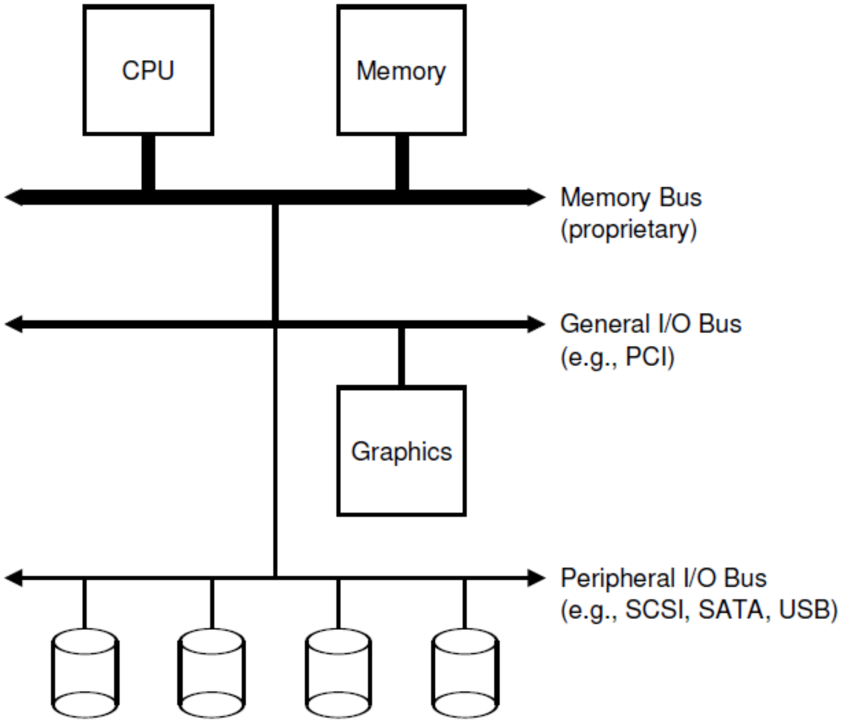
I/O Devices & SSD

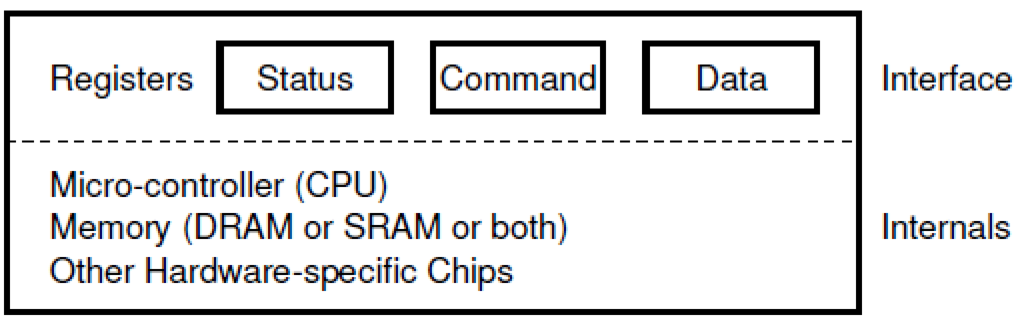
System Architecture

- **Hierarchical approach**
- **Memory bus**
  - CPU and memory
  - Fastest
- **I/O bus**
  - e.g., PCI
  - Graphics and higher-performance I/O devices
- **Peripheral bus**
  - SCSI, SATA, or USB
  - Connect many slowest devices



Device Structure

- **Hardware Interface**
  - Allows the system software to control its operation
  - status register: can be read to see the current status of the device
  - command register: can be written to tell the device to perform a certain task
  - data register: transfer data to/from the device
- **Internal Structure**
  - Implementation specific
  - Simple devices
    - have one or a few hardware chips to implement their functionality;
  - Complex devices
    - include a simple CPU running firmware, some general purpose memory, and other device-specific chips



Protocol

- By reading and writing internal registers of a device, the operating system can control device behavior.

```
While (STATUS == BUSY)
        ; // wait until device is not busy -> polling
Write data to DATA register
Write command to COMMAND register
        (Doing so starts the device and executes the command)
While (STATUS == BUSY)
        ; // wait until device is done with your request
```

- Inefficiencies and Inconveniences
  - Polling
    - Repeatedly reading the status register
  - Programmed I/O (PIO)
    - The main CPU is involved with the data movement

Lowering CPU Overhead With Interrupts

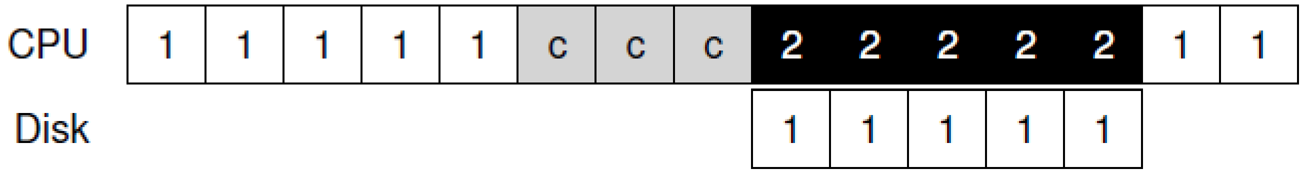
- **Interrupt**
  - OS can issue a request, put the calling process to sleep, and context switch to another task.
  - When the device is finally finished with the operation, it will raise a hardware interrupt, causing the CPU to jump into the OS at a pre-determined interrupt service routine (ISR)
  - The handler in operating system code that will finish the request
    - e.g., reading data and perhaps an error code from the device
  - Wake the process waiting for the I/O
  - Allow for overlap of computation and I/O
- Interrupt is not *always* the best solution
  - If a device is fast, it may be best to poll.
  - Otherwise, use interrupt

Interrupt Optimization

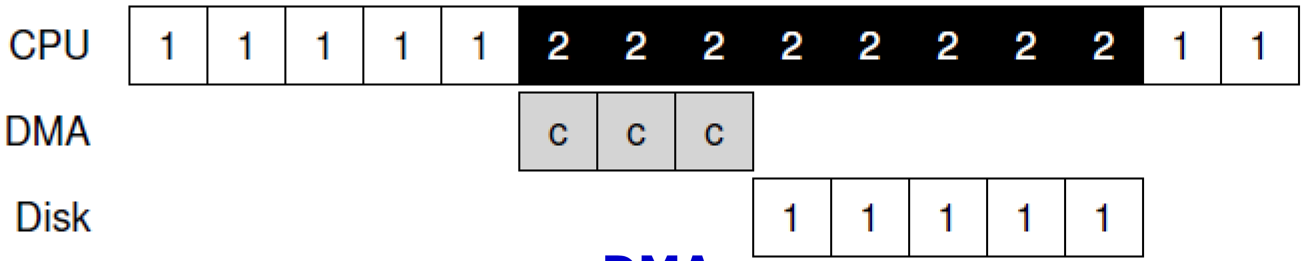
- **Unknown device: hybrid approach**
  - polls for a little while and then, if the device is not yet finished, uses interrupts. (two-phased approach)
- **Network systems**
  - When a huge stream of incoming packets each generate an interrupt, it is possible for the OS to livelock
    - only processing interrupts and never allowing a user-level process to run
  - Occasionally use polling to better control what is happening in the system (e.g., slashdot effect)
- **Interrupt Coalescing**
  - Multiple interrupts are coalesced into a single interrupt delivery
    - lower the overhead of interrupt processing
  - Long waiting
    - Many interrupts can be coalesced
    - Increase the latency of a request

More Efficient Data Movement With DMA

- OS programs the DMA engine
  - Source/destination address in memory or device
  - Data amount
- Other processes can be serviced
- When the DMA is complete, the DMA controller raises an interrupt, and the OS thus knows the transfer is complete.



