CS250B: Modern Computer Systems

Storage Technologies Introduction

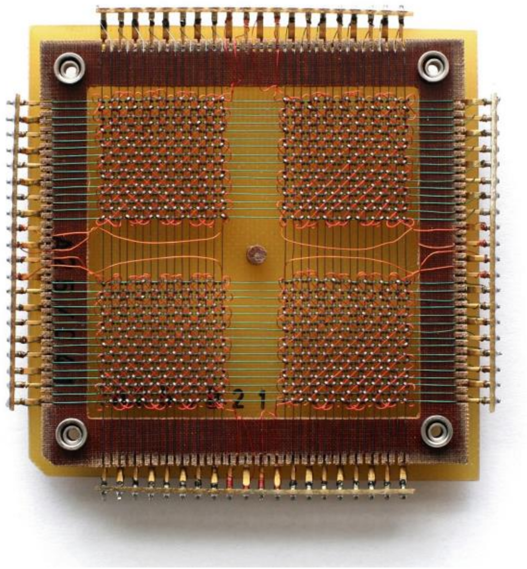
Sang-Woo Jun

UCI

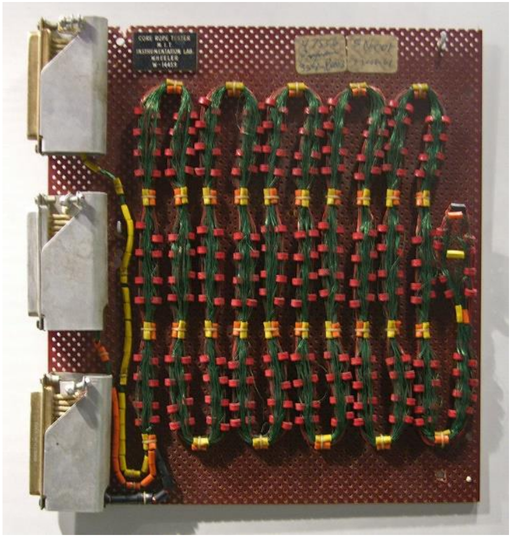
Storage Used To be a Secondary Concern

- Typically, storage was not a first order citizen of a computer system
  - As allured by its name “secondary storage”
  - Its job was to load programs and data to memory, and disappear
  - Most applications only worked with CPU and system memory (DRAM)
  - Extreme applications like DBMSs were the exception
- Because conventional secondary storage was very slow
  - Things are changing!

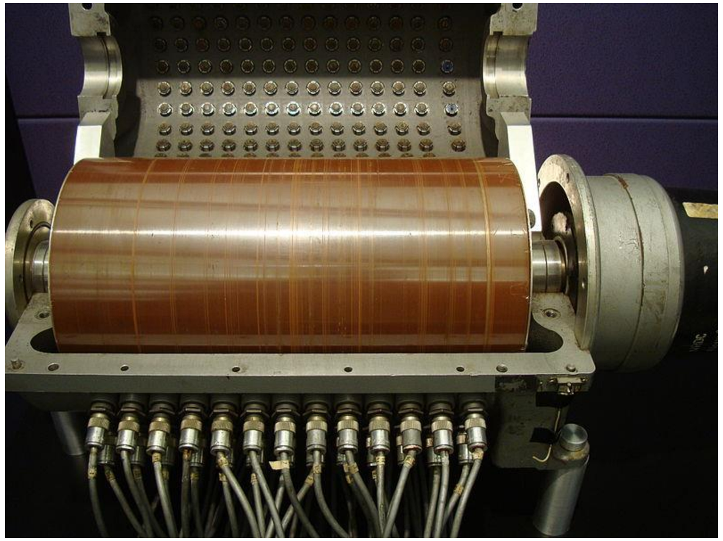
Some (Pre)History



Magnetic core memory
1950~1970s
(1024 bits in photo)



Rope memory (ROM) 1960’s
72 KiB per cubic foot!
Hand-woven to program the
Apollo guidance computer



Drum memory
100s of KiB
1950’s

Photos from Wikipedia

Some (More Recent) History

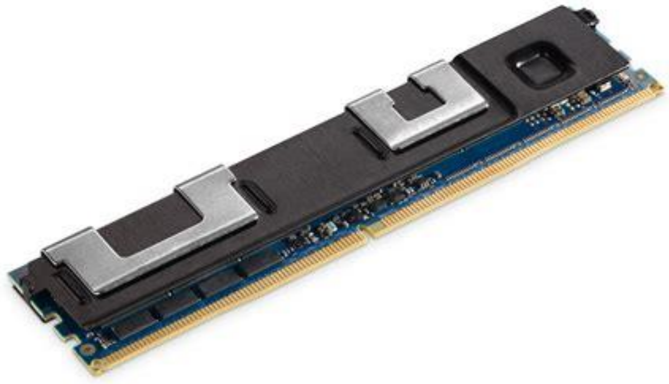
Floppy disk drives
1970's~2000's
100 KiBs to 1.44 MiB

Hard disk drives
1950's to present
MBs to TBs

Photos from Wikipedia

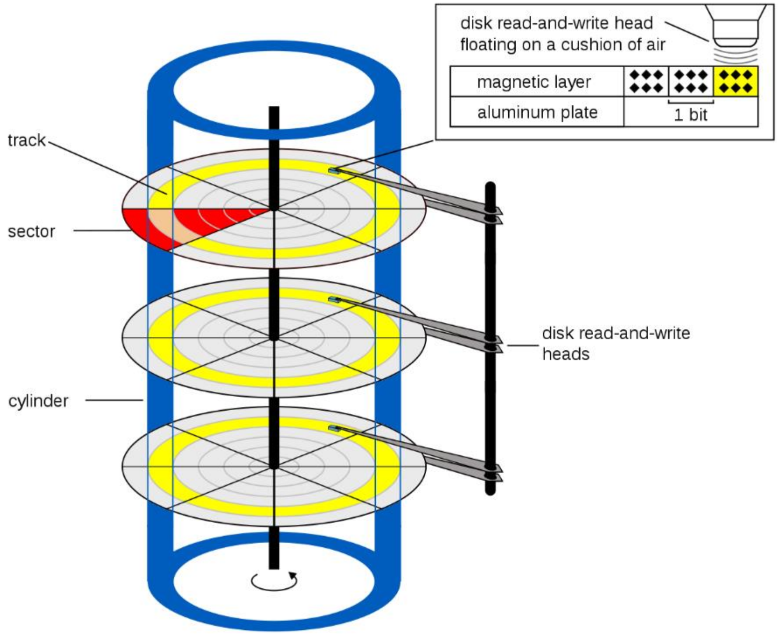
Some (Current) History

Solid State Drives
2000's to present
GB to TBs



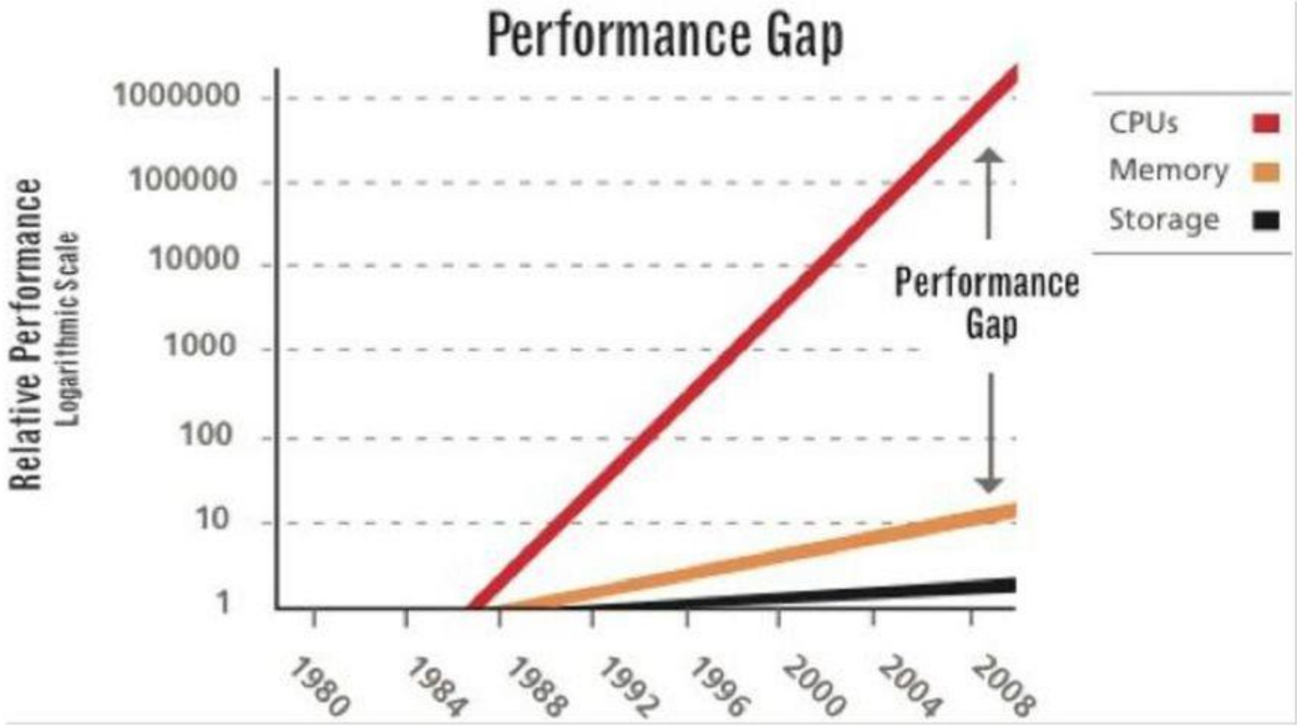
Non-Volatile Memory
2010's to present
GBs

Hard Disk Drives



- Dominant storage medium for the longest time
  - Still the largest capacity share
- Data organized into multiple magnetic platters
  - Mechanical head needs to move to where data is, to read it
  - Good sequential access, terrible random access
    - 100s of MB/s sequential, maybe 1 MB/s 4 KB random
  - Time for the head to move to the right location (“seek time”) may be ms long
    - 1000,000s of cycles!
- Typically “ATA” (Including IDE and EIDE), and later “SATA” interfaces
  - Connected via “South bridge” chipset

