sure that any data you care about in the block has been copied elsewhere (to memory, or perhaps to another flash block) before executing the erase
  - **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.

The API to a Flash chip: Three operations

- **Program (a page)**:
  - Once a block has been erased, the program com-mand 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.
  - **Writing too often to a page will cause it to wear out! SSD degrades over time. Can’t keep writing to the same page over and over again.**

Visualizing as a state machine

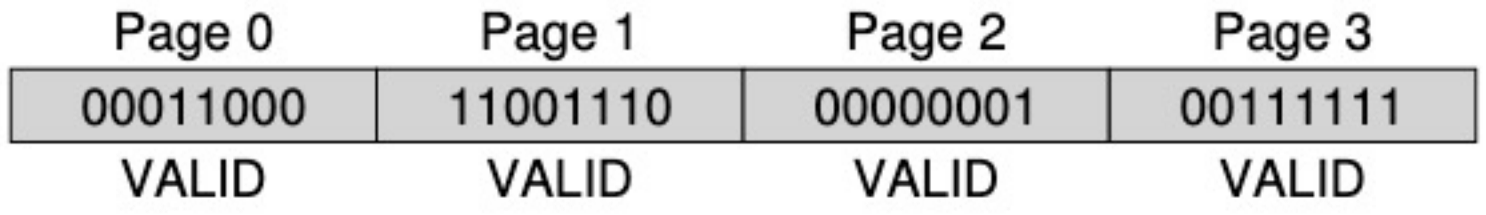




		iiii	*Initial: pages in block are invalid* (i)
Erase()	→	EEEE	*State of pages in block set to erased* (E)
Program(0)	→	VEEE	*Program page 0; state set to valid* (V)
Program(0)	→	**error**	*Cannot re-program page after programming*
Program(1)	→	VVEE	*Program page 1*
Erase()	→	EEEE	*Contents erased; all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

An example

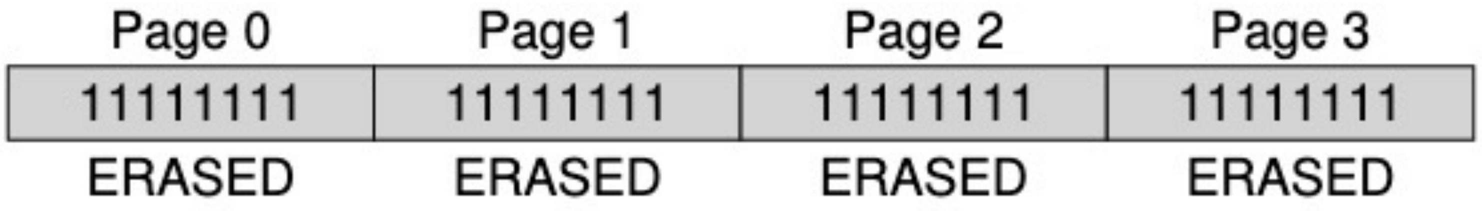
- imagine we have the four 8-bit pages, within a 4-page block (both unrealistically small sizes, but useful within this example);
- each page is VALID as each has been previously programmed



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

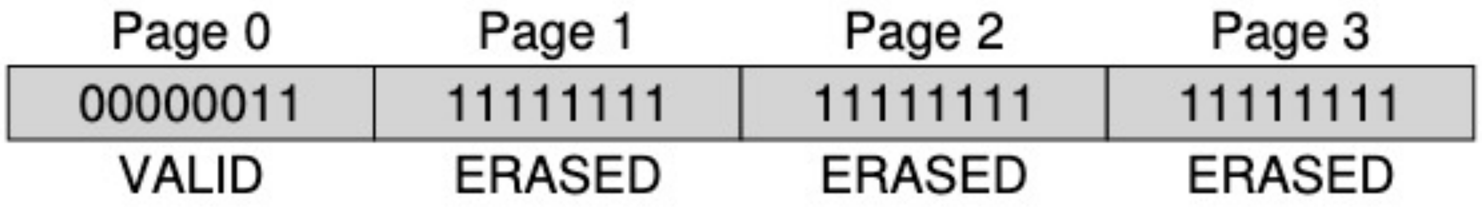
An example

- say we wish to write to page 0, filling it with new contents.
- To write any page, we must first erase the entire block



An example

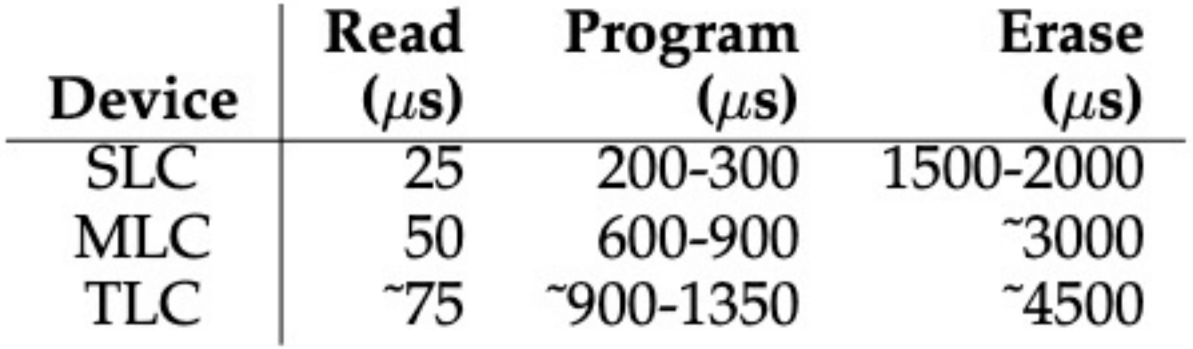
- Now program page 0 with required content, e.g., 00000011, overwriting the old page 0 (old contents 00011000) as desired