Programmed IO



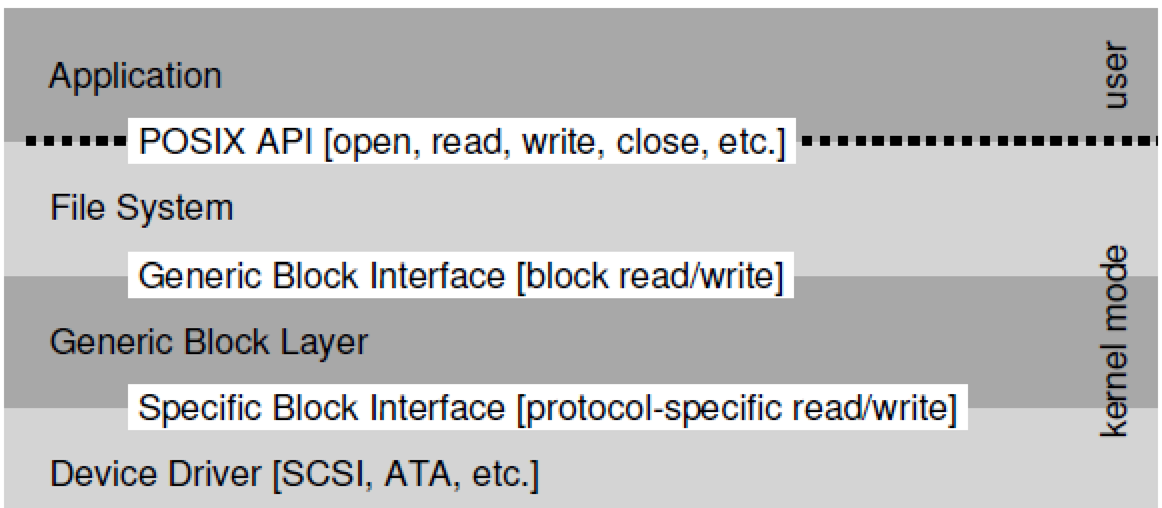
DMA

Methods Of Device Interaction

- **I/O instructions**
  - E.g., in and out on x86
  - To send data to a device, register → **port**.
  - Usually privileged instructions
- **Memory-mapped I/O**
  - Device registers are mapped to memory address space
  - load (to read) or store (to write)

Device Driver

- How can we keep most of the OS device-neutral, thus hiding the details of device interactions from major OS subsystems?
- Solution: **Abstraction**
  - The device driver in the OS know in detail how a device works
  - Any specifics of device interaction are encapsulated within D/D



Issues block requests to the generic block layer

Routes block requests to the appropriate D/D

handles the details of issuing the specific request

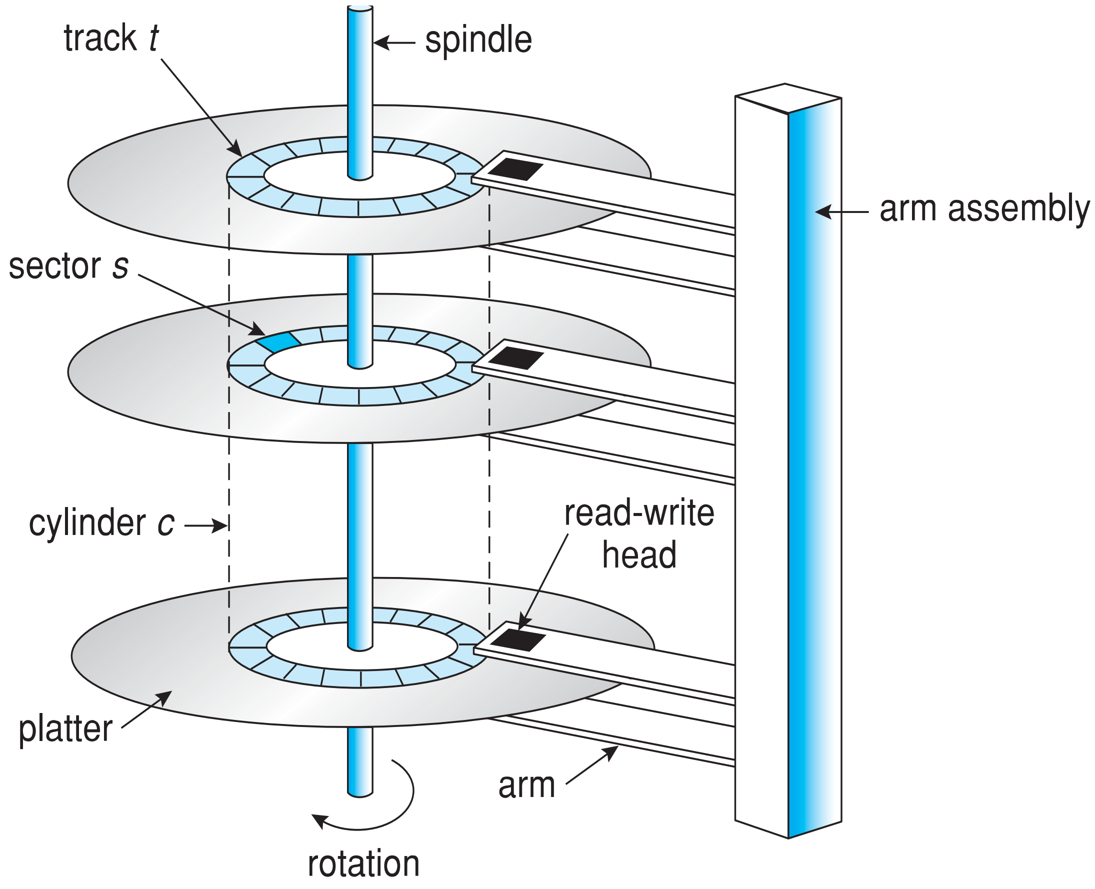
File System Stack

Device Driver

- Drawbacks of encapsulation
  - Even if a device has many special capabilities, it has to present a generic interface to the rest of the kernel, those special capabilities will go unused.
  - e.g., SCSI devices which have very rich error reporting

- Over 70% of OS code is found in device drivers
  - for any given installation, most of that code may not be active
  - have many more bugs and thus are a primary contributor to kernel crashes

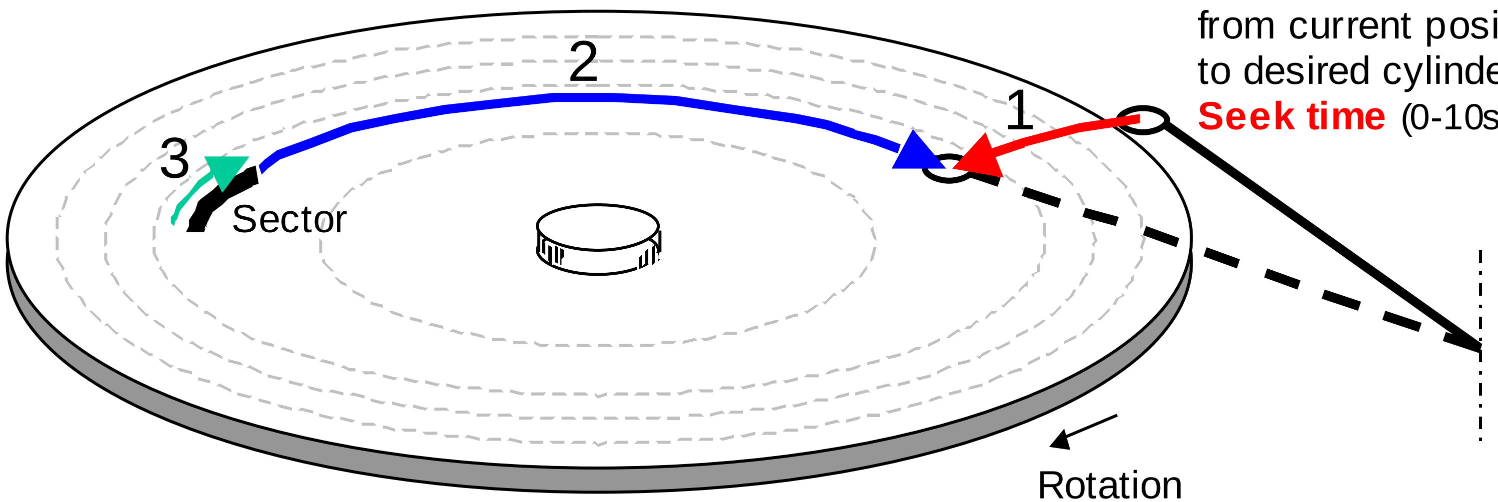
Disk System



track t
spindle
arm assembly
sector s
cylinder c
read-write
head
platter
arm
rotation