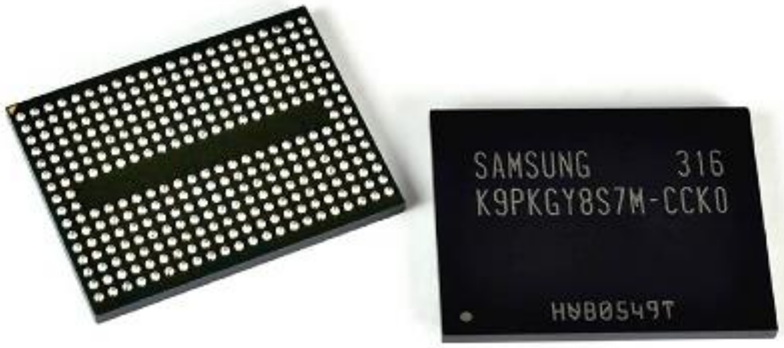
Big picture: performance gap

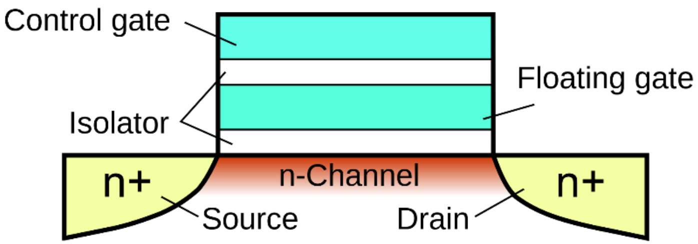


Ding Yuan, “Operating Systems ECE344 Lecture 11: File System”

Solid State Drives

- "Solid state", meaning no mechanical parts, addressed much like DRAM
  - Relatively low latency compared to HDDs (10s of us, compared to ms)
  - Easily parallelizable using more chips – Multi-GB/s
- Simple explanation: flash cells store state in a "floating gate" by charging it at a high voltage
  - High voltage acquired via internal charge pump (no need for high V input)

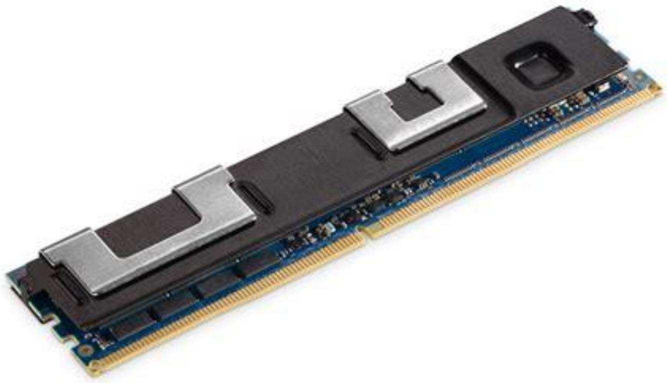




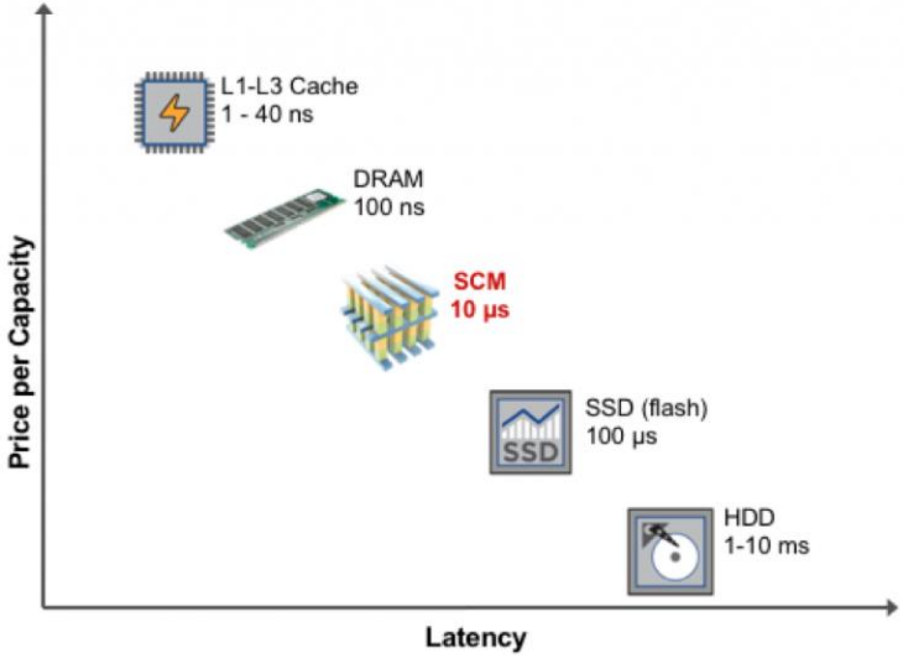
Solid State Drives

- Serial ATA (SATA) interface, over Advanced Host Controller Interface (AHCI) standard
  - Used to be connected to south bridge,
  - Up to 600 MB/s, quickly became too slow for SSDs
- Non-Volatile Memory Express (NVMe)
  - PCIe-attached storage devices – multi-GB/s
  - Redesigns many storage support components in the OS for performance

Non-Volatile Memory



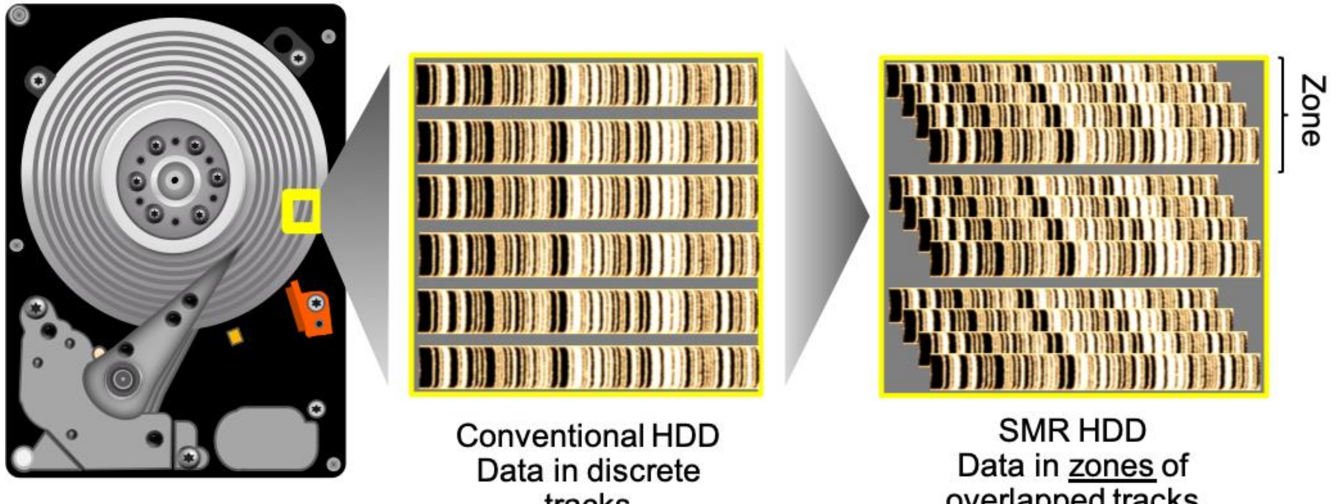
- Naming convention is a bit vague
  - Flash storage is also often called NVM
    - Storage-Class Memory (SCM)?
  - Anything that is non-volatile and fast?
- Too fast for even PCIe/NVMe software
  - Plugged into memory slots, accessed like memory
  - e.g., Intel Optane
- But not quite as fast as DRAM
  - Latency/Bandwidth/Access granularity
  - Usage under active research!



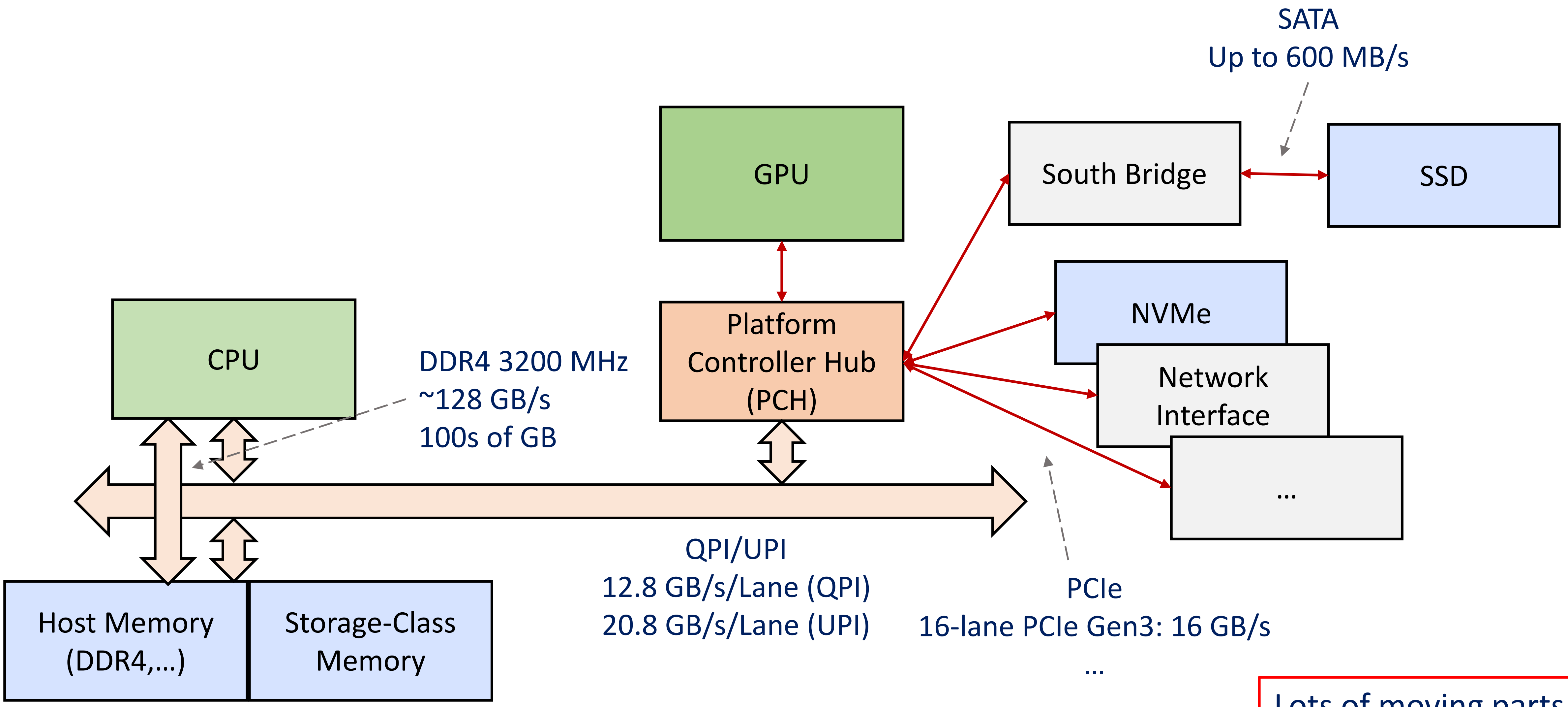
Souce: NetApp blog, “Storage Class Memory: What’s Next in Enterprise Storage,” 2018

Shingled Magnetic Recording (SMR): Larger/Slower Magnetic Disks

- Hard disk scaling was slowing due to limit in density scaling
  - Limit in making data write header smaller
- SMR: Tracks on a platter are overlapped to improve density
  - Organized into “zone” groups of tracks
  - Writing earlier tracks of a zone can destroy data in later zones
  - Reading is largely unchanged, because read header width is narrower
- Slower speed, lower resilience
- More storage per dollar



System Architecture Snapshot



Lots of moving parts!

