Thinking out of the (SSD) box
How to evolve?
First there were HDDs

• Large capacity
• Cheap
• Slooooonóow
• Read/write in-place
Then there was flash

- Basic SSDs have ruled the earth for a decade
- Fast random access
- More expensive than HDDs
- Had to adapt
  - use same old interfaces as HDDs
SSD adapted

• Simple API:
  – Read/write(start sector, # of sectors)
  – SATA, SAS
• SSDs had to fit into the legacy storage stack
• Consequently, SSDs *emulated* an HDD to the host
• Similar drivers, generally same simple API...
• Q: So where’s the problem?
Imagine a future with driverless cars only

But we can't have it all now, there are still some annoying humans on the road...
SSDs emulating HDDs

• **Cons:**
  
  — Legacy software with irrelevant assumptions
    • Really slow random accesses, no endurance limit, write in-place support...
  
  — Legacy hardware with irrelevant assumptions
    • Disk can handle one concurrent random request, throughput is limited (400 MB/s)
How to evolve?

- We covered many problems stemming from using thin interfaces with SSDs
- Creates information gap between application and firmware
- Also covered some evolutionary steps to overcome the gap
  - Q: 1st step?
Discards

• Using less capacity is expensive (equivalent to buying a larger SSD)

• Can also improve over-provisioning dynamically
  – Users leave free space, delete files
  – File systems rarely utilize ALL logical addresses

• TRIM/DISCARD/UNMAP(logical addresses)
  – Nullify mapping entry for logical addresses
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>OS Logical View</strong></td>
<td>File A</td>
<td>File B</td>
<td>File A</td>
</tr>
<tr>
<td></td>
<td>SSD writes new data; only SSD knows about OP</td>
<td>Only OS knows location C1 &amp; C2 are no longer valid and SSD keeps rewriting it during GC</td>
<td>OS writes new file to old location; SSD marks old location ready for GC and file E gets written elsewhere</td>
</tr>
</tbody>
</table>

http://www.thessdreview.com
• Legacy interfaces augmented with relevant SSD-specific command support (circa 2008)
• Discard supported by all major storage interfaces, SSDs, OSs and file systems
Other special APIs?

• DISCARD Born out of SSD’s indirection gap
What about copying?

- Applications often move data on the same disk to different locations
  - In HDDs logical=physical
  - We must physically copy to other logical addresses
- Q: can we do better in SSDs?
• By default we can move
  – Copy to new logical addresses
    → copy to new flash pages
• Result: Same same (data) but different (PHYSICAL locations)
• Why not unite them?
Can update indirect mapping
• **Q: what can we use this for?**
  
  – **Hint:** at least 3 uses we already talked about during the course
Fragmentation

• Most file systems attempt to store file data in contiguous groups of disk sectors

• Over time files are modified / created / deleted / moved...

→ blocks unavoidably scattered on disk

Defragmentation

• On runtime we can try improve block allocation
  – Try to allocate same-file data contiguously
  – Try to cache (to increase chances of contiguous allocations)

• On idle times
  – Proactively find file data scattered across disk
  – Gather data in free space

• Q: what about SSDs?
HDD vs. SSD grep

- Cool benchmark: grep on Linux kernel git
  - 10,000 pulls from the Linux git repository
  - Every 100 pulls recursively grep and measure run time
- Conclusion: fragmentation on HDD is much worse...but affects SSD as well!
  - F2Fs is worst!

File Systems Fated for Senescence? Nonsense, Says Science!, FAST’17
Fragmentation & app launch time

(a) Six applications on N6.  
(b) Twitter on five smartphones.

Fig. 4: Changes in app launching times.

Improving File System Performance of Mobile Storage Systems
Using a Decoupled Defragmenter, ATC’17
Defragging smartphones

- Defrag moves a LOT of data around
  - Can shorten lifetime significantly

Fig. 5: The amount of data copies by file defragmentation with different defragmentation periods.

- The data Partition Size
  - N5: 26.7 GB
  - N6: 26 GB
  - S6: 25 GB
• Remapping to the rescue 😊
  – Copyless copy of defragged data
• Q: one minor problem?

![Diagram of file system performance improvement](image)

Fig. 10: An example of defragmentation in janusd.

Improving File System Performance of Mobile Storage Systems
Using a Decoupled Defragmenter, ATC’17
• Spare area stores logical address
• Remap creates inconsistency between L2P and P2L mapping information

• Q: Solution?
• Spare area stores logical address
• Remap creates inconsistency between L2P and P2L mapping information

• **Q: Solution?**
  – Special log of large remap operations
  – How to avoid write amplification?
Deduplication

• **Compression**
  – What is the most succinct representation of this data?