Disk System

- **Data access in disk system**
  - Seek time
  - Rotational delay (latency time)
  - Data transmission time

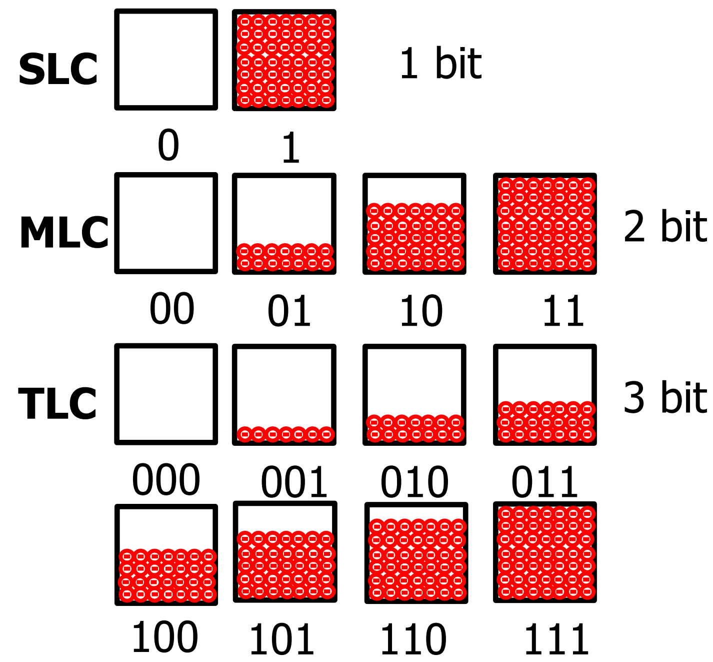


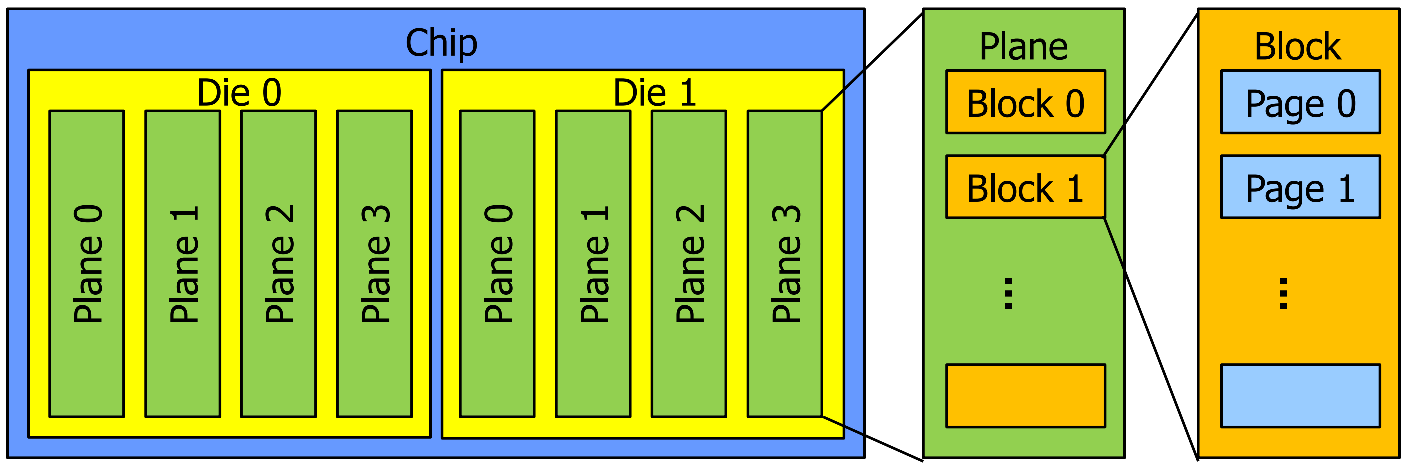
Disk System

- **Average time to access some target sector**
  - $T_{access} = T_{avg\text{-}seek} + T_{avg\text{-}rotation} + T_{avg\text{-}transfer}$
    - Seek time ($T_{avg\text{-}seek}$)
      - Time to position heads over cylinder containing target sector
      - Typical $T_{avg\text{-}seek}$ is 3~9ms
    - Rotational latency ($T_{avg\text{-}rotation}$)
      - Time waiting for first bit of target sector to pass under head
      - $T_{avg\text{-}rotation} = 1/2 \times 1/\text{RPMs} \times 60\text{s}/1\text{min}$
      - Typical $T_{avg\text{-}rotation}$ = 1~4ms
    - Transfer time ($T_{avg\text{-}transfer}$)
      - Time to read the bits in the target sector
      - $T_{avg\text{-}transfer} = 1/\text{RPM} \times 1/(\text{avg. \# sectors/track}) \times 60\text{s}/1\text{min}$

Flash-based SSD

- **NAND flash memory**
  - Page: read/write unit, 4KB~16KB
  - Block: erase unit, erase-before-write, 128 KB or 256 KB
  - Limited lifetime: wear out
  - SLC, MLC, TLC





Flash Operations

- **Read (a page)**
  - Random access
- **Erase (a block)**
  - set each bit to the value 1
- **Program (a page)**
  - change some of the 1's within a page to 0's

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

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

Erase Block

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

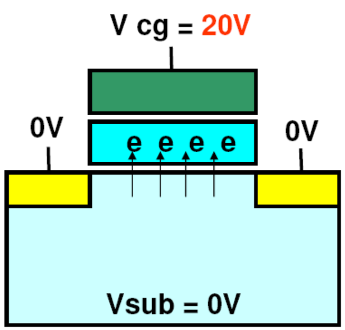
Program Page 0

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

Previous contents of Page 1, 2, 3 are all gone!
They must be moved before the block erase operation.

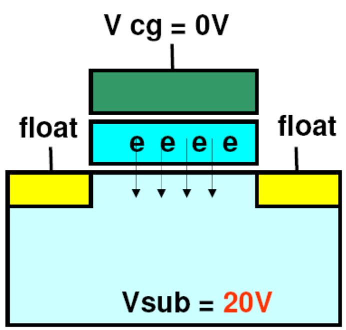
Flash Performance And Reliability

- **Wear out**
  - Erased/program slowly accrues a little bit of extra charge.
  - As that extra charge builds up, the block becomes unusable.
  - MLC block: 10,000 P/E (Program/Erase) cycle lifetime;
  - SLC block: 100,000 P/E cycles.
- **Disturbance**
  - When accessing a particular page within a flash, it is possible that some bits get flipped in neighboring pages
  - read disturbs or program disturbs



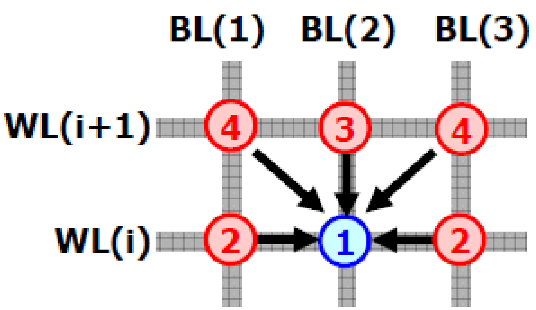
Program
F-N Tunneling

Off cell
(Solid-0)