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

- Bad news: the previous contents of pages 1, 2, and 3 are all gone!
  - 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)
  - nature of erase will have a strong impact on how we design flash-based SSDs

Summary of operations



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

- Reading a page is easy:
  - just read the page;
  - very fast; thus, random read performance is just great!
- Writing a page is trickier:
  - Entire block must be erased;
  - Any data we care about must be copied to another location;
  - Expensive operation to Erase, not to mention costs of copying data;
  - Wear-out of locations that are written-to (i.e., programmed) many times;

Reliability of flash chips

- No need to worry about head crashes, like traditional hard disks.
- Primary concern: **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.
  - MLC-based chips: 10,000 P/E (Program/Erase) cycles per block
  - SLC-based chips: 100,000 P/E (Program/Erase) cycles per block

From flash chips to a storage device

- Internally, an SSD consists of some number of flash chips (for persistent storage).
- An SSD also contains some amount of volatile (i.e., nonpersistent) 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 in this lecture.
- Finally, an SSD contains control logic to orchestrate device operation.
- Control logic satisfies client reads and writes, turning them into internal flash operations as need be.
- Called the **flash translation layer, or FTL**

Flash translation layer (FTL)

- FTL's main job:
  - Take read and write requests on logical blocks (that comprise the device interface)
  - And turn them into low-level read, erase, and program commands on the underlying physical blocks and physical pages (that comprise the actual flash device)
- The device interface is the level at which the OS generates read/writes, at the granularity of blocks.
- The FTL converts them into physical device operations on the SSD

FTL: Performance considerations

- utilize multiple flash chips in parallel;
- reduce **write amplification**,
  - defined as 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.
  - Lowest possible value for write amplification is 1. Lower is better.
- FTL should try to spread writes across the blocks 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 design

Strawman design for FTL: Direct mapping

- Read to logical page N is mapped directly to a read of physical page N
- Write to logical page N:
  - FTL first reads in the entire block that page N is contained within; I
  - t then erases the block;
  - finally, the FTL programs the old pages as well as the new one.
- Why not an ideal solution?
  - **Performance problems**: HUGE write amplification. The device has to read in the entire block (costly), erase it (quite costly), and then program it (costly).
  - **Reliability problems**: 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.
- Application workload determines how SSD wears!

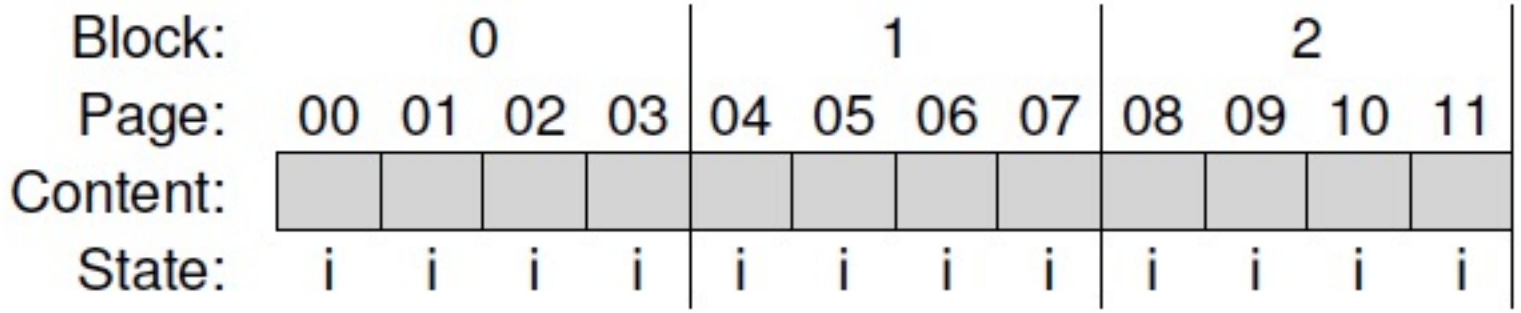
Log-structured FTL

- **Last class**: Log-structured FILE SYSTEM.
- **This class**: FTL logic is LOG-STRUCTURED. The file system above it can be based on any design (it can be log-structured too, but need not be)
- Basic idea:
  - Upon a write to logical block N, the device appends the write to the next free spot in the currently-being-written-to block on the SSD;
  - 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);
- **Mapping table** stores the physical address of each logical block in the system.