• **Deduplication**
  – Hasn’t this data block/chunk appeared before?
  – Similar to compression, but at entire *dataset* level

![Original Data](image1)

![Deduplicated Data](image2)
Dedup basics

• Fingerprint each 4KB block using a hash function
  – Sha1, Sha256, others...

• Store index of all hashes in system

• New block:
  – Compute hash
  – Look up hash up in index table
  – New hash → add to index
  – Known hash → store as pointer to existing data (!)
Dedup and SSDs

• **Cons:**
  – Cant dedup files, only blocks
  – Already limited memory resources (more tables...)
  – Dedup gains as dataset is bigger, but SSD is (relatively) small
  – Limited hardware (hashing computational latency)
Deduplication and remap

• Mapping info still resides on SSD
  – Including for duplicates
  – Persistency still the firmware’s job
• Can move other functionalities to host 😊
  – Hashing blocks
  – Remap duplicated blocks at device levels
• Don’t need to use weak SSD CPU & hardware for hashing
  – Only to call remap() when identical fingerprint detected
The Crash Consistency Problem

• Disk guarantees that sector writes are atomic
  – No way to make multi-sector writes atomic

• How to ensure consistency after a crash?
  1. Don’t bother
     • Accept that the file system may be inconsistent after a crash
     • Checkfile system during bootup (SLOW)
     • File system checker (fsck)
  2. Use log to make multi-writes atomic
     • Log stores a history of all writes to the disk
     • After a crash the log can be “replayed” to finish updates
     • Journaling file system
Ext4 Journaling

• Make writes **transactional** by using a write-ahead log
  – One logical execution unit, *either performed entirely or not at all* (overly simplistic...)
  – Commonly referred to as a *journal*

• Ext3/4 and NTFS (windows) use journaling
• Key idea: writes to disk first written to journal
  – Reminiscent of the “log” concept
  – After journal is written, writes execute normally and copied to final place in disk
Remapping journal

- **Logical** view of journaling

JFTL: A Flash Translation Layer Based on a Journal Remapping for Flash Memory, ToS'09
• **Physical** view of journaling
• With remapping
• Q: why didn't remap catch on?
  – Defrag?
  – Dedup?
  – Journaling?
Transactional flash

• Alternative solution to remapping journals
• Journal use transactions
  – series of operations that perform a particular function
  – must succeed or fail as a complete unit

• Example: File Creation (Transaction)
  – requires multiple writes (operations)
  – update the metadata of the parent directory and new file.
Transactional flash

- SSD exports a transactional interface
- Transaction need ACID properties
- **Atomicity**: all or none of the write operations in a transaction are executed
- **Consistency**: In case of failure, the System should remain in its last consistent state
- **Isolation**: no conflicting writes from concurrent transactions (to same pages)
- **Durability**: after commit, data written by transaction is made durable on SSD

Transactional Flash, OSDI’08
Data Journaling Example

• File systems use journaling and copy-on-write
• File append example
  – Three writes: inode v2, data bitmap v2, data $D_2$
• Before executing these writes, first log them

1. Begin a new transaction with a unique ID=$k$
2. Write the updated meta-data block(s)
3. Write the file data block(s)
4. Write an end-of-transaction with ID=$k$
• SSDs are already “copy on write”, i.e. log-structured
• Highly parallelized
• Fast random access
• Spare area for each page can store metadata (also regarding transaction)
- **WriteAtomic**\((p_1...p_n)\)
  - Transaction on multiple pages
  - Returns Tx ID
  - Multiple function commands gathered into one
- **Abort**\((Tx \text{ ID})\)
  - Abort ongoing transaction
- Logical addresses remapped to new locations when a transaction commits
Traditional commit protocol

- Dumb mimicry of file system journal logic
- Page has Tx metadata in spare area
- AFTER “data intention” pages (p1...pn) write completed, write commit record
  \( \rightarrow \) extra latency per write transaction
- Commit record erased only after previous “data intention” records are made obsolete (by new transactions)
  \( \rightarrow \) extra space occupied

- More efficient protocols possible (see paper)
Q: Why didn't transactional SSDs take off?
Nameless writes (de-indirection)

<table>
<thead>
<tr>
<th>Traditional Block Device</th>
<th>Nameless “Block Device”</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ</td>
<td>READ</td>
</tr>
<tr>
<td>WRITE</td>
<td>ALLOCATE-AND-WRITE</td>
</tr>
<tr>
<td>(TRIM)</td>
<td>TRIM</td>
</tr>
</tbody>
</table>

- Regular write specify data & logical address
- Nameless write specifies only data
- Device free to choose physical address and return it to host!