Erase
F-N Tunneling

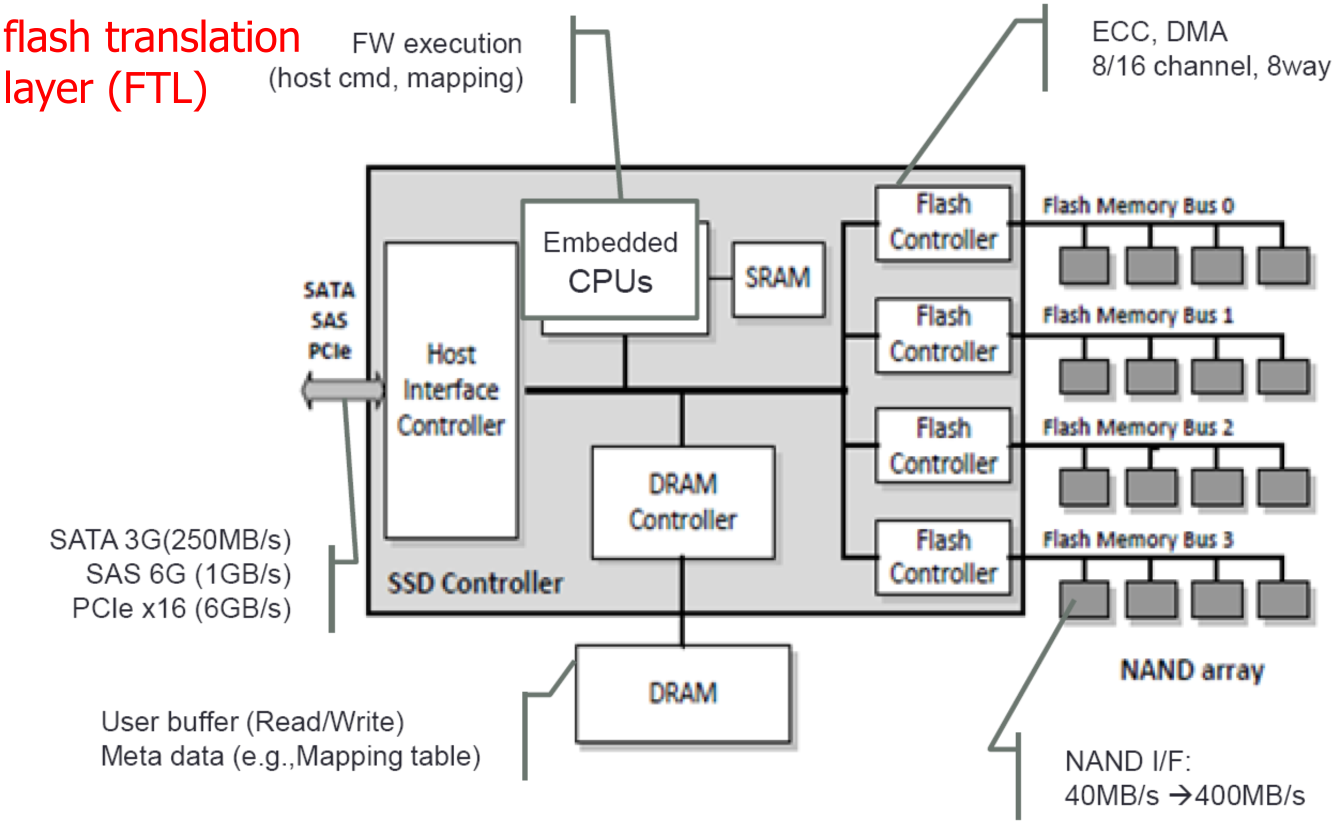
On cell
(Solid-1)



cell-to-cell interference

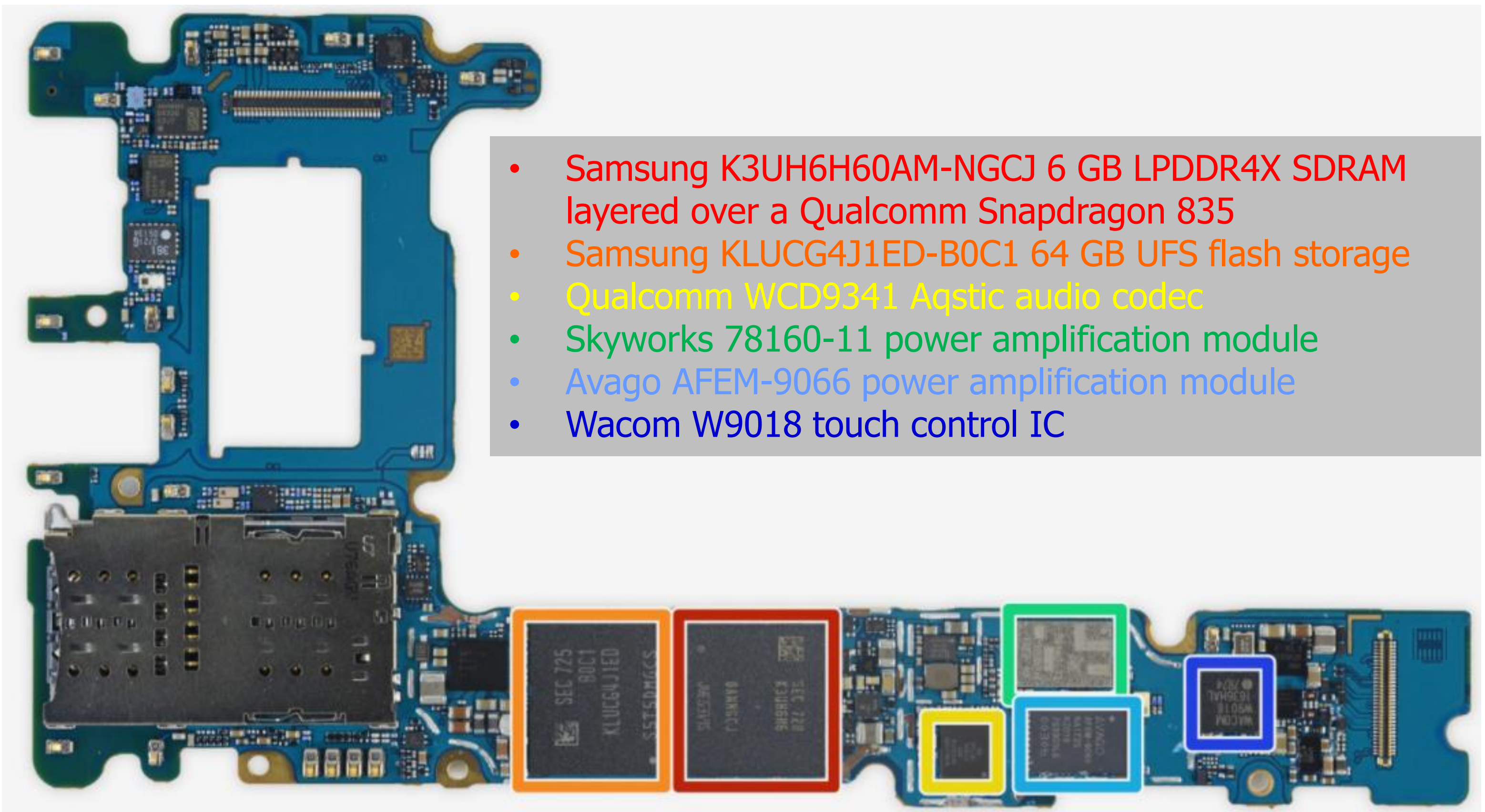
From Raw Flash to Flash-Based SSDs

- Flash-based SSD provides standard block interface atop the raw flash chips inside it.
  - Sector (512 KB) read/write

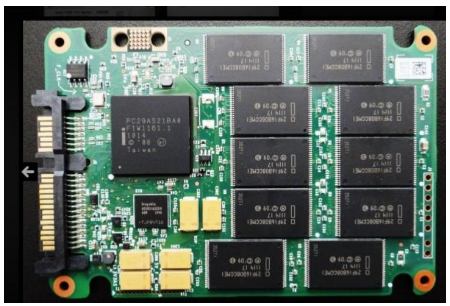


Samsung Galaxy Note8

- Samsung K3UH6H60AM-NGCJ 6 GB LPDDR4X SDRAM layered over a Qualcomm Snapdragon 835
- Samsung KLUCG4J1ED-B0C1 64 GB UFS flash storage
- Qualcomm WCD9341 Aqstic audio codec
- Skyworks 78160-11 power amplification module
- Avago AFEM-9066 power amplification module
- Wacom W9018 touch control IC