An example

- Assume that the client is reading or writing 4-KB sized chunks;
- Assume that the SSD contains some large number of 16-KB sized blocks, each divided into four 4-KB pages;
- Workload (shows logical addresses)
  - Write(100) with contents a1
  - Write(101) with contents a2
  - Write(2000) with contents b1
  - Write(2001) with contents b2
- FTL 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.

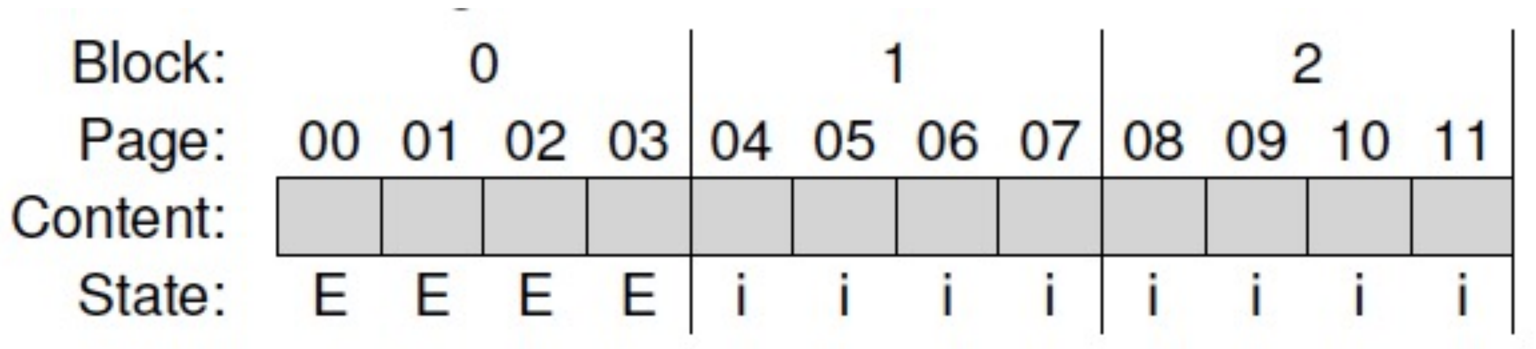
Example: initial state



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

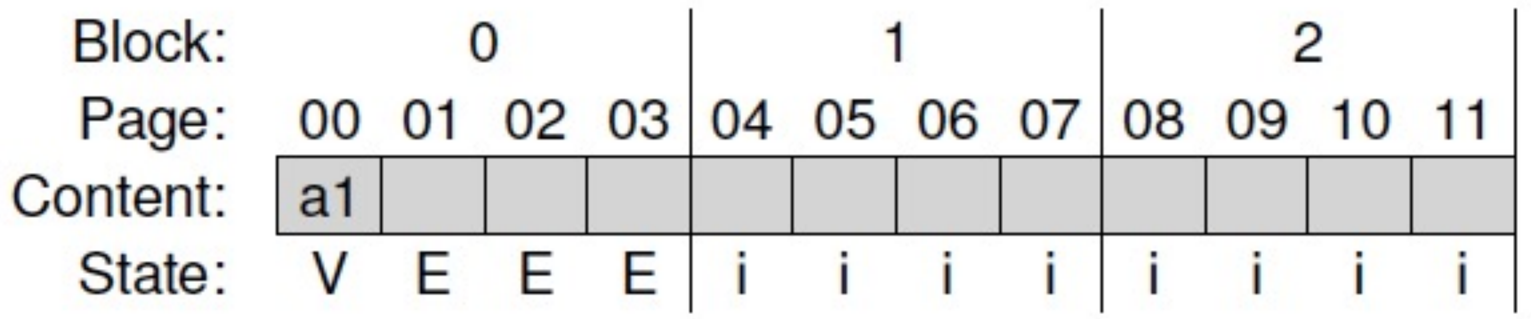
Write(100) with contents a1

- 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.



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

Write(100) with contents a1



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 how to read what we just wrote? How to find where it is? Put the mapping in an **in-memory mapping table**