Storage for Analytics

Fine-grained,
Irregular access

Terabytes in size

TB of DRAM

$$$ $8000/TB, 200W

The goal:

$ $400/TB, 10W

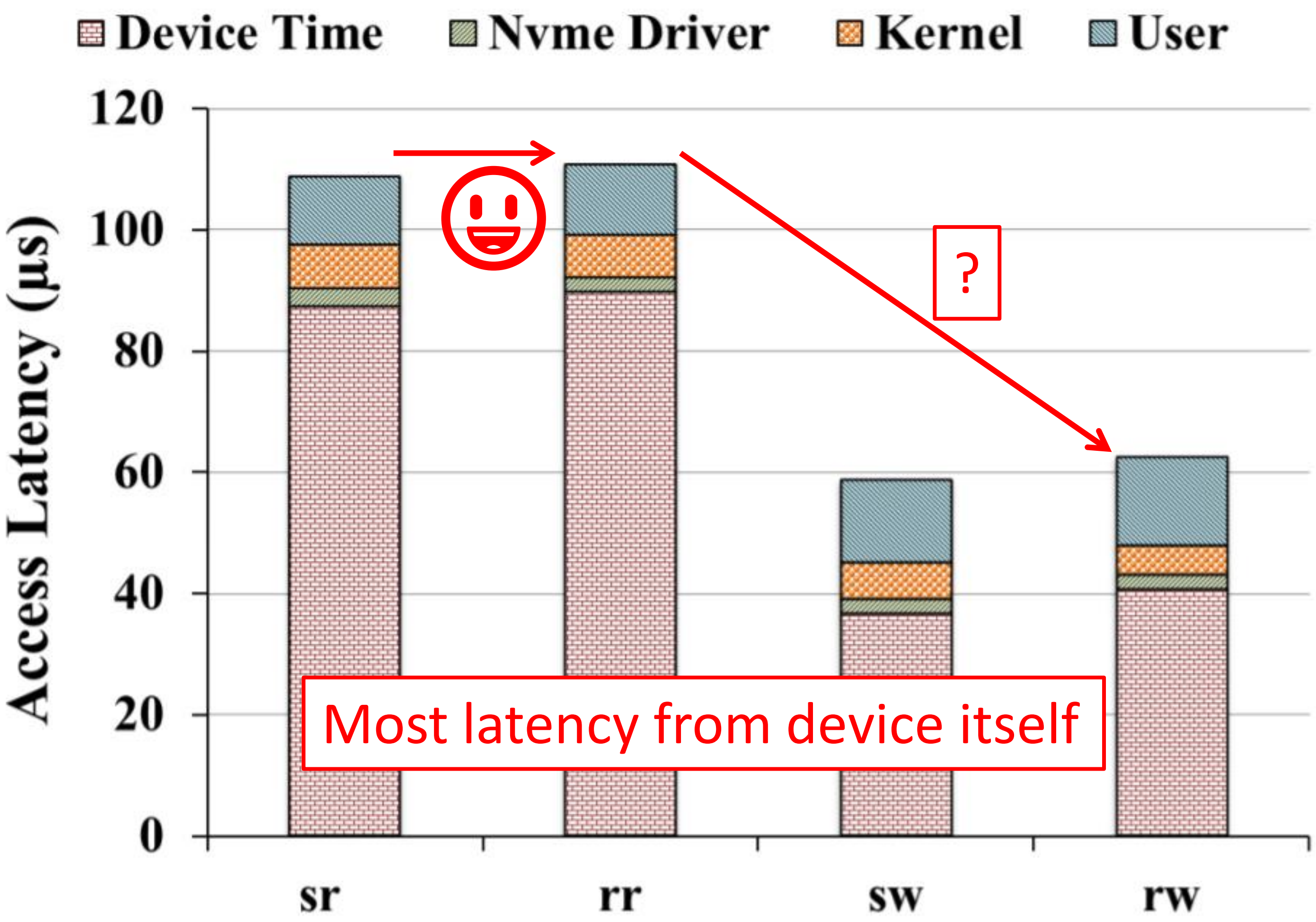
$ $150/TB, 2W

Performance Challenges in Flash Storage 1

	Flash	DRAM
Bandwidth:	0.6-10 GB/s	~50 GB/s

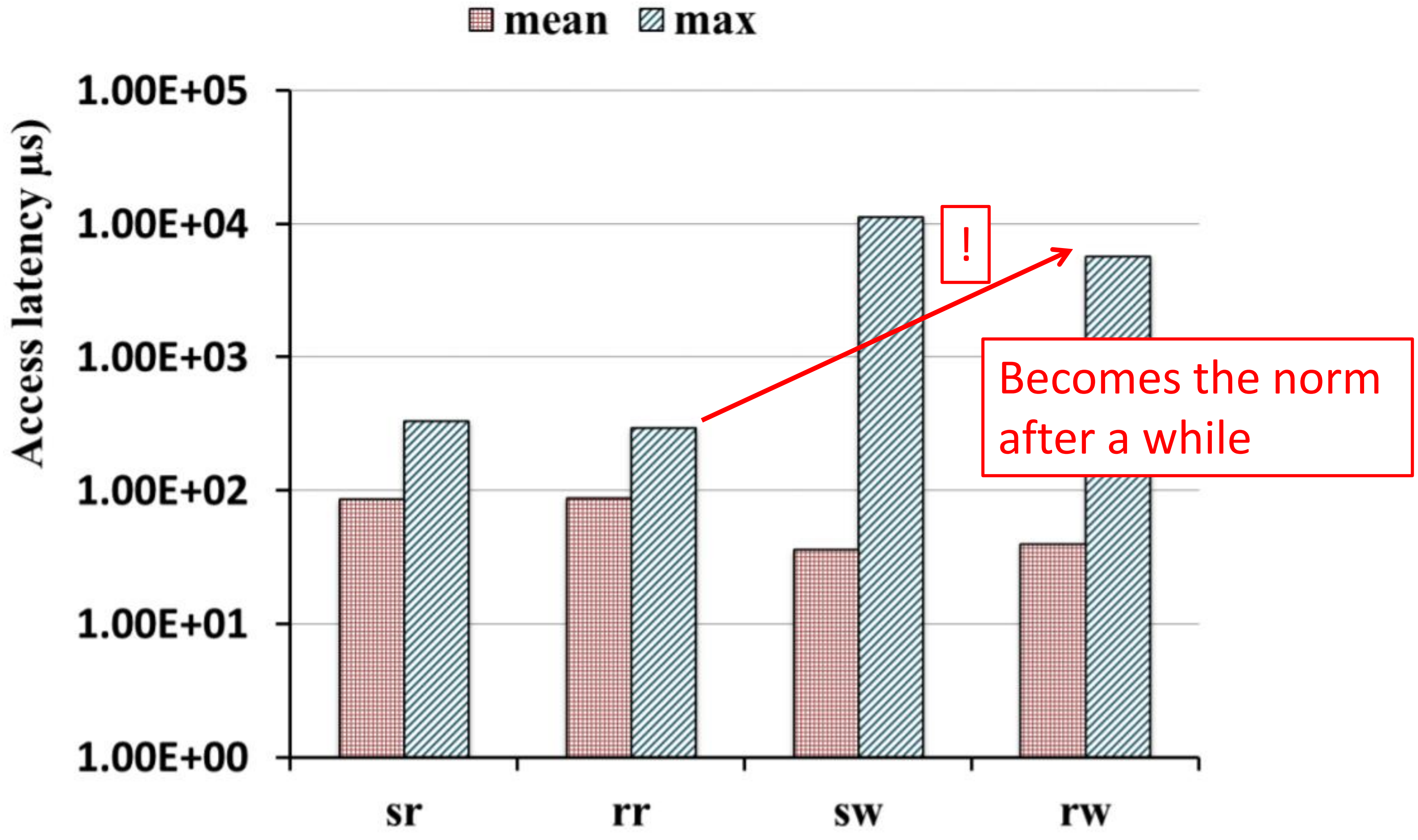
Not bad! Considering local DRAM and RAID

Performance Challenges in Flash Storage 2



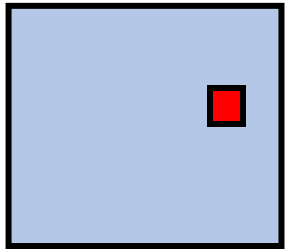
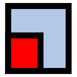
Xu et. al., “Performance Analysis of NVMe SSDs and their Implication on Real World Databases” SYSTOR 2015

Performance Challenges in Flash Storage 2



Xu et. al., “Performance Analysis of NVMe SSDs and their Implication on Real World Databases” SYSTOR 2015

Performance Challenges in Flash Storage 3

	Flash	DRAM
Bandwidth:	0.6-10 GB/s	~50 GB/s
Latency:	~100 µs	~15 ns
Access Granularity:	 8192 Bytes	 128 Bytes

* Wastes performance by
not using most of fetched page

CS250B: Modern Computer Systems

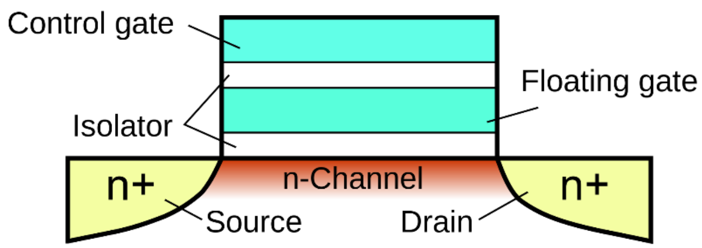
Flash Storage

Sang-Woo Jun

UCI

Flash Storage

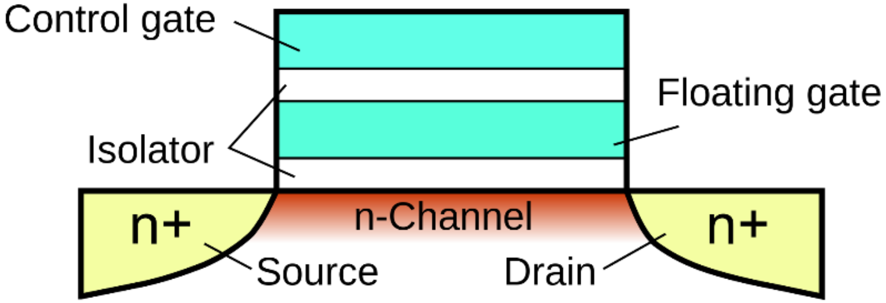
- Most prominent solid state storage technology
  - Few other technologies available at scale (Intel X-Point one of few examples)
- Flash cells store data in "floating gate" by charging it at high voltage*
- Cells configured into NOR-flash or NAND-flash types
  - NOR-flash is byte-addressable, but costly – In phones and embedded devices
  - NAND-flash is "page" addressable, but cheap – In secondary storage
- Many bits can be stored in a cell by differentiating between the amount of charge in the cell
  - Single-Level Cell (SLC), Multi (MLC), Triple (TLC), Quad (QLC)
  - Typically cheaper, but slower with more bits per cell