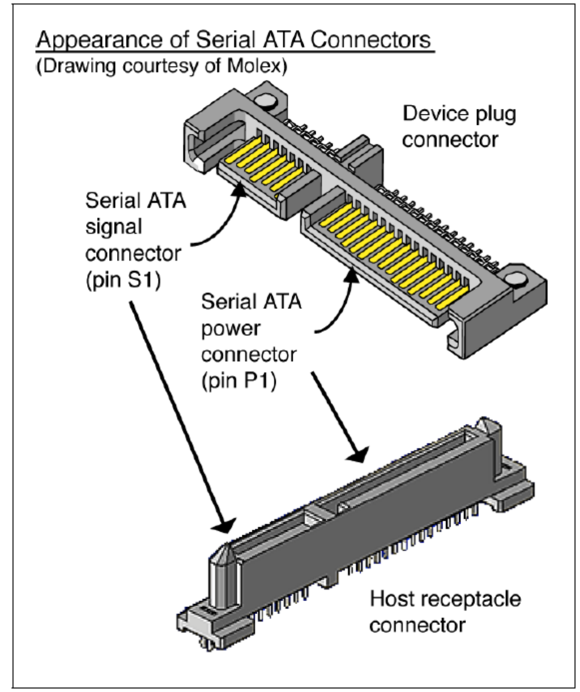


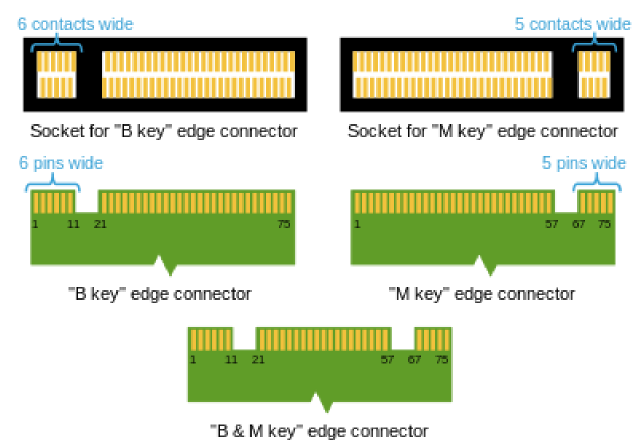
Solid-State Disk



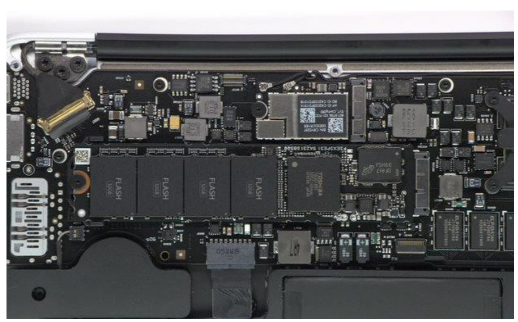
SATA SSD

NVMe SSD





M.2 NVMe



MacBook Air

FTL

- 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
- Performance issues
  - Utilizing multiple flash chips in parallel
  - Reduce write amplification
    - total write issued to flash chips/total write issued by host
- Reliability issues
  - Wear leveling
    - spread writes across the blocks of the flash as evenly as possible
    - ensure that all of the blocks of the device wear out at roughly the same time
  - Program disturbance
    - sequential-programming
      - program pages within an erased block in order
      - low page → high page

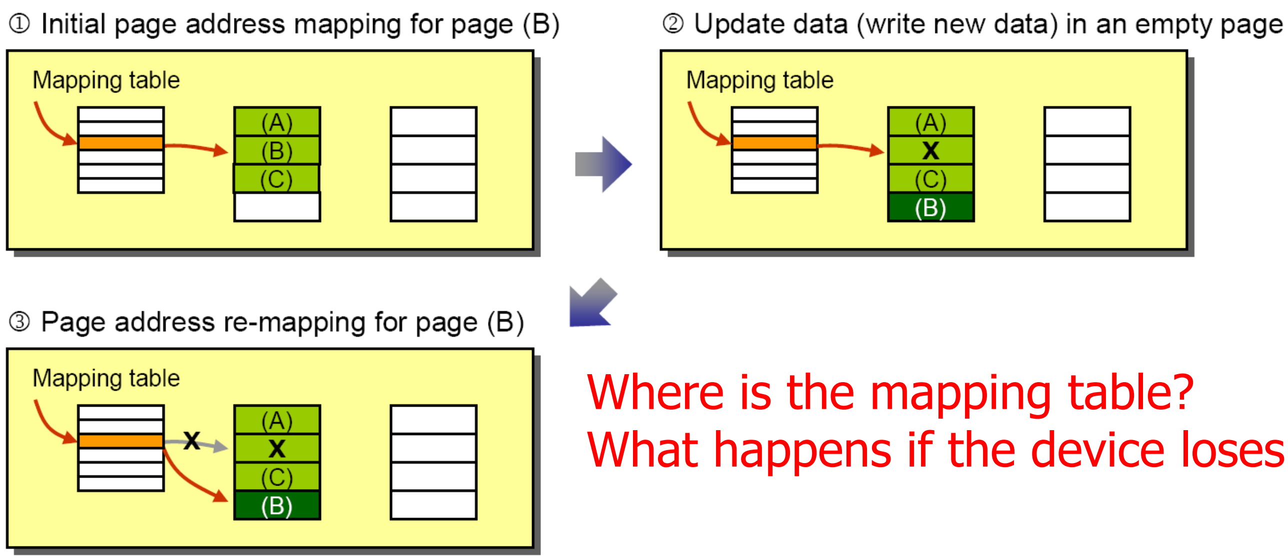
FTL: Direct Mapped

- **A write to logical page N**
  - Read in the entire block that page N is contained within
  - Erase the block
  - Program the old pages as well as the new one.
  - Read-modify-write approach

- **Poor performance & high write amplification**
- **The physical blocks containing popular data will quickly wear out.**

A Log-Structured FTL

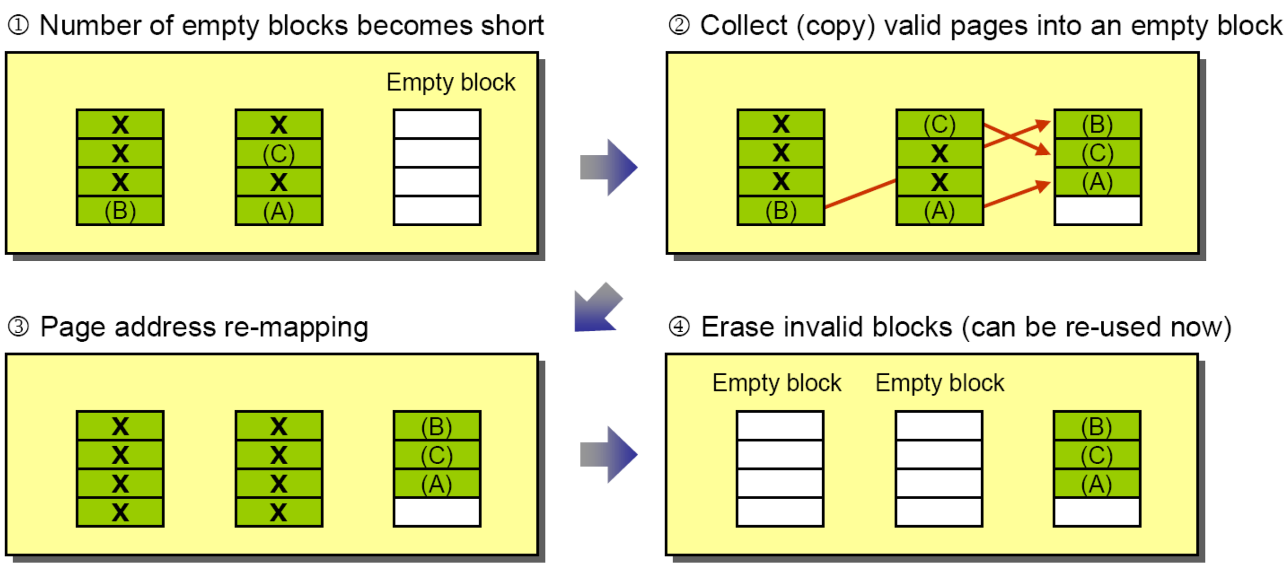
- Upon a write to logical block N
  - The device appends the write to the next free spot in the currently-being-written-to block
  - no overwrite, out-of-place update
  - To allow for subsequent reads of block N, the device keeps a mapping table, which stores the physical address of each logical block.
- Problems: GC & mapping table



Where is the mapping table?
What happens if the device loses power?