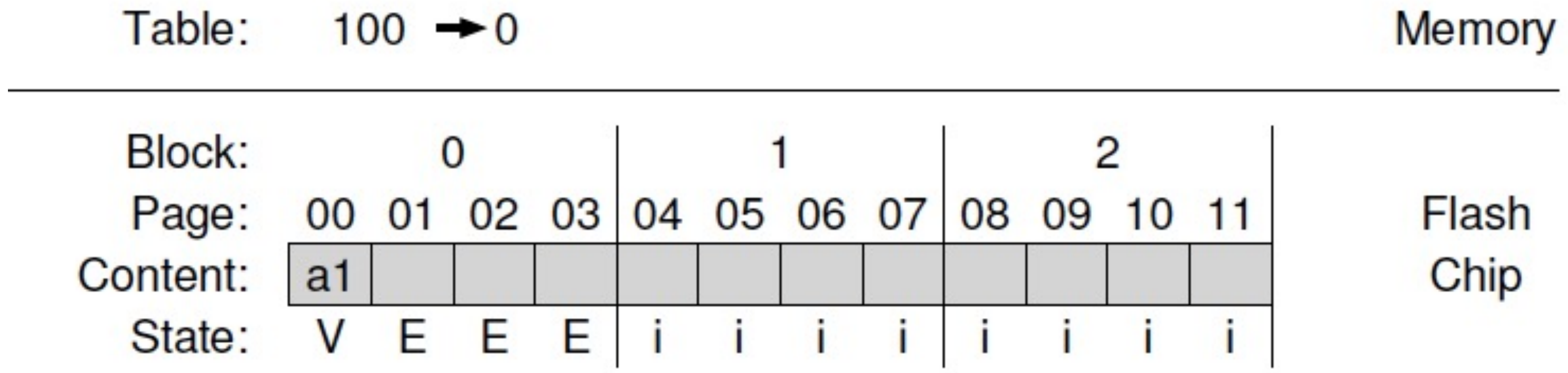


Table: 100 → 0
Memory
Block: 0 1 2
Page: 00 01 02 03 04 05 06 07 08 09 10 11
Content: a1
State: V E E E i i i i i i i i
Flash Chip

After all four writes

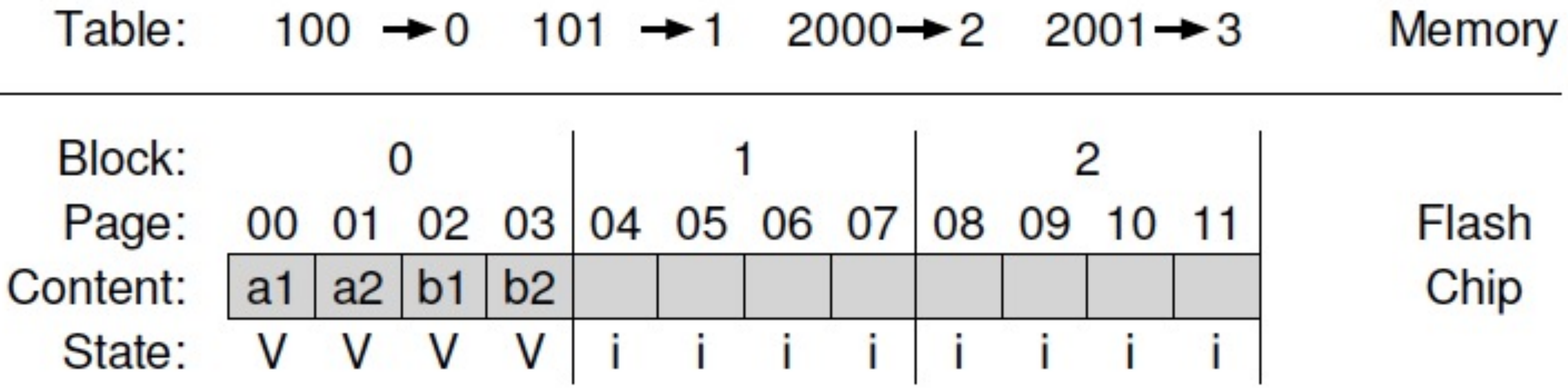


Table: 100 → 0 101 → 1 2000 → 2 2001 → 3
Memory
Block: 0 1 2
Page: 00 01 02 03 04 05 06 07 08 09 10 11
Content: a1 a2 b1 b2
State: V V V V i i i i i i i i
Flash Chip

- 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)
- 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

Two major problems

- **First,** overwrites of logical blocks lead to garbage, i.e., old versions of data around the drive and taking up space.
  - 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 and lowers performance
- **Second**, high cost of in-memory mapping tables; the larger the device, the more memory such tables need

- Next: FTL solutions to the above two problems.

Garbage collection (via example)

- Assume that blocks 100 and 101 are written to again, with contents c1 and c2.
- Writes continue into physical block 1: SSD has to first erase that block and make it ready for programming

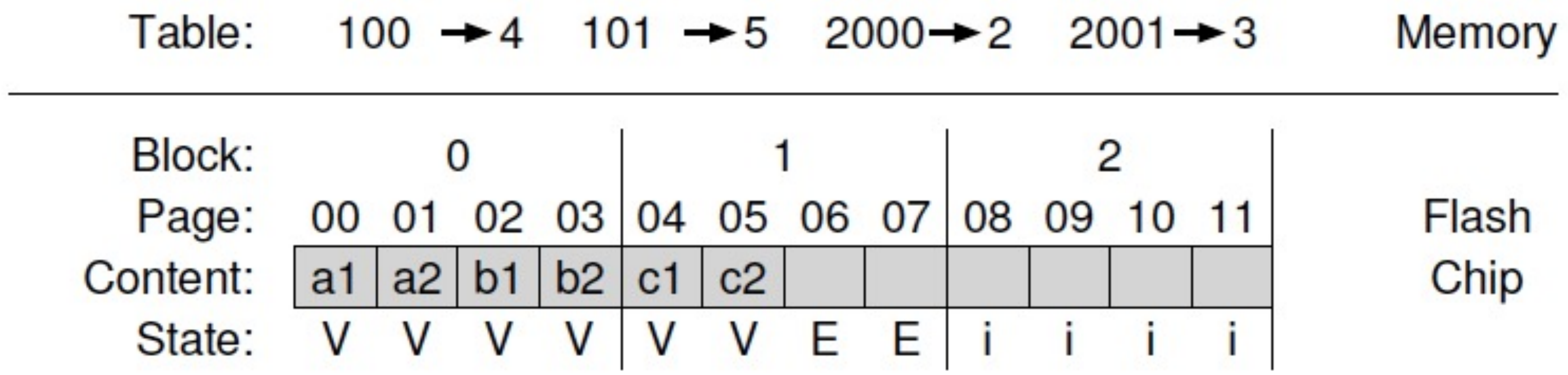


Table: 100 → 4 101 → 5 2000 → 2 2001 → 3 Memory

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	

- We have accumulated two pages of garbage.