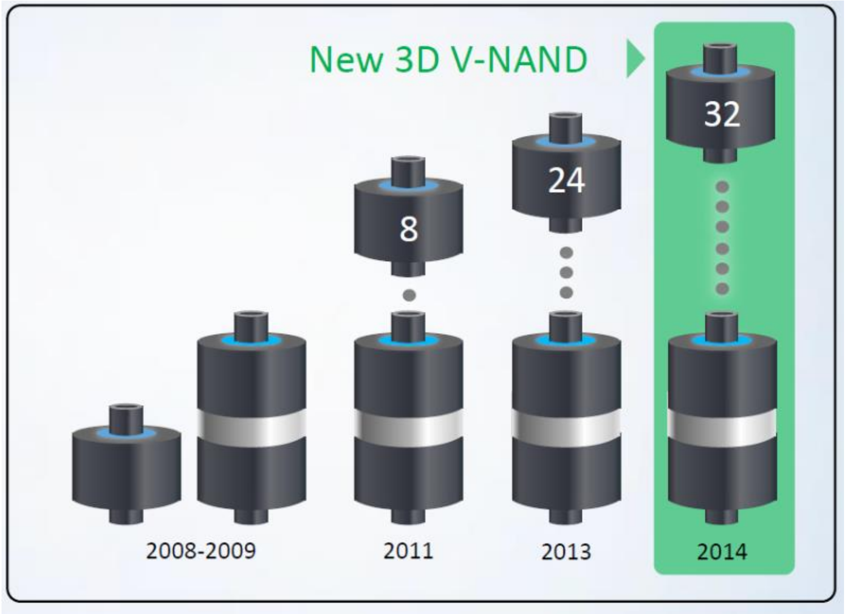


*Variations exist, but basic idea is similar

3D NAND-Flash

- NAND-Flash scaling limited by charge capacity in a floating gate
  - Only a few hundred can fit at current sizes
  - Can’t afford to leak even a few electrons!
- Solution: 3D stacked structure… For now!



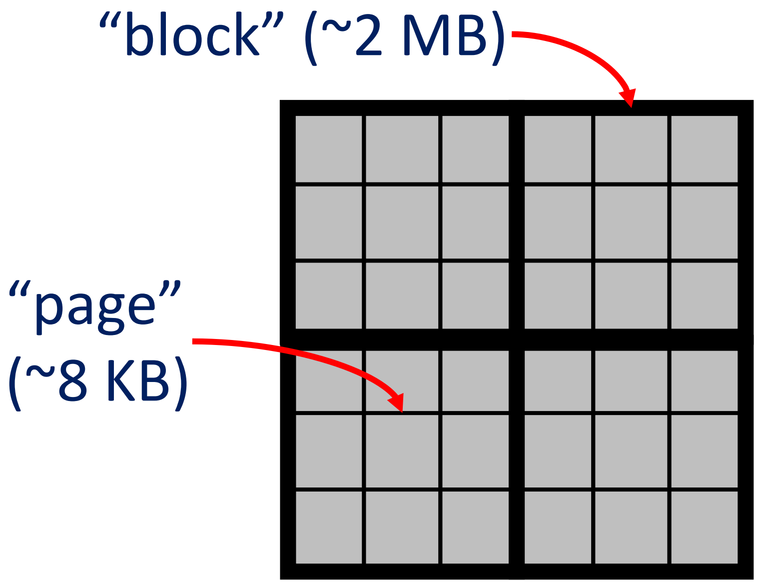


NAND-Flash Fabric Characteristics

- Read/write in “page” granularity
  - 4/8/16 KiB according to technology
  - Corresponds to disk “sector” (typically 4 KiB)
  - Read takes 10s of us to 100s of us depending on tech
  - Writes are slower, takes 100s of us depending on tech
- A third action, “erase”
  - A page can only be written to, after it is erased
  - Under the hood: erase sets all bits to 1, write can only change some to 0
  - **Problem :** Erase has very high latency, typically ms
  - **Problem :** Each cell has limited program/erase lifetime (thousands, for modern devices) – Cells become slowly less reliable

NAND-Flash Fabric Characteristics

- Performance impact of high-latency erase mitigated using large erase units (“blocks”)
  - Hundreds of pages erased at once
- What these mean: in-place updates are no longer feasible
  - In-place write requires whole block to be re-written
  - Hot pages will wear out very quickly
    - One reason SSDs not recommended for swap space!
- People would not use flash if it required too much special handling

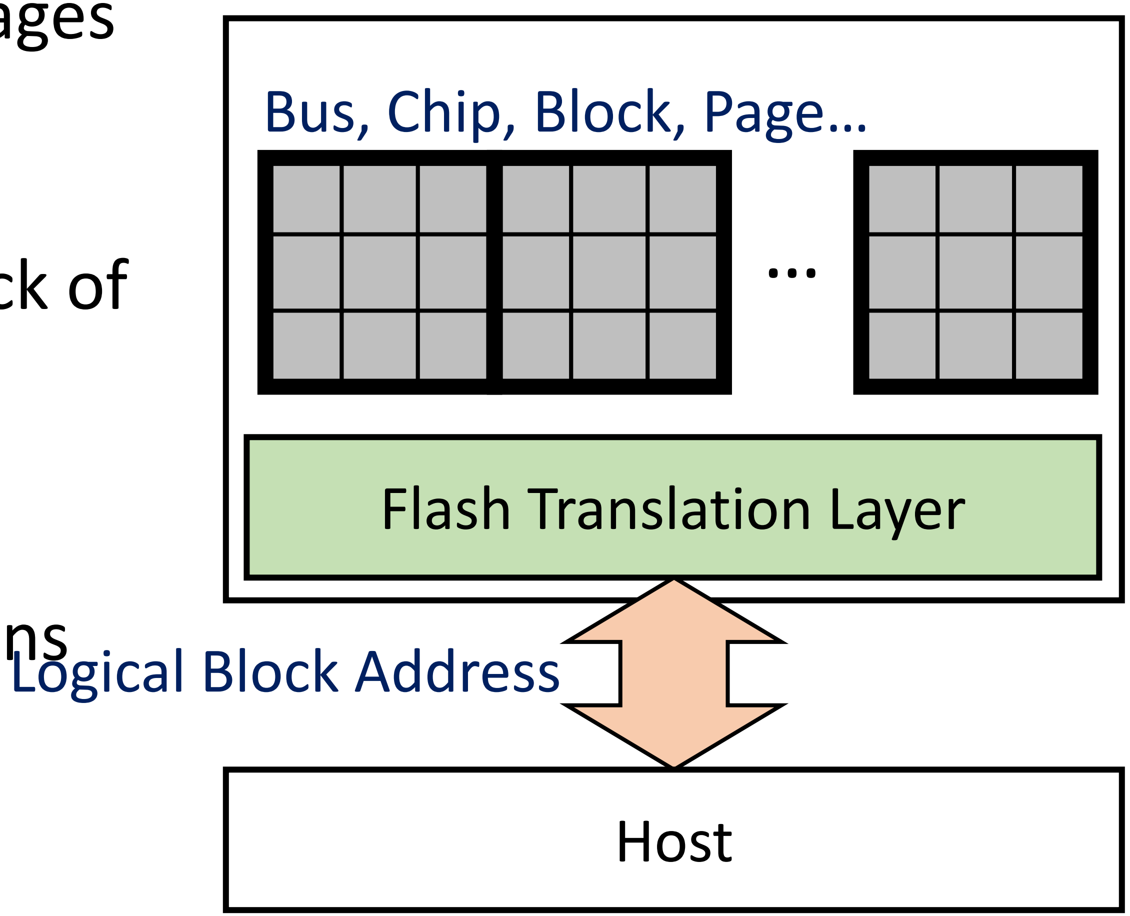


NAND-Flash SSD Architecture

- High bandwidth achieved by organizing many flash chips into many buses
  - Enough chips on a bus to saturate bus bandwidth
  - More busses to get more bandwidth
- Many dimensions of addressing
  - Bus, chip, block, page
- Write/erase needs to be intelligent to get performance/lifetime

The Solution: Flash Translation Layer (FTL)

- Exposes a logical, linear address of pages to the host
  - Drop-in replacement for disks
- A "Flash Translation Layer" keeps track of actual physical locations of pages and performs translation
  - Physicalpage = map[logicalpage];
- Transparently performs many functions for performance/durability



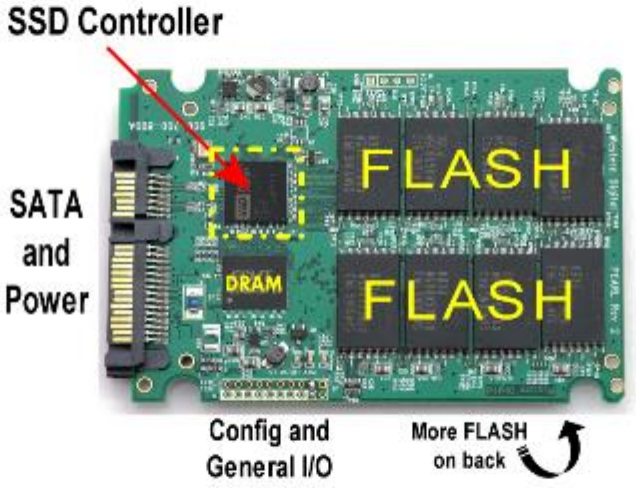
Some Jobs of the Flash Translation Layer

- Logical-to-physical mapping
- Bad block management
- Wear leveling: Assign writes to pages that have less wear
- Error correction: Each page physically has a few more bits for error codes
  - Reed-Solomon, BCH, LDPC, …
- Deduplication: Logically map pages with same data to same physical page
- Garbage collection: Clear stale data and compact pages to fewer blocks
- Write-ahead logging: Improve burst write performance
- Caching, prefetching,…

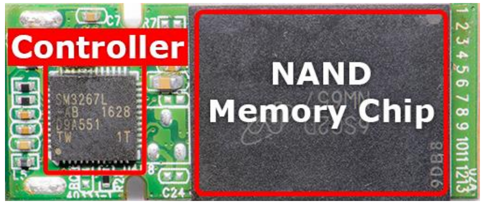
That's a Lot of Work for an Embedded System!