Garbage Collection

- dead-block reclamation to provide free space for new writes
  - 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
  - (finally) reclaim the entire block for use in writing
- Must know
  - Whether each page is live or dead ➔ mapping table or bitmap
  - The number of live pages in each block for victim selection
- Overprovision can delay GC, and push to the background



Mapping Table Size

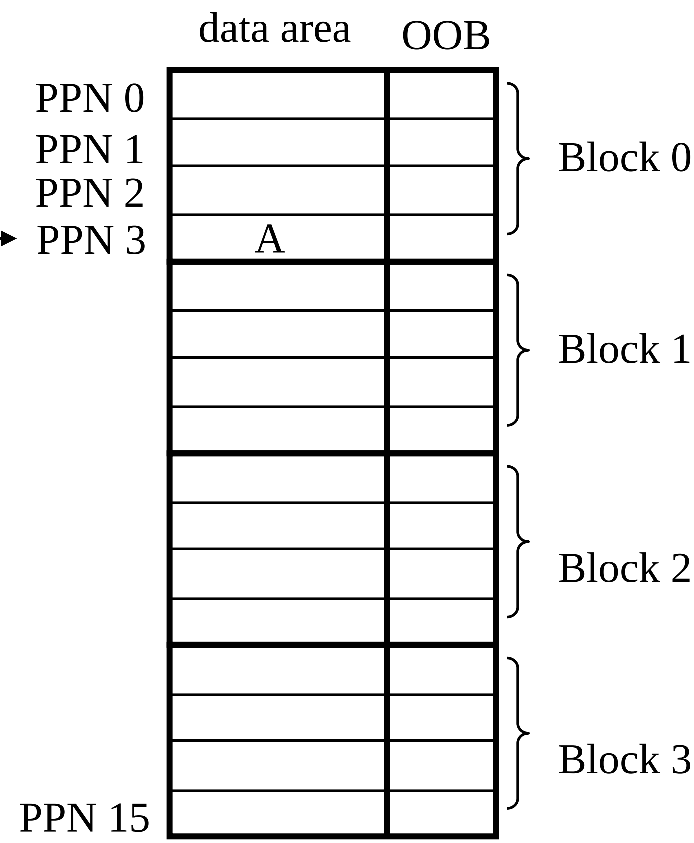
- Page-level mapping
  - With a large 1-TB SSD, a single 4-byte entry per 4-KB page results in 1 GB of memory needed the device

LPN: logical page number
PPN : physical page number

“write(9, A)”