Garbage collection (from time to time)

- 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 garbage collect, 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)
- How does garbage collector know which pages are **live** within each block?
  - Check the mapping table!

After garbage collection

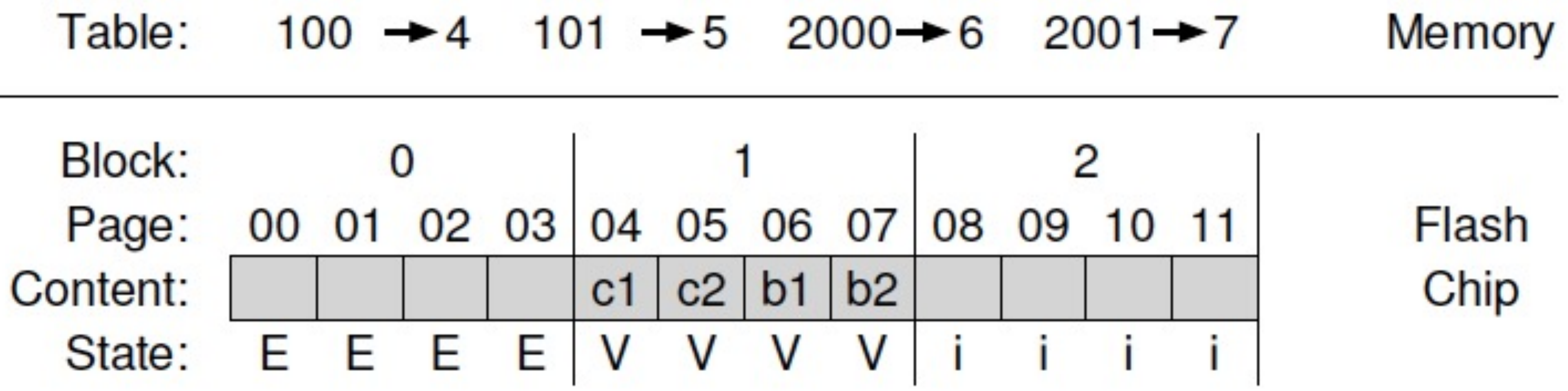


Table: 100 → 4 101 → 5 2000 → 6 2001 → 7
Memory
Block: 0 1 2
Page: 00 01 02 03 04 05 06 07 08 09 10 11
Content: c1 c2 b1 b2
State: E E E E V V V V i i i i
Flash Chip

- Rather an expensive operation, since blocks need to be erased and writes need to happen

Mapping table structure

- Suppose one entry for each 4-KB page of the device
- With 1TB SSD disk, with 1 4-byte entry per page in the mapping table, **1 GB of memory needed by the device, just for these mappings.**
- Let’s look at different methods of storing the mapping table:
  - Block-based mapping
  - Hybrid mapping

Block-based mapping table structure

- One pointer **per block** instead of one pointer **per page**
- Similar to large pages/huge pages idea you saw in memory management!
- **Problem?** Doesn't work very well for small writes.
  - 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.

Example

- Assume 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.

Example

- 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).

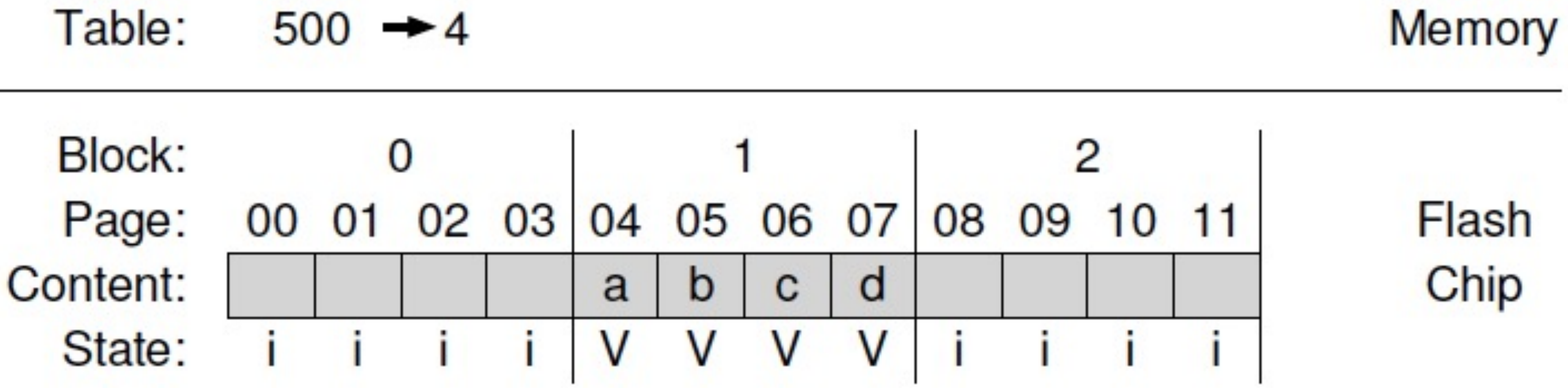


Table: 500 → 4
Memory
Block: 0 1 2
Page: 00 01 02 03 04 05 06 07 08 09 10 11
Content: a b c d
State: i i i i V V V V i i i i
Flash Chip