- ❑ Needs to maintain multi-GB/s bandwidth
- ❑ Typical desktop SSDs have multicore ARM processors and gigabytes of memory to run the FTL
  - o FTLs on smaller devices have sacrifice various functionality

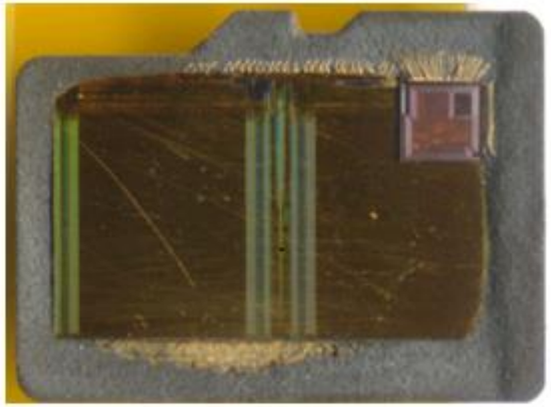
SATA SSD



USB Thumbdrive



MicroSD



Thomas Rent, "SSD Controller," storagereview.com
Jeremy, "How Flash Drives Fail," recovermyflashdrive.com
Andrew Huang, "On Hacking MicroSD Cards," bunniestudios.com

Some FTL Variations

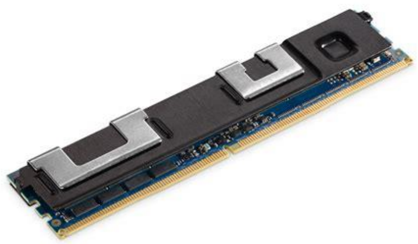
- Page level mapping vs. Block level mapping
  - 1 TB SSD with 8 KB blocks need 1 GB mapping table
  - But much better performance/lifetime with finer mapping
- Wear leveling granularity
  - Honest priority queue is too much overhead
  - Many shortcuts, including group based, hot-cold, etc
- FPGA/ASIC acceleration
- Open-channel SSD – No FTL
  - Leaves it to the host to make intelligent, high-level decisions
  - Incurs host machine overhead

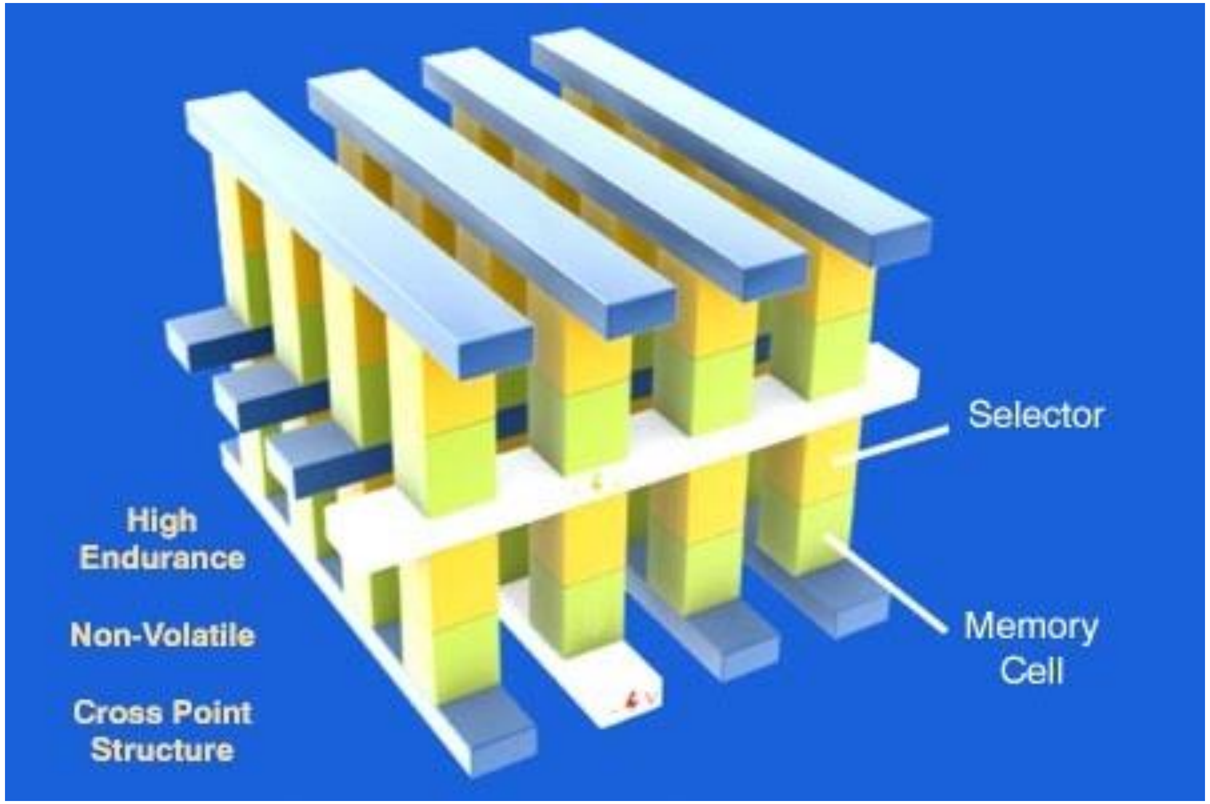
Managing Write Performance

- Write speed is slower than reads, especially if page needs to be erased
- Many techniques to mitigate write overhead
  - Write-ahead log on DRAM
  - Pre-erased pool of pages
  - For MLC/TLC/QLC, use some pages in “SLC mode” for faster write-ahead log – Need to be copied back later

Aside: Intel 3D XPoint

- ❑ Phase Change Memory? (PCM)
- ❑ Byte addressable*
- ❑ No explicit erase required
- ❑ Lower latency
- ❑ Expensive!
- ❑ Available as storage & memory