LPN	PPN
0	12
1	11
2	10
3	9
4	8
5	7
6	6
7	5
8	4
9	3
10	2
11	1
12	0

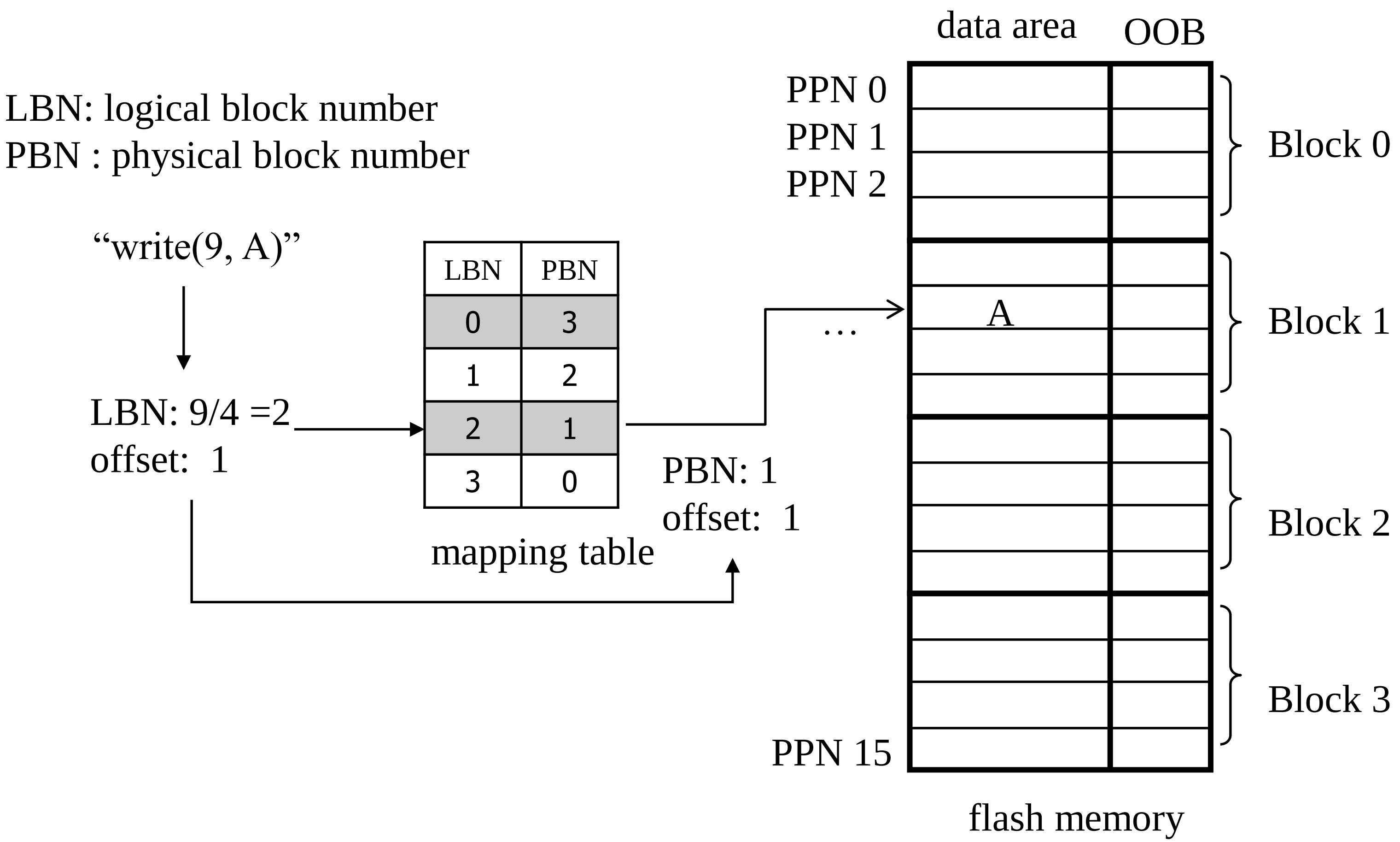
mapping table



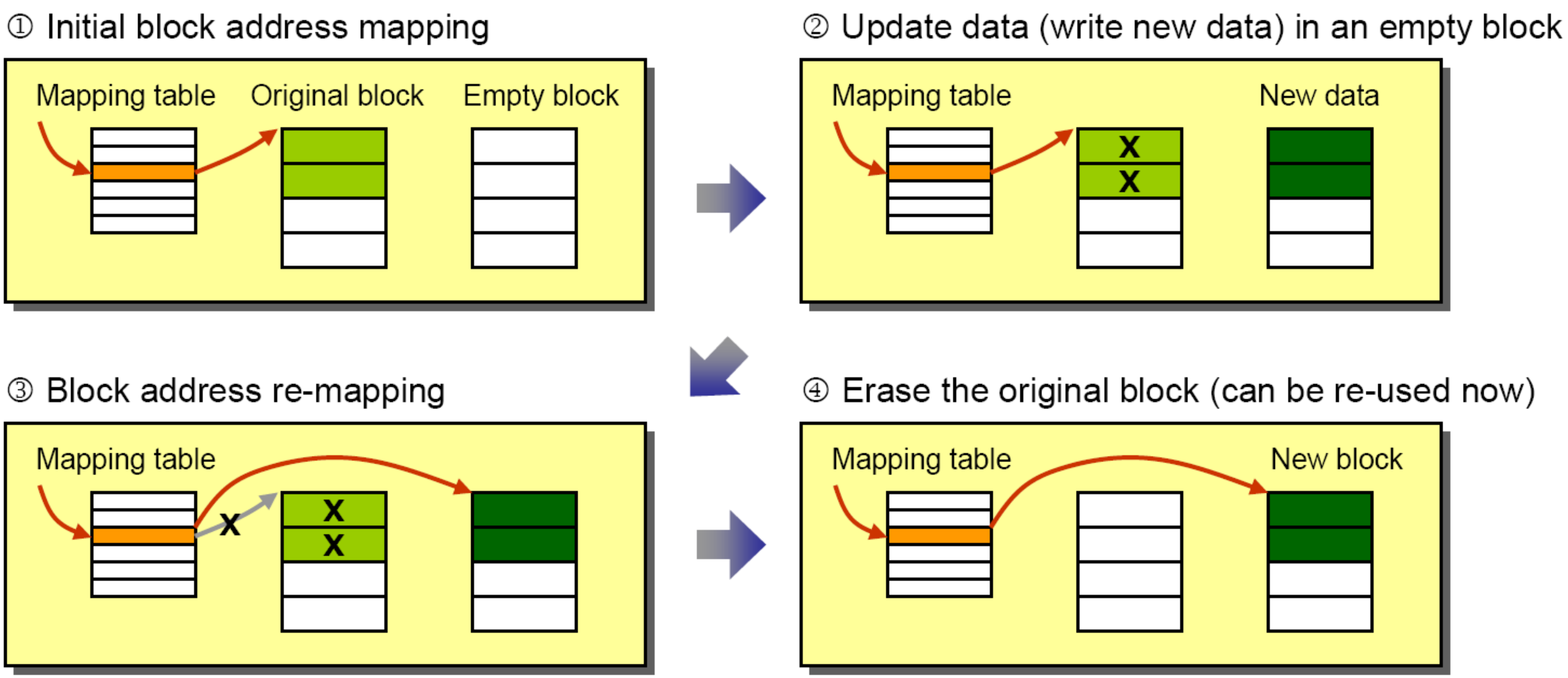
flash memory

Mapping Table Size

- Block-level mapping



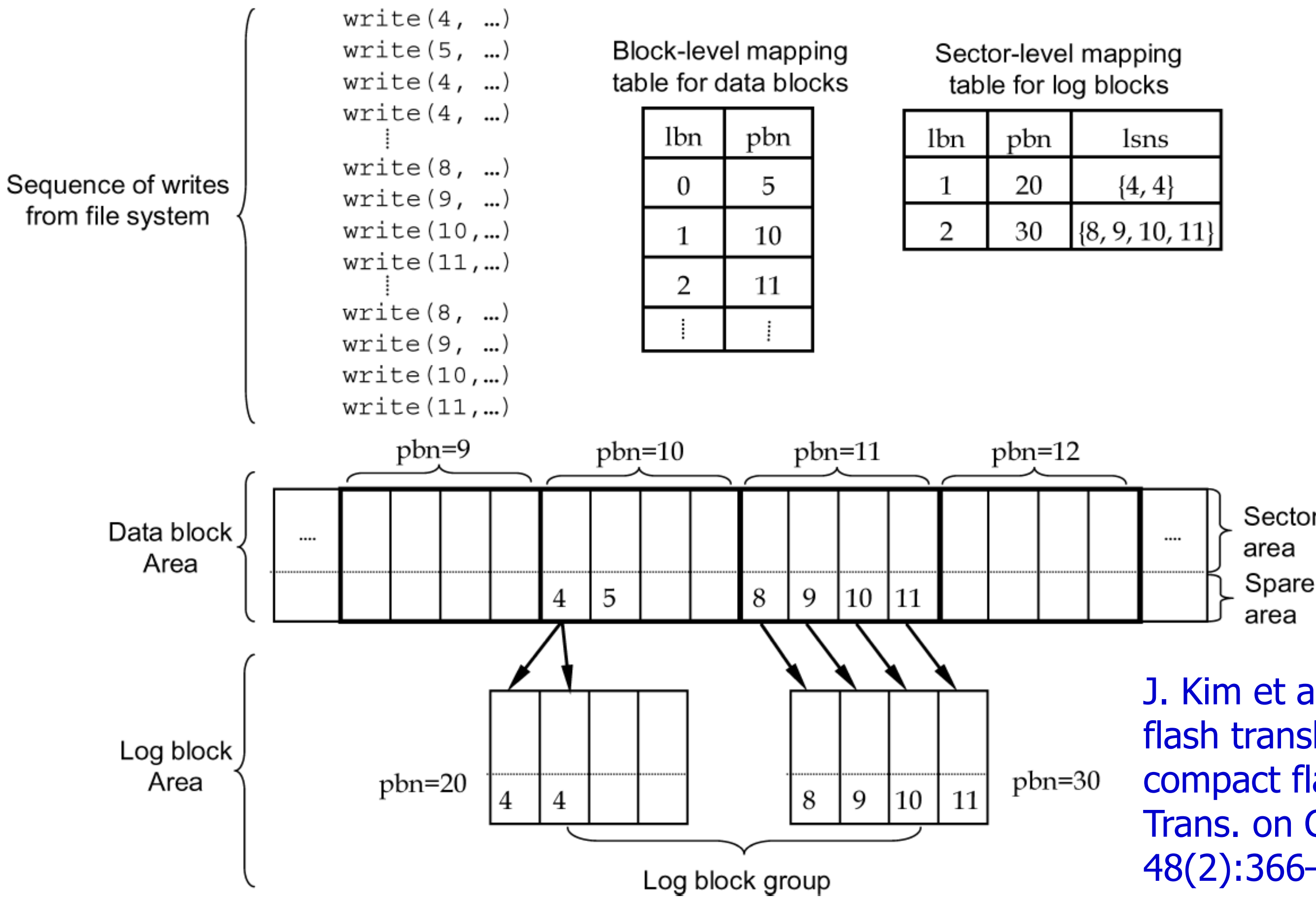
Update at Block Mapping



Hybrid Mapping

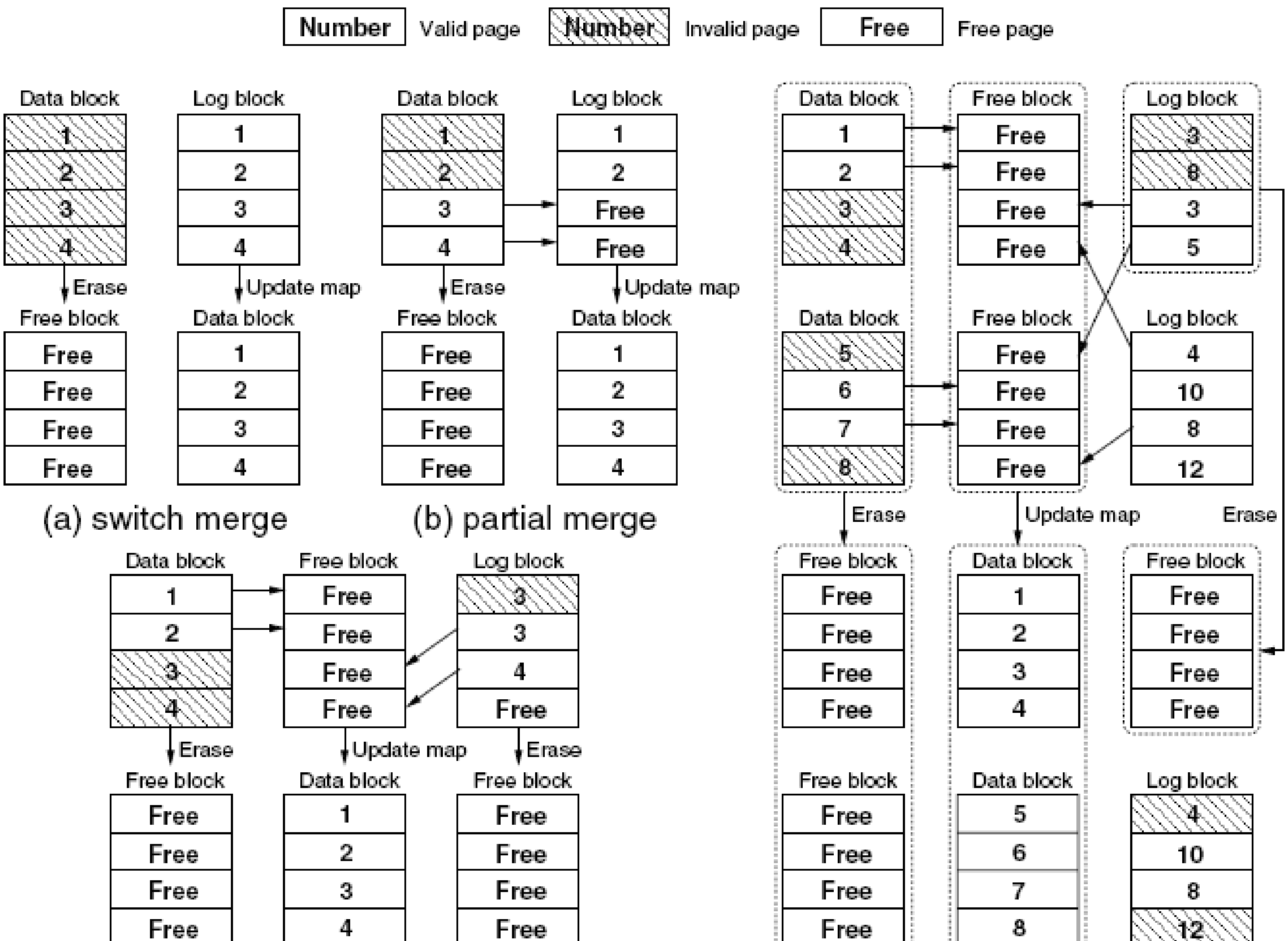
- To enable flexible writing but also reduce mapping costs
- FTL keeps a few log blocks and directs all writes to them
  - Page-level mapping (log table)
  - Small number of log blocks ➔ small-sized log table
- Normal data blocks
  - Block-level mapping (data table) ➔ small-sized data table
- For read request
  - first consult the log table; if not found, consult the data table
- To keep the number of log blocks small, log blocks must be merged with data blocks if no free space is available in log blocks.

Hybrid Mapping (BAST)



J. Kim et al., “A space efficient flash translation layer for compact flash systems,” IEEE Trans. on Consumer Electronics, 48(2):366–375, 2002.