NANDFS: A Flexible Flash File System for RAM-Constrained Systems, EMSOFT’09

De-indirection for Flash-based SSDs with Nameless Writes, FAST’12
• Upside: page-mapping not strictly necessary
  – Writes incur no mapping changes
• For reads even block-level/hybrid mapping is enough
Implications

• TRIM more essential now
  – Why?
• Garbage collection performed in-place
  – Why?
• Wear levelling requires host callbacks
  – Why?
• Some data must be allocated traditionally (host-allocated addresses)
  – Why?

• Hasn’t caught on
The case for programmable SSDs

- Data ideally processed as close to storage as possible
- Reduced data movement across storage/network/memory/CPU for compute
  - Avoid expensive transfer overheads
  - Maximize local processing for ideal parallelism

PCIe gen3 (4 lanes)
Hardware b/w = 4 GB/s
Achievable b/w = 2-2.5 GB/s

(PCIe) Flash SSD
32-channel, 1 package/channel
32 x 50 MB/s = 1.6 GB/s
4-plane packages?
4 packages/channel?
Use case: in-situ processing

• Big data processing is here
  – Larger databases, high-performance computing, genetic data processing
• Huge datasets, PetaBytes+ of data
• Processing centrally is SLOW
  – Must transfer data piece by piece to central processing server
  – Serializing processing
  – Large latencies just for transferring data to where it’s processed
Modern big data processing is distributed and parallelized across multiple machines
— E.g, Hadoop
Case Study: OCP storage server (1/2)

PCIe gen3 (16 lanes)
Hardware b/w = 16 GB/s

16 SSDs x 1.6 GB/s = 40 GB/s
4-plane packages? 4 packages/channel?

Microsoft research, Storage Developer Conference 2017
Programmable storage

• SSDs are getting faster
  – More flash $\rightarrow$ more IOPs and throughput
• SSD interfaces too (NVMe)
• But regardless of improvement the problem remains:
  – Still want to avoid redundant transfer times
  – Including from SSD$\rightarrow$host!
• Possible solution: perform (some) data processing on the SSD itself!
Leading examples

• Currently more for research purposes
  – Samsung Smart SSD
  – DragonFire Card by DellEMC and NXP

• Key idea: user defined programs can execute within the SSD

SSD In-Storage Computing for List Intersection
DaMoN’16

Figure 1: Smart SSD architecture
• **Pros:**
  – SSDs already perform complex tasks with non-trivial HW
  – Save transfer time to host
  – Save host CPU time
• **Cons?**
• **Pros:**
  – SSDs already perform complex tasks with non-trivial HW
  – Save transfer time to host
  – Save host CPU time
• **Cons:**
  – SSD CPU is low-clocked
    • e.g. 400 MHz vs. 3.4GHz of host
  – SSD CPU is less sophisticated
    • Smaller L1/L2 (or none)
    • Less pipelining (or none)
  – SSD RAM is slow and expensive
    • E.g. 18ns 4-byte read vs. 4ns on host DRAM
NVMe

- A quantum leap in SSD evolution
- Industry initiative to standardize PCI attached flash storage (2011)
  - Dedicated interface for fast non-volatile storage devices
  - Designed with SSDs in mind
• Multiple queues
  – Separate submission/completion
  – Fits parallelized nature of SSDs, higher throughputs & IOPS rates
• Format/deallocate (discard) support (optional)
• Multiple namespaces
  – Multiple volumes
• Can be shared by multiple devices
NVMe goes out of the box

- Host Memory Buffer (HMB),
  - Optional NVMe feature

"During initialization, host software may provide a descriptor list that describes a set of host memory address ranges for exclusive use by the controller"
NVMe 1.21a, Nov 2014

- Q: What’s it good for?
DRAM-less SSD

• Entire new line of DRAM-less SSDs introduced in recent years
  – Toshiba OCZ TL100, SanDisk SSD Plus, Patriot Spark, Phison S11
• Less power, cheaper, physically smaller
• Use host memory

Phison, FMS’15

<table>
<thead>
<tr>
<th></th>
<th>DRAM-less Design</th>
<th>DRAM Design</th>
</tr>
</thead>
<tbody>
<tr>
<td>32GB (2 die)</td>
<td>Active</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>Idle</td>
<td>X</td>
</tr>
<tr>
<td>64GB (4 die)</td>
<td>Active</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>Idle</td>
<td>X</td>
</tr>
</tbody>
</table>

DRAM-less design shows power advantages.
Even more out of the box!