CS250B: Modern Computer Systems

Efficient Use of High Performance Storage

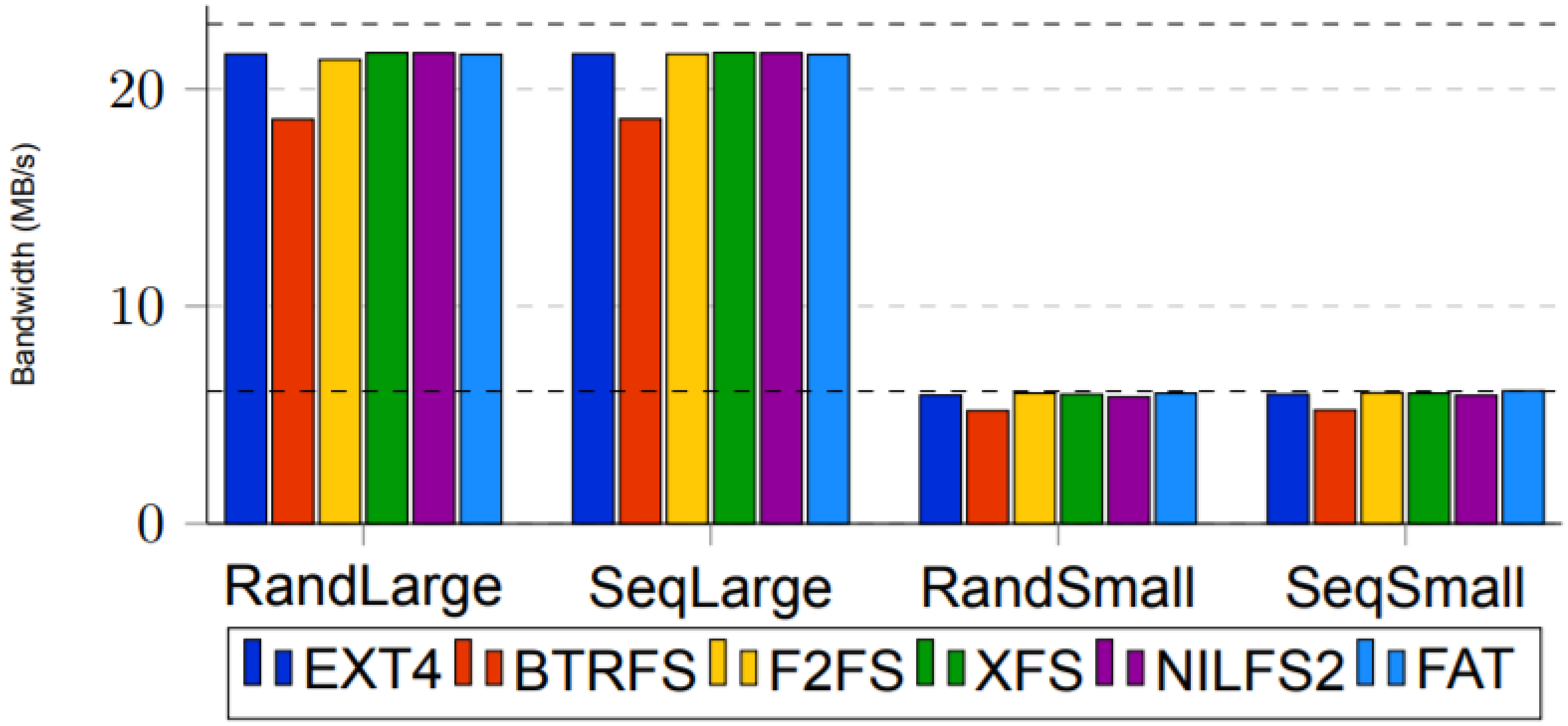
Sang-Woo Jun

UCI

Flash-Optimized File Systems

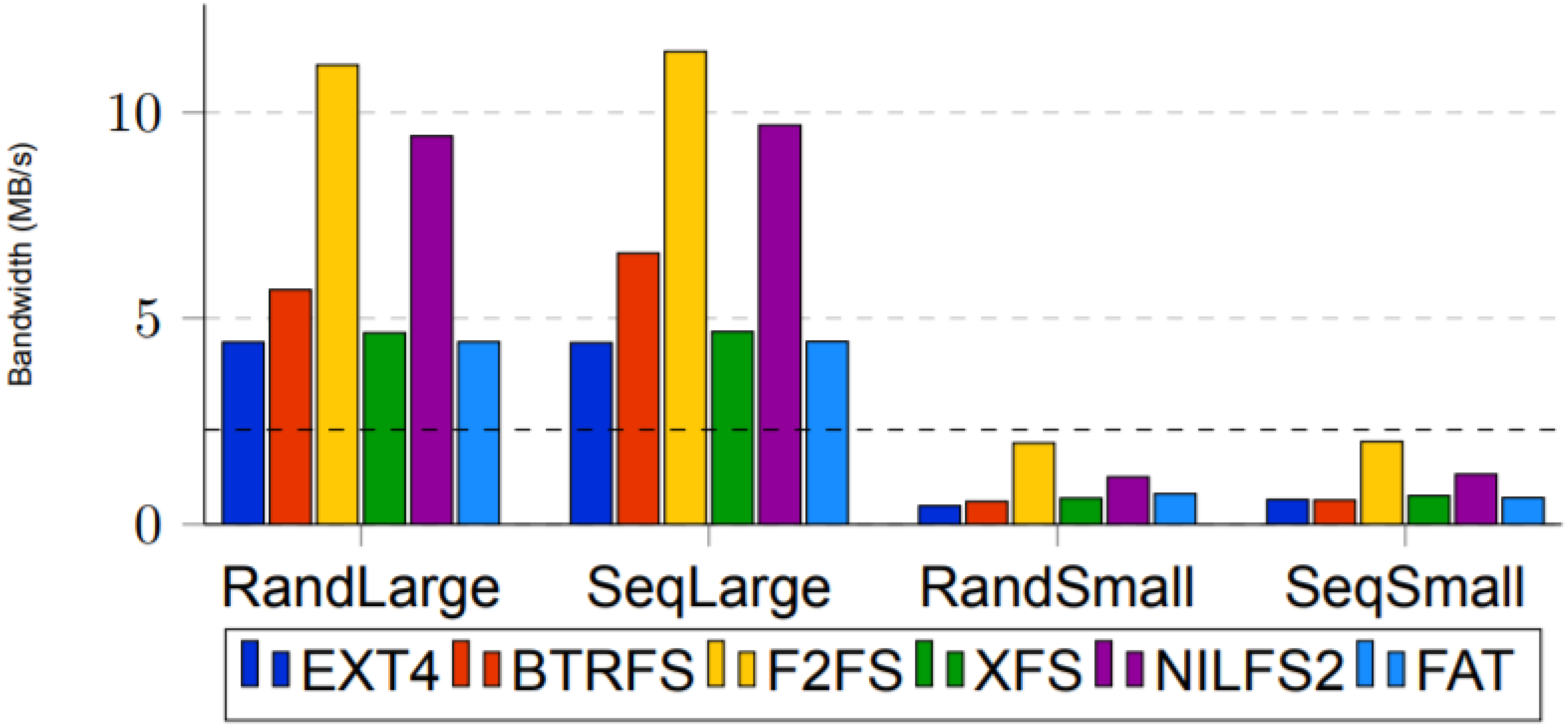
- Try to organize I/O to make it more efficient for flash storage (and FTL)
- Typically “Log-Structured” File Systems
  - Random writes are first written to a circular log, then written in large units
  - Often multiple logs for hot/cold data
  - Reading from log would have been very bad for disk (gather scattered data)
- JFFS , YAFFS, F2FS, NILFS, …

Direct Read Performance Comparisons



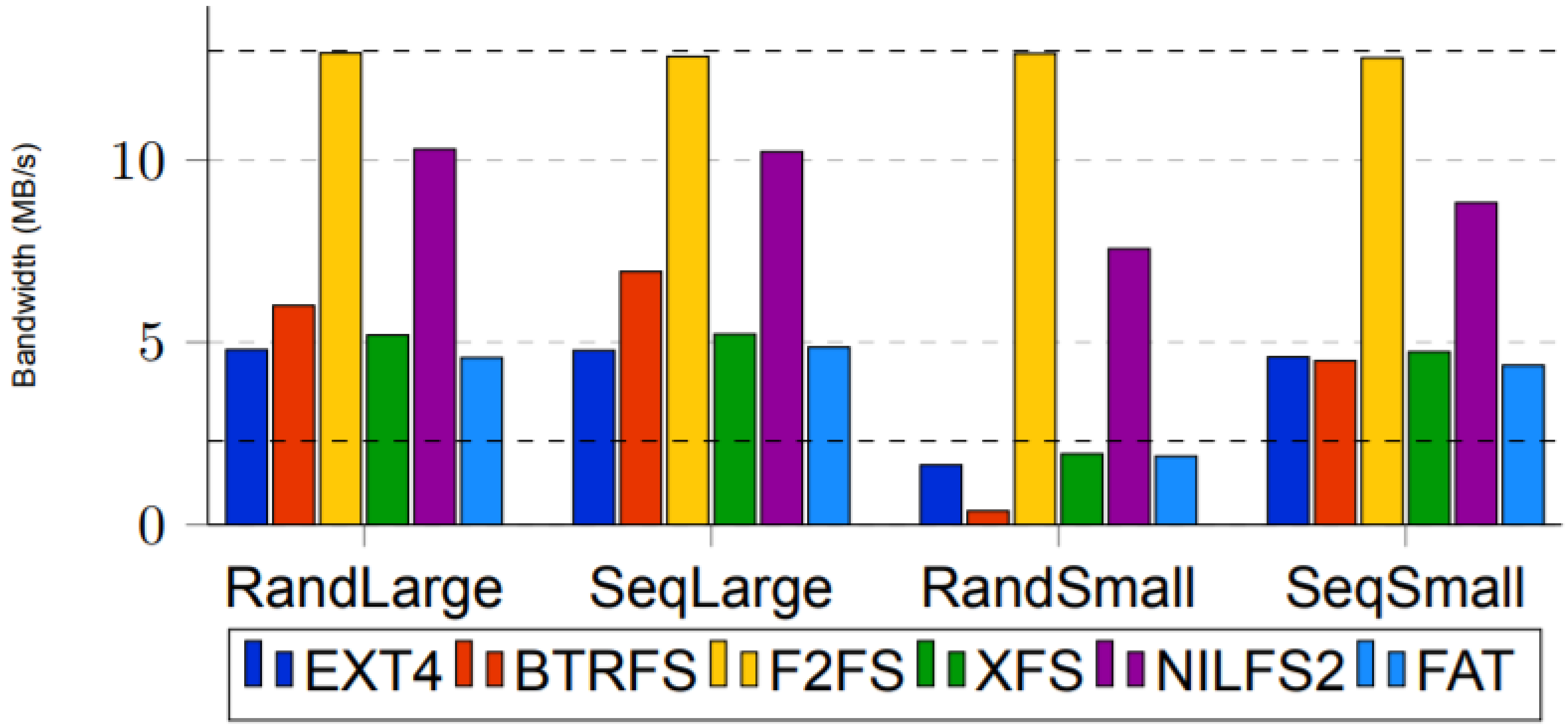
Tristan Lelong, “Filesystem considerations for embedded devices,” ELC 2015

Direct Write Performance Comparisons



Tristan Lelong, “Filesystem considerations for embedded devices,” ELC 2015

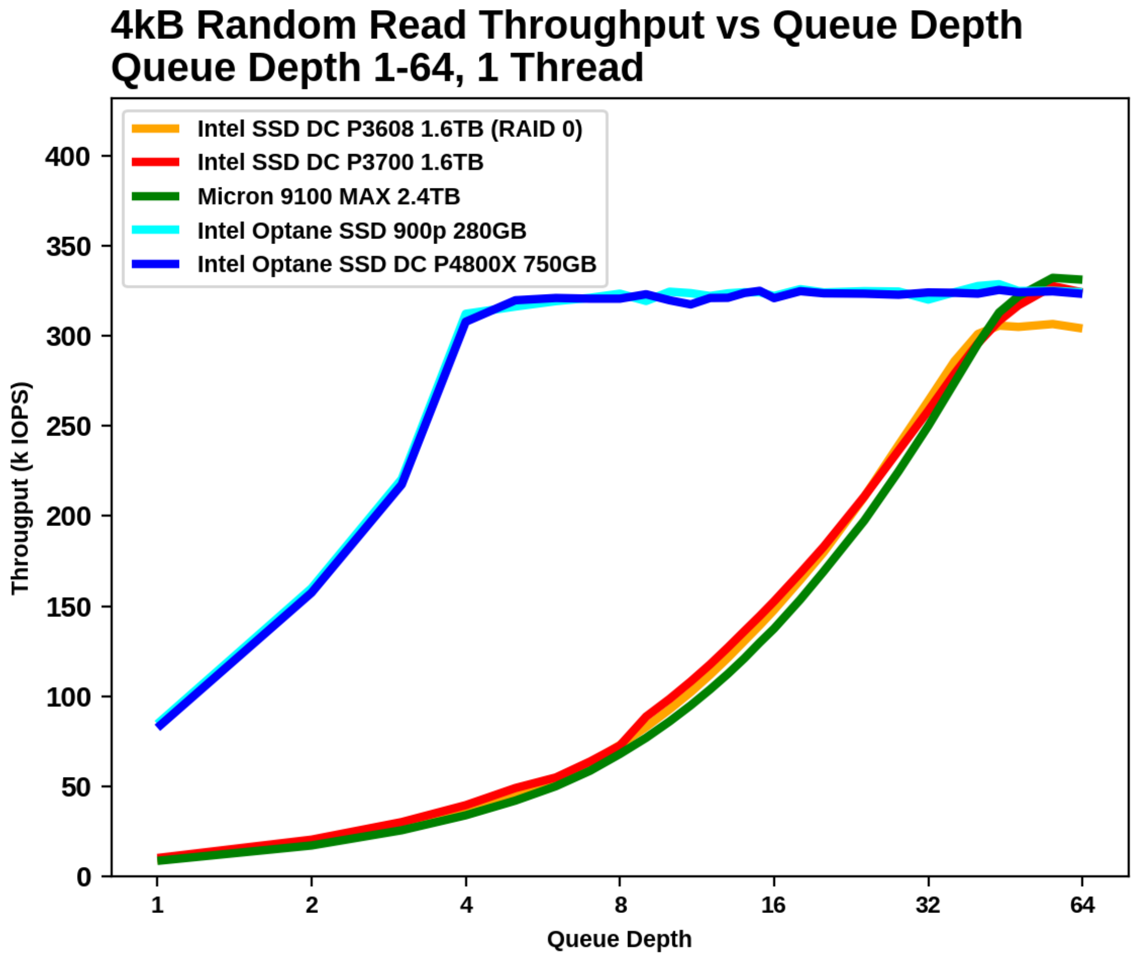
Buffered Write Performance Comparisons



Tristan Lelong, “Filesystem considerations for embedded devices,” ELC 2015

Queue Depth and Performance

- For high bandwidth, enough requests must be in flight to keep many chips busy
  - With fread/read/mmap, need to spawn many threads to have concurrent requests
  - Traditionally with thread pool that makes synchronous requests (POSIX AIO library and many others)