Reading in a block-based FTL

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

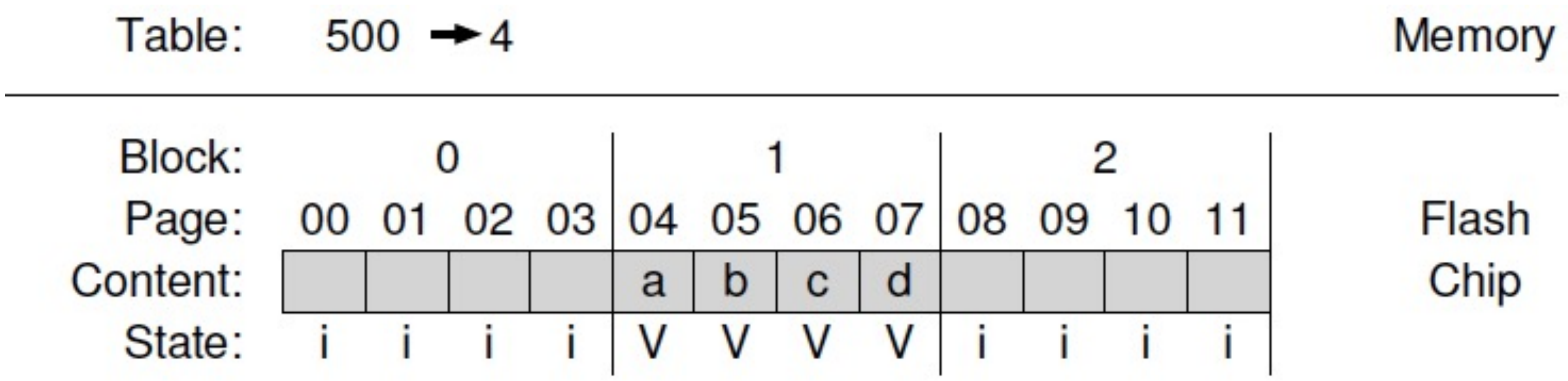


Table: 500 → 4
Memory
Block: 0 1 2
Page: 00 01 02 03 04 05 06 07 08 09 10 11
Content: a b c d
State: i i i i V V V V i i i i
Flash Chip

Writing in a block-based FTL

- Say client writes to logical block 2002 (with contents c')?
- 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.

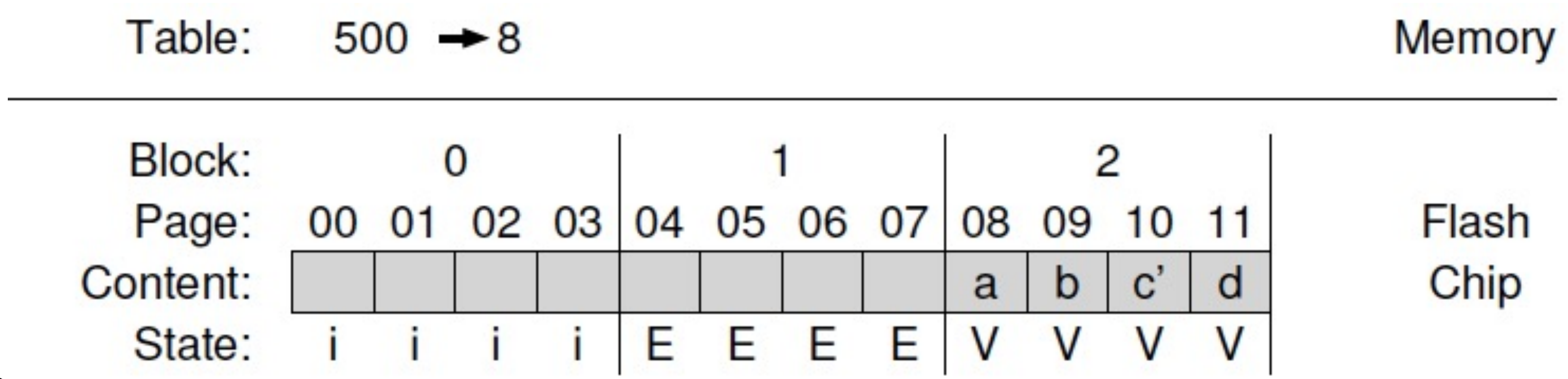


Table: 500 → 8
Memory
Block: 0 1 2
Page: 00 01 02 03 04 05 06 07 08 09 10 11
Content: a b c' d
State: i i i i E E E E V V V V
Flash Chip

- 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.