“There is no FTL...”

• Latest trend in SSD design
  – But also an old trend
• Idea: stop trying to bridge the information gap with partial knowledge
Out-of-box SSDs

• Goal: empower the software managing SSD hardware, and make it smarter

• So far tried making SSD hardware more capable
  – Bigger buffers
  – Hardware accelerators (compression, replication)

• Also tried making software smarter
  – Discard
  – Transaction support
  – Remap

• Encountered multiple inefficiencies and constraints 😞
• Lets just move it to the host and get it over with 😊
• Move the stuff that we really want
• Expose geometry of flash device
  – Expose a set of physical LUNs
    • Potentially access multiple addresses from set as one logical unit (e.g., dial-die pages)
  – Performance
    • Media Timings (Read/Write/Erase)
  – Media-specific metadata
    • Out of band size
  – Controller features
    • E.g., Device-side media buffering vs host-side media buffering

• Read/Write/Erase
  – Using pre-compiled library
  – Vector I/O to support parallelized commands
• Logic moved to host
  – Mapping
  – GC
  – Refresh
  – Write buffering

• No need to deal with specific NAND chip details
  – But may also restrict specific optimizations
Memory Technology Device (MTD)

- Old Linux management system for raw flash devices
- Unix traditionally knew block/char devices
- Flash is neither
- Abstraction layer for embedded devices with flash
  - /dev/mtd0
  - Provides calls to read/write/erase data
  - Driver calls are abstracted away
- Part of Linux kernel since early 2000s (!)
- Good enough for small embedded devices and applications, not much more
Second wave of trend

• In 2011 FusionIO proposed a similar notion
  – PCIe-attached SSD
  – Host-side FTL
• Had many great ideas
  – Optimized key-value store
  – Tailor-made FTLs
• And...
Bizarro world

- Fragmentation sucks 😞
  - Complex management, defragging, need optimizations...

- Let's make fragmentation impossible!

- "DFS: A File System for Virtualized Flash Storage", FAST’10

- A flash-dedicated file-system, proposed by FusionIO (now Sandisk)
• Proprietary SSD exposes special API to host
  – Read, write, trim blocks
  – So far, so good...
• Q: so what happened to mtd and FusionIO-like devices?
• Part of Linux kernel since 2016 (4.4)
• Q: will it continue to spread?

Alibaba Launches Dual-mode SSD to Optimize Hyper-scale Infrastructure Performance

March 19, 2018
A Dual-mode SSD (Solid State Drive), a storage device which supports both Open-Channel Mode and native NVMe mode, has been developed by the Alibaba Infrastructure Services team, and ai

NVMe Chipset Solutions

The Marvell® 88SS1098 and 88SS1088 are PCIe Gen3x4 SSD controllers supporting single and dual port functionality, the NVMe 1.3 standard and emerging open channel architectures. Both controllers are powered by Marvell Gen4 NANDEdge™ LDPC error correction technology, providing support for 3D NAND TLC and QLC technologies, extending SSD lifetime while maintaining
Faster interfaces

• NVMe and open-channel SSD attach to PCIe bus
• PCIe bus is
  – Popular
  – Fast
  – Long history
  – Scalable

• Q: there’s another popular fast(er) bus on the board...?
The memory bus: the future?

- PCIe lane 1 GB/s
- Memory bus has $O(10\text{GB/s})$ speed per lane
- Can attach flash to the memory bus
- Many a papers and start-ups ventured this way
- Goal: cheap and/or non-volatile memory solution

Q: how can we incorporate DRAM and flash?
Non-volatile RAM

- DRAM device backed by flash
  - Small battery/supercapacitor
  - On power down save DRAM data to flash
  - On power up reload
- DRAM size==flash size
- Really just expensive DRAM
- Sometimes attached to PCIe
Hybrid memories

• Medium-size DRAM cache
• Large flash backend
• Mapping flash into the processor’s address space
  – Expose flash-sized memory to host
• Several companies tried to ride the wave circa 2014
  – IBM, Sandisk, Diablo (Start up)
• Spoiler: all pretty much abandoned. Why?
• **Issues as memory replacement:**
  
  – Latency variance: accessing flash and DRAM variably (OS assumes memory latency is uniform)
  
  – Byte addressable (potential endurance and performance killer for flash)
  
  – OS needs persistence awareness
• Q: did/will it work? Is DRAM+flash the future?
• Q: can we utilize flash characteristics for better byte-addressability?
Next time: other branches