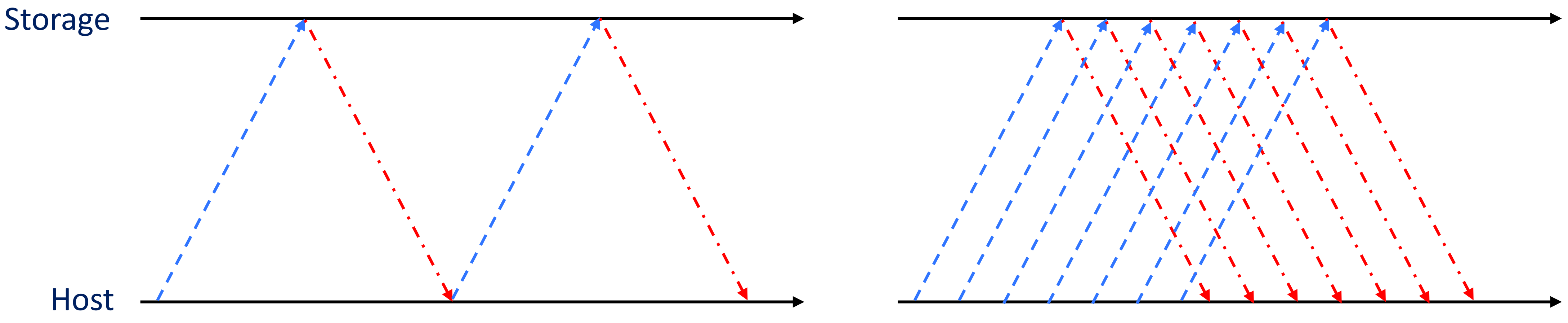
Billy Tallis, "Intel Optane SSD DC P4800X 750GB Hands-On Review," AnandTech 2017

Some Background – Page Cache

- Linux keeps a page cache in the kernel that stores some pages previously read from storage
  - Automatically tries to expand into unused memory space
  - Page cache hit results in high performance
  - Data reads involve multiple copies (Device → Kernel → User)
  - Tip: Write “3” to /proc/sys/vm/drop_caches to flush all caches
- Page cache can be bypassed via “direct mode”
  - “open” syscall with O_DIRECT
  - Lower operating system overhead, but no benefit of page cache hits
  - Useful if application performs own caching, or knows there is zero reuse

Asynchronous I/O

- Many in-flight requests created via non-blocking requests
  - Generate a lot of I/O requests from a single thread



Synchronous

Asynchronous with queue depth >= 7

Asynchronous I/O

- Option 1: POSIX AIO library
  - Creates thread pool to offload blocking I/O operations – Queue depth limited by thread count
  - Part of libc, so easily portable
  - Can work with page caches
- Option 2: Linux kernel AIO library (libaio)
  - Asynchrony management offloaded to kernel (not dependent on thread pool)
  - Despite efforts, does not support page cache yet (Only O_DIRECT)
  - Especially good for applications that manage own cache (e.g., DBMSs)
- Option 3: Linux kernel Uring
  - Relatively new! Supports non O_DIRECT

Linux Kernel libaio

- Basic flow
  - aio_context_t created via io_setup
  - struct iocb created for each io request, and submitted via io_submit
  - Check for completion using io_getevents
- Multiple aio_context_t may be created for multiple queues
  - Best performance achieved by multiple contexts across threads, each with large nr_events
  - Multi thread not because of aio overhead, but actual data processing overhead

```
int io_setup(unsigned nr_events, aio_context_t *ctx_idp);
int io_submit(aio_context_t ctx_id, long nr, struct iocb **iocbpp);
int io_getevents(aio_context_t ctx_id, long min_nr, long nr,
                 struct io_event *events, struct timespec *timeout);
```

libaio Example

- Create context

```
if( io_setup( AIO_DEPTH, &m_io_ctx ) != 0 ) {
>   fprintf(stderr, "%s %d io_setup error\n", __FILE__, __LINE__);
}
```

- Send request
  - Arguments to recognize results

```
io_prep_pwrite(&ma_iocb[idx], fd, block.buffer, bytes, offset);
IocbArgs* args = &ma_request_args[idx];
...
ma_iocb[idx].data = args;
struct iocb* iocbs = &ma_iocb[idx];
int ret_count = io_submit(m_io_ctx, 1, &iocbs);
```

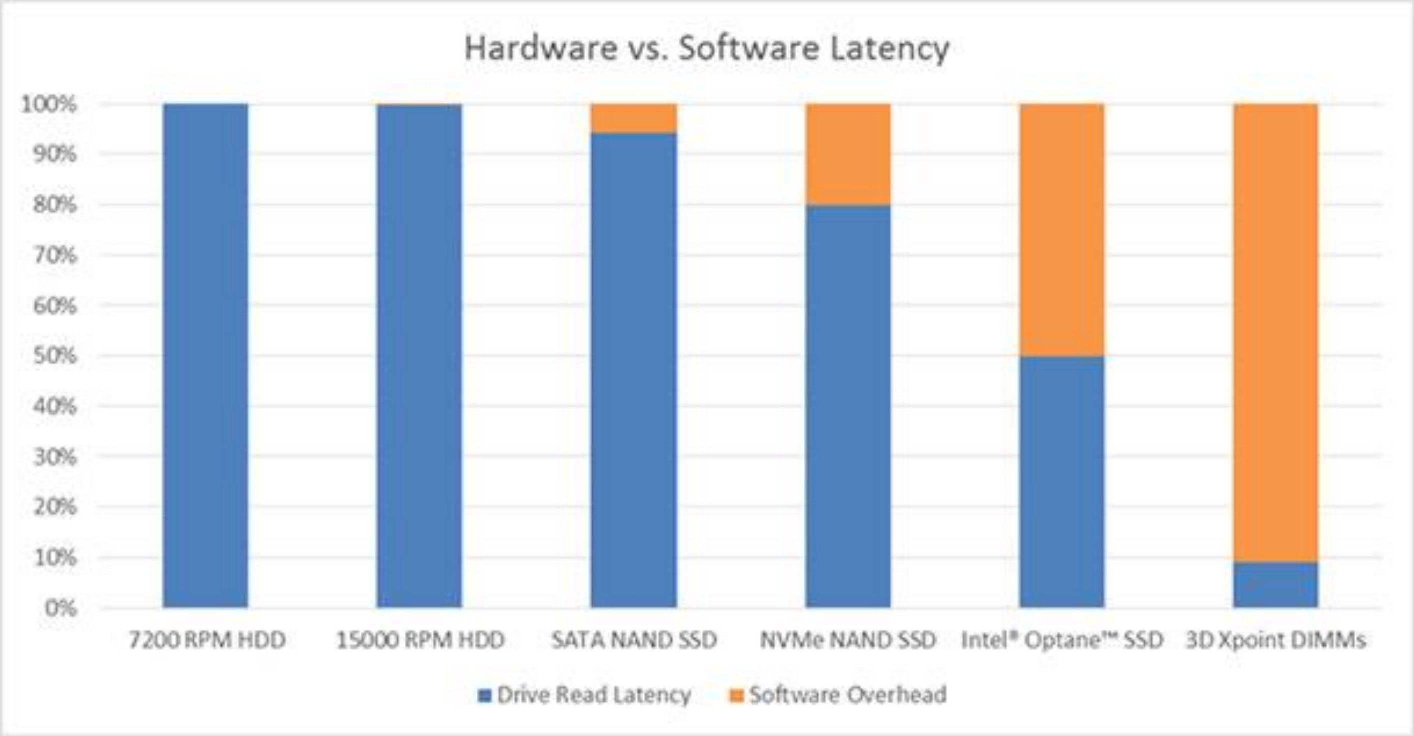
- Poll results
  - Recognize results with arguments

```
int num_events = io_getevents(m_io_ctx, 0, AIO_DEPTH, ma_events, NULL);
for ( int i = 0; i < num_events; i++ ) {
>   struct io_event event = ma_events[i];
>   IocbArgs* arg = (IocbArgs*)event.data;
```

Even with 8 KB random access, single thread can saturate multi-GB/s NVMe!

User-Space I/O Libraries

- Syscall and kernel-user data copying has become relatively expensive
- e.g., Intel Storage Performance Development Kit (SPDK)
  - User-space, lock-free, interrupt-free (polling)



Some Data Structures for Storage

- Wide class of algorithms and data structures optimized for storage
  - “External” or “out-of-core” algorithms and data structures
  - Forces coarse granularity (Multi-KBs – MBs)
  - Prioritized sequential accesses
- Most of what we learned about cache-oblivious data structures also work here

B-Tree

- Generalization of a binary search tree, where each node can have more than two children
  - Typically enough children for each node to fill a file system page (Data loaded from storage is not wasted)
  - If page size is known, very effective data structure
    - Remember the performance comparison with van Emde Boas tree