Hybrid mapping

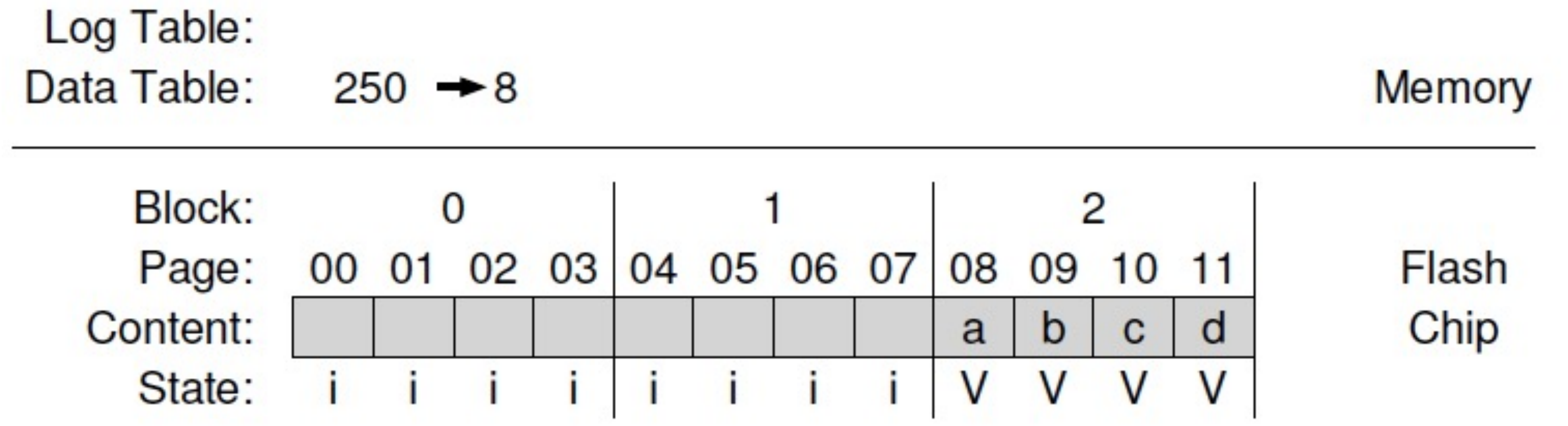
- FTL keeps a few blocks erased and directs all writes to them; these are called **log blocks**.
- FTL wants to be able to write any page to any location within the log block without all the copying required by a pure block-based mapping, it **keeps per-page mappings for these log blocks.**
- FTL thus logically has two types of mapping tables in its memory:
    - small set of per-page mappings --- the log table;
    - Larger set of per-block mappings in the data table.
- **Lookup:** 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

Hybrid mapping

- 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:
  - Switch merge
  - Partial merge
  - Full merge

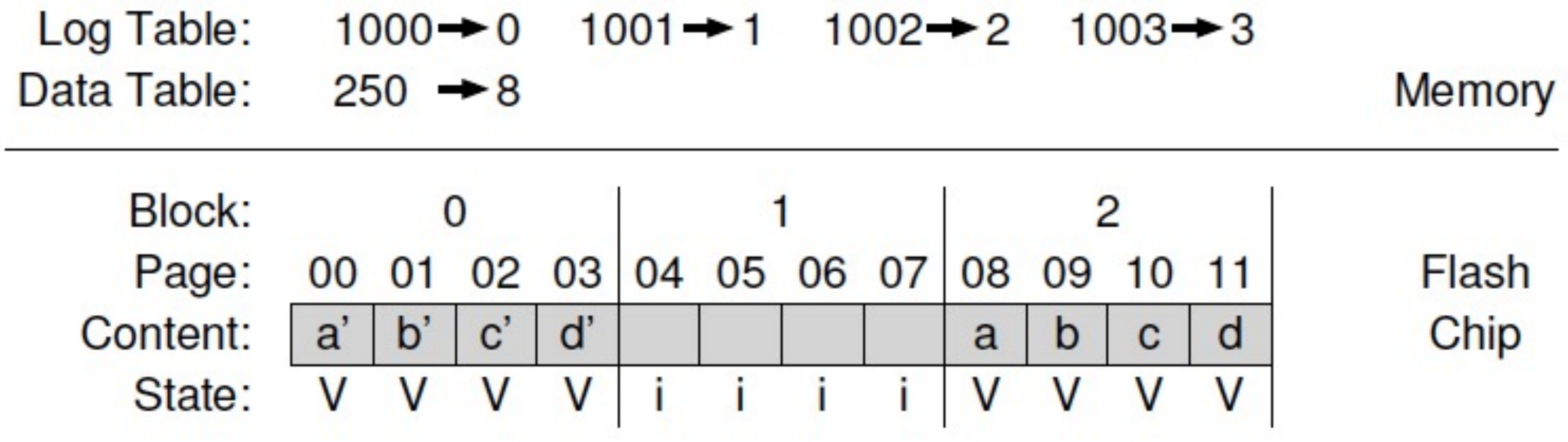
An example

- 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



An example

- 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).