Hybrid Mapping – Merge Operation



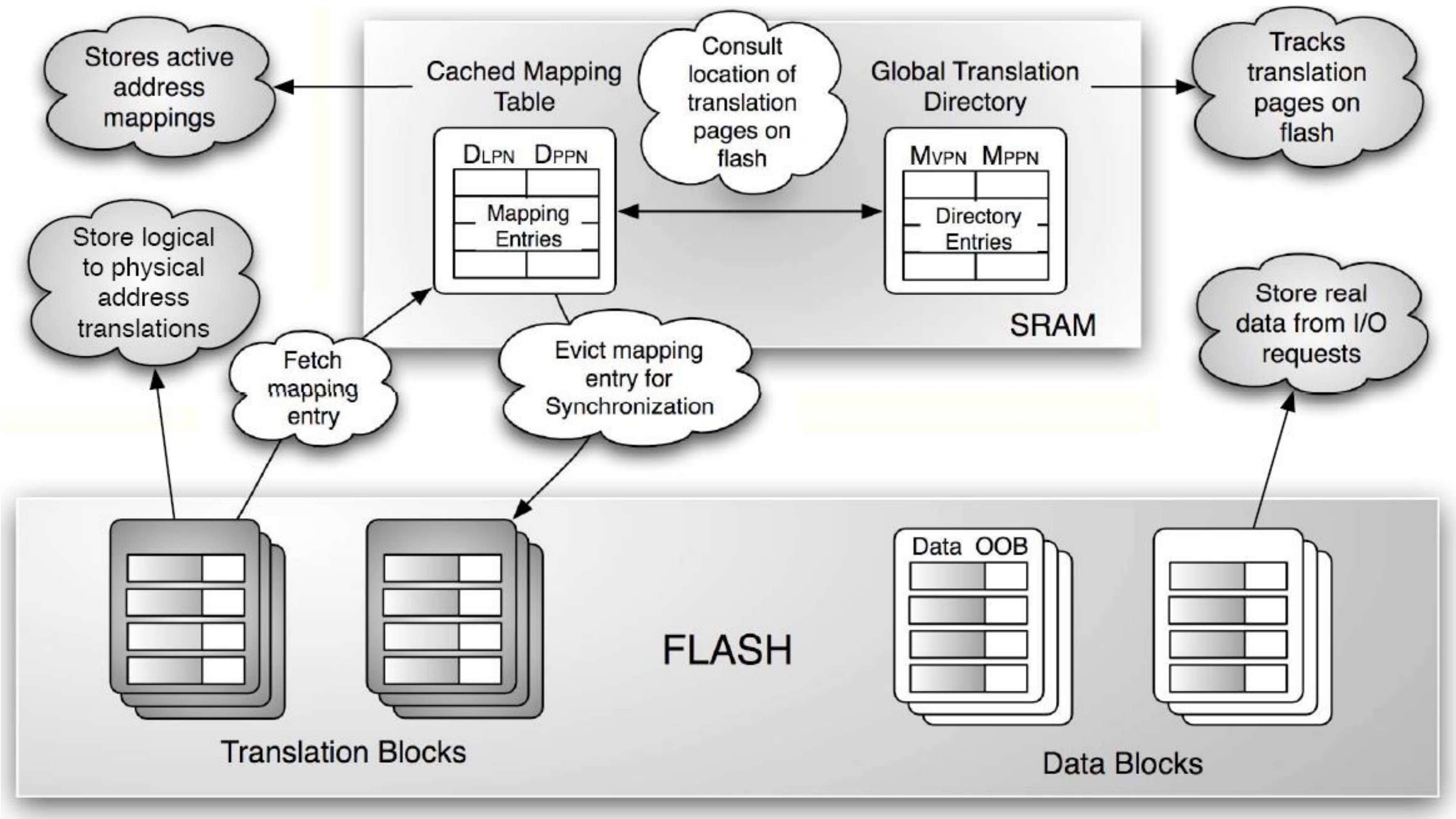
(a) switch merge

(b) partial merge

(c) full merge in BAST

(d) full merge in FAST

DFTL



Stores active address mappings
Cached Mapping Table
Consult location of translation pages on flash
Global Translation Directory
Tracks translation pages on flash
DLPN DPPN
Mapping Entries
MVPN MPPN
Directory Entries
SRAM
Store logical to physical address translations
Fetch mapping entry
Evict mapping entry for Synchronization
Store real data from I/O requests
Data OOB
FLASH
Translation Blocks
Data Blocks

SSD Performance

- Random I/O
  - Large difference between SSD and HDD
- Sequential I/O
  - Much less of a difference between SSD and HDD
  - HDD still a good choice
- SSD random read performance is not as good as SSD random write performance
  - Write buffer
- Performance difference between sequential and random in SSD
  - many of the file system techniques for HDDs are still applicable to 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