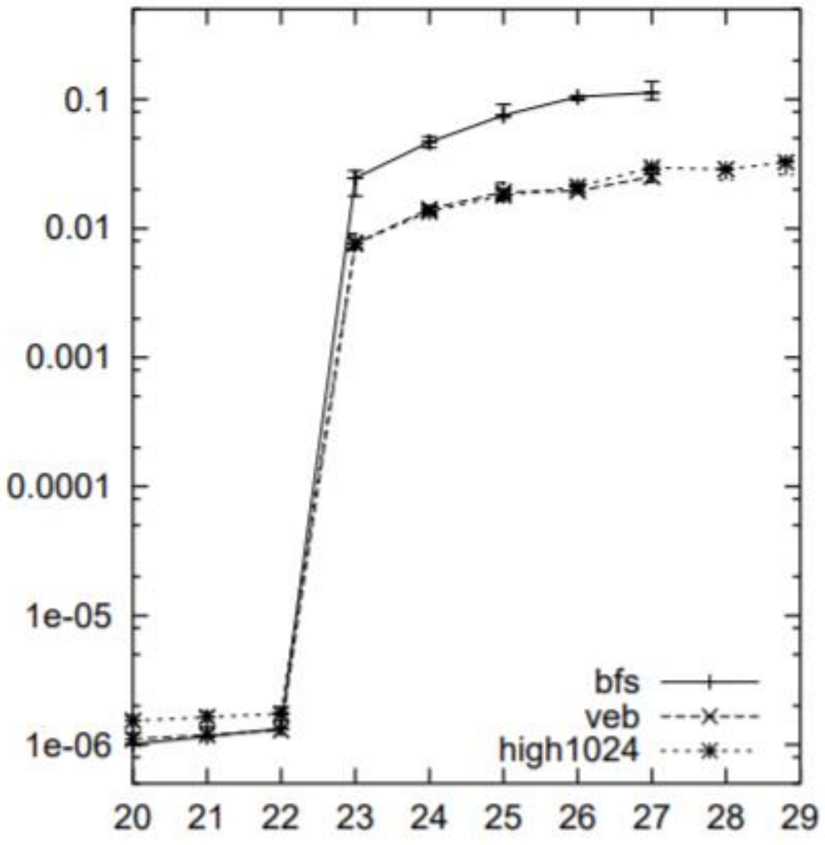


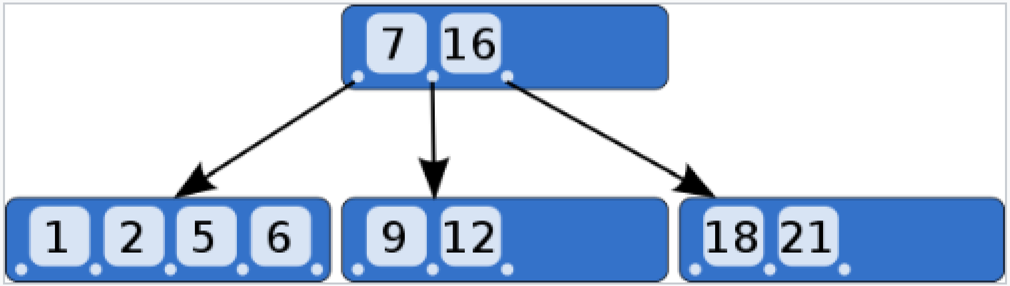
Figure 8: Beyond main memory

Brodal et.al., "Cache Oblivious Search Trees via Binary Trees of Small Height," SODA 02

B-Tree – Quick Recap

- Self-balancing structure!
- Insertion is always done at a leaf
  - If the leaf is full, it is split
  - If leaf splitting results in a parent overflow, split parent, repeat upwards
  - If root overflows, create a new root, and split old root
- Tree height always increases from the root, balancing the tree
- Deletion requires some handling for balance
  - Rotations in case of node underflow

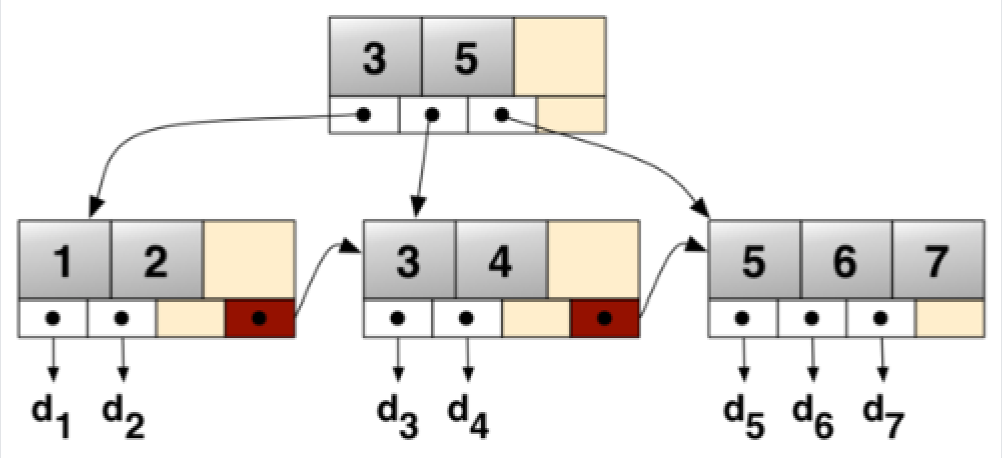
Image from wikipedia



A B-tree (Bayer & McCreight 1972) of order 5 (Knuth 1998).

B+Tree

- B-Tree modified to efficiently deal with key-value pairs
- Two separate types of nodes: internal and leaf
  - B-Tree had elements in both intermediate nodes and leaves
  - Internal nodes only contain keys for keeping track of children
  - Values are only stored in leaf nodes
  - All leaves are also connected in a linked list, for efficient range querying-



A simple B+ tree example linking the keys 1–7 to data values d_1-d_7. The linked list (red) allows rapid in-order traversal. This particular tree's branching factor is b=4.

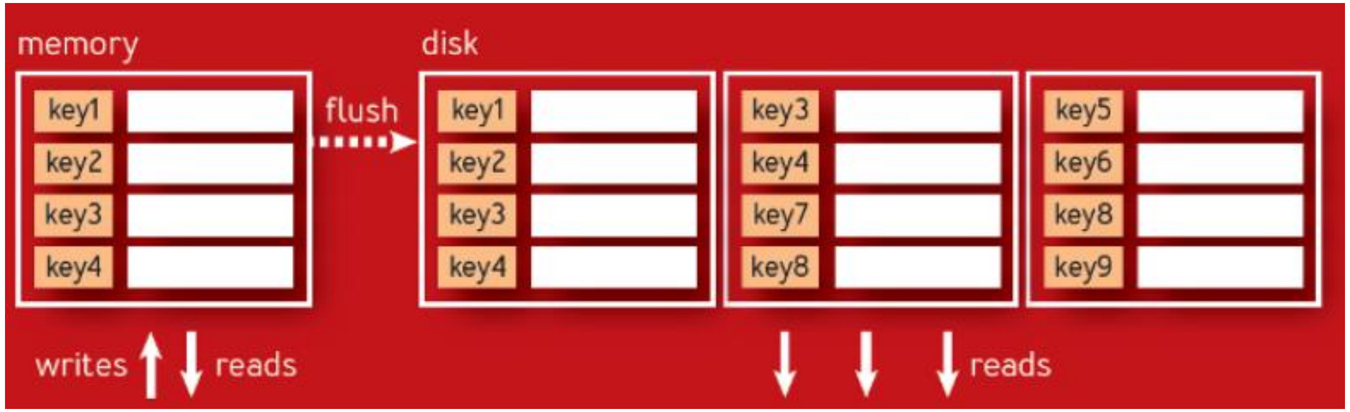
Log-Structured Merge (LSM) Tree

- Storage-optimized tree structure
  - Key component of many modern DBMSs (RocksDB,Bigtable,Cassandra, …)
- Consists of mutable in-memory data structure, and multiple immutable external (in-storage) data structures
  - Updates applied to in-memory data structure
  - In-memory data structure regularly flushed to new instance in storage
  - Lookups must search the in-memory structure, and potentially all instances in storage if not

Log-Structured Merge (LSM) Tree

- In-memory: mutable, search-optimized data structure like B-Tree
  - After it reaches a certain size (or some time limit reached), flushed to storage and starts new
- External component: many immutable trees
  - Typically search optimized external structure like Sorted String Tables
  - New one created every time memory flushes
  - Updates are determined by timestamp, deletions by placeholder markers
  - Search from newest file to old

Like clustered indices

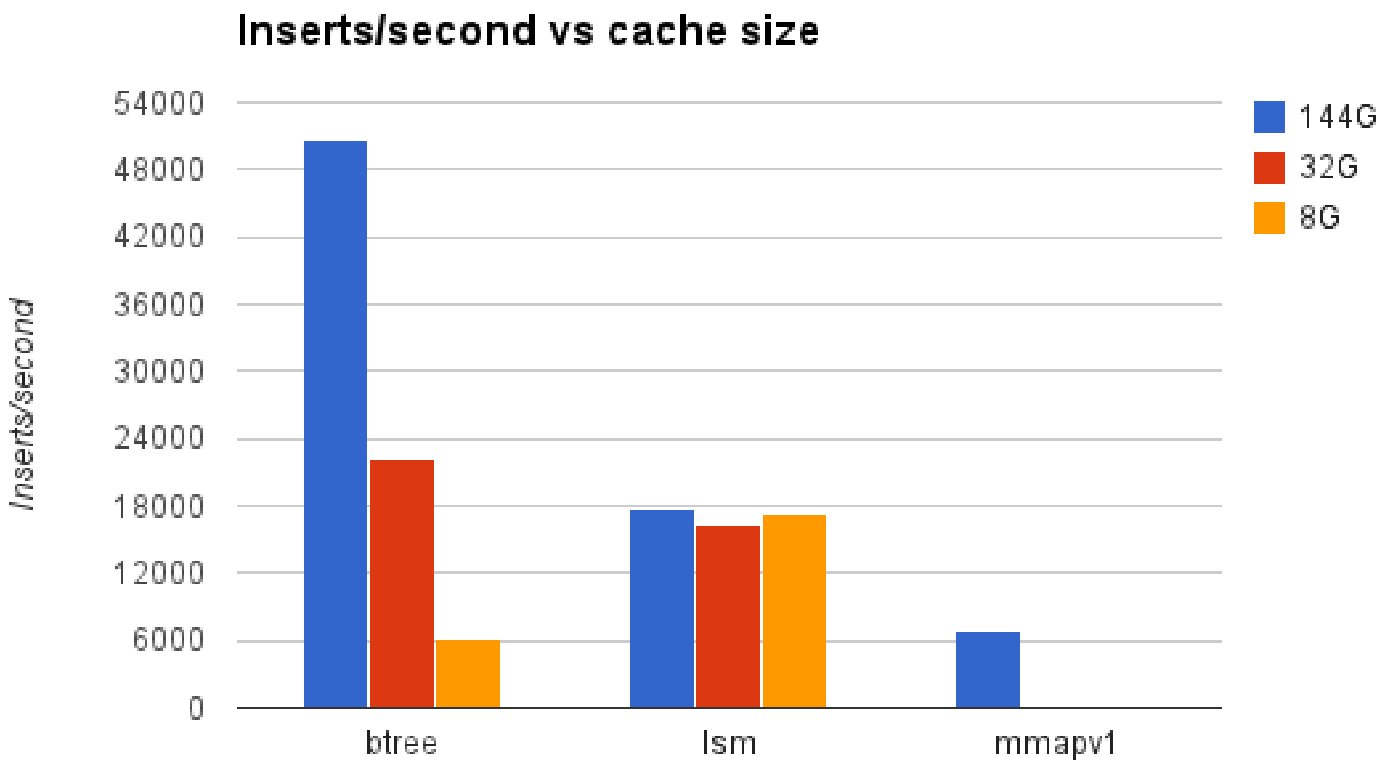


Alex Petrov, "Algorithms Behind Modern Storage Systems," ACM Queue, 2018

Log-Structured Merge (LSM) Tree

- Because external structures are immutable and only increase, periodic compaction is required
  - Overhead!
  - Since efficient external data structures are sorted, typically simple merge-sort is efficient
  - Key collisions are handled by only keeping new data

Some Performance Numbers



Data from iibench for MongoDB

Small Datum, “Read-modify-write optimized,” 2014 (http://smalldatum.blogspot.com/2014/12/read-modify-write-optimized.html)