Log Table: 1000 → 0 1001 → 1 1002 → 2 1003 → 3

Data Table: 250 → 8

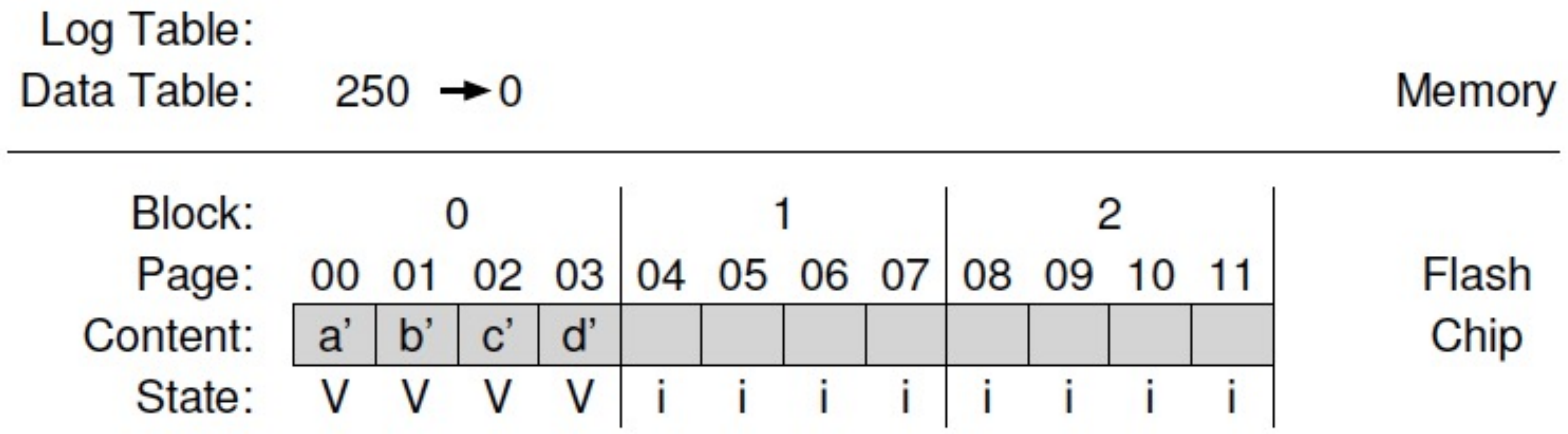
Memory

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

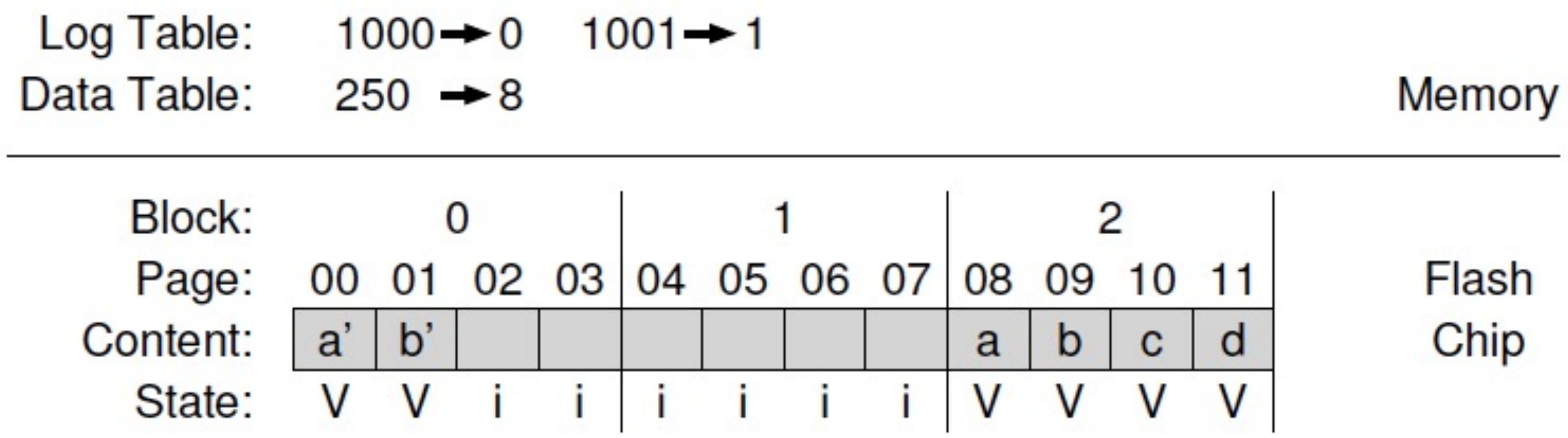
Flash Chip

Switch merge

- 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.



Partial merge



- Unfortunately, sometimes the FTL is not so lucky.
- Imagine 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.
- Partial merge: logical blocks 1002 and 1003 are read from physical block 2, and then appended to the log.
- 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

Full merge

- FTL must pull together pages from many other blocks to perform cleaning;
- 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:
  - 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.
  - merge must do the same for logical block 4, finding 5, 6, and 7 and reconciling them into a single physical block.
  - The same must be done for logical blocks 8 and 12, and then (finally), the log block A can be freed.