Tiering and Caching in Flash-Based Storage

Richard Lary
Chief Scientist
X-IO Technologies

Caching and Tiering

- Caching and Tiering are techniques for improving the performance of a hierarchy of storage devices
- Caching **copies** data residing on a slow storage device to a **temporary** residence on a faster device
- Tiering **moves** data residing on a slow storage device to a new **permanent** residence on a faster device
- Caching and tiering use different policies to drive their operations, due to their nature and history

Storage Hierarchy Members

(Ranked Fastest to Slowest)

Storage Technology	Latency	Bandwidth	8KB I/O	Ratio	Comments
SRAM	0.5-5 ns	> 100 GB/sec	80 ns	-	Byte Addressable
DRAM	30-80 ns	> 15 GB/sec	600 ns	8:1	Byte Addressable
3D Xpoint (NVMe storage device)	10 uS	> 1 GB/sec	18 uS	30:1	Byte Addressable, needs wear leveling
Flash SSD (SLC, MLC, TLC) (NVMe, SAS, SATA)	20-60 us	200-700 MB/sec	35-100 uS	2-5:1	
Magnetic Disk HDD (15K, 10K, 7200 RPM) (SAS, SATA)	3-8 ms	100 MB/sec	3-8 ms	30-200:1	
Optical / Tape	50 ms - ∞	20-300 MB/sec	50 ms - ∞	> 10:1	"I'm not dead, yet!"

Some simple equations concerning storage hierarchies

- Latency ≈ $\sum$ (P(data in L_i) * Latency(L_i)) for all hierarchy levels i
  - H_i = P(data in L_i) / (1 - $\sum$ P(data in L_j), j < i) is called the "hit rate" of level i of the hierarchy, and 1 – H_i is called the "miss rate" of level i
- Performance ≈ $\sum$ (P(data in L_i) * Performance(L_i))
  - This is "maximum theoretical performance" we're talking about here
- Overhead ≈ $\sum$ Moves($L_i \rightarrow L_j$) for all hierarchy levels i, j
  - Overhead is controller work that does not move data to/from clients
- Goal: minimize latency while keeping overhead down

A general rule for managing storage hierarchies

The smaller the performance ratio between two tiers,

The more thought you should put into deciding to move or copy data between those tiers to improve overall performance

Cache Design: Write Policy

- Read caches (aka Write-through caches) never contain the latest copy of data. All writes are sent to the slower device, and either allocated, replaced, or invalidated in the cache.
- Write-back caches accept write data into the cache and then copy (flush) it to its real residence at some later time.
  - Multiple contiguous user writes are combined into large flush writes
- Write-back caching provides significant latency and performance gains, but the cache must take pains to insure it never loses dirty data (write data that has not yet been copied to its real residence).
  - Write-back caches must be non-volatile and fault-tolerant

Cache Design: Associativity

- Associativity describes how many locations in the cache may map a particular data item in the slower device.
- Higher associativity uses the cache more efficiently but makes cache lookups more complicated.
- CPU caches have low associativity – generally 2-8
- Storage caches today are fully associative
  - Any cache location can hold any data item from slower device
  - Needs sophisticated lookup structure for performance

Cache Design: Read Ahead

- If read-ahead enabled, the cache logic looks for sequential patterns in data read accesses
  - E.g. a read miss adjacent to multiple consecutive cache entries
- If a pattern is detected, the cache logic copies adjacent data from the slow device in expectation of a future hit
  - By increasing the amount of data copied to the cache
  - Or, in a storage cache, by starting an asynchronous process that copies a large amount of unread sequential data into the cache (based on continued sequential user accesses)

Cache Design: Read-Ahead (2)

- Read-ahead is a valuable tool for turning future cache misses into cache hits, and multiple small reads into fewer large reads; but it assumes that sequentially-accessed data is generally stored contiguously on media. This is not true in a storage cache if:
  - Traditional file system data is highly fragmented
  - Log-structured file systems or disks are behind the cache

  in which case read-ahead has fewer benefits

Cache Design: Replacement Policy

- Replacement Policy selects the data to be thrown out of the cache (the “victim”), to make room for new data.
- Replacement Policy is generally the most complex cache policy in storage caches, and one that has a significant effect on cache performance. It takes into account:
  - If data is Read data, Read-ahead data, or dirty Write data
  - How recently the data was last accessed
  - How many times the data has been accessed while in the cache
  - For dirty data, how much dirty data is “near” it on the slow device
  - How much total dirty data is in the cache

Cache design: Cache metadata

- The metadata for a cache consists of all the information required to access and maintain the cache. Each storage cache entry requires the following metadata in DRAM:
  - Residence location of the data (LUN + VBA)
  - Size of the cache entry in blocks
  - Location of the data in the cache (allowing scatter/gather)
  - Pointers for optimizing searches (e.g. search tree links)
  - Pointers for Replacement Policy (LRU links)
  - Data structure for locking
  - Flags (Dirty, read-ahead status, data present, flush status, etc, etc)
- Overall, about 64-128 bytes per cache entry
  - If a cache entry represents 8KB of data, that's ~1-1.5% of the data

Cache Design: Victim Cache

- A Victim Cache is a storage hierarchy member that receives data from a faster member of the hierarchy rather than a slower member.
- A Victim Cache serves to improve the Replacement Policy of the cache “under” it, by mitigating the bad effects of ejecting the “wrong” data from the cache.
- Intel now ships a CPU Victim Cache with its Haswell architecture as an “L4 Cache”.

History of Caching

- Introduced in 1970's in CPU design
  - SRAM cache in front of DRAM (or core) main memory
    - SRAM was 10-20x the performance of main memory at the time
  - Direct-mapped, write-through caches at first
  - Write-back caches, limited associativity shipping by 1975
  - Victim Caches introduced with DEC Alpha CPU in the 1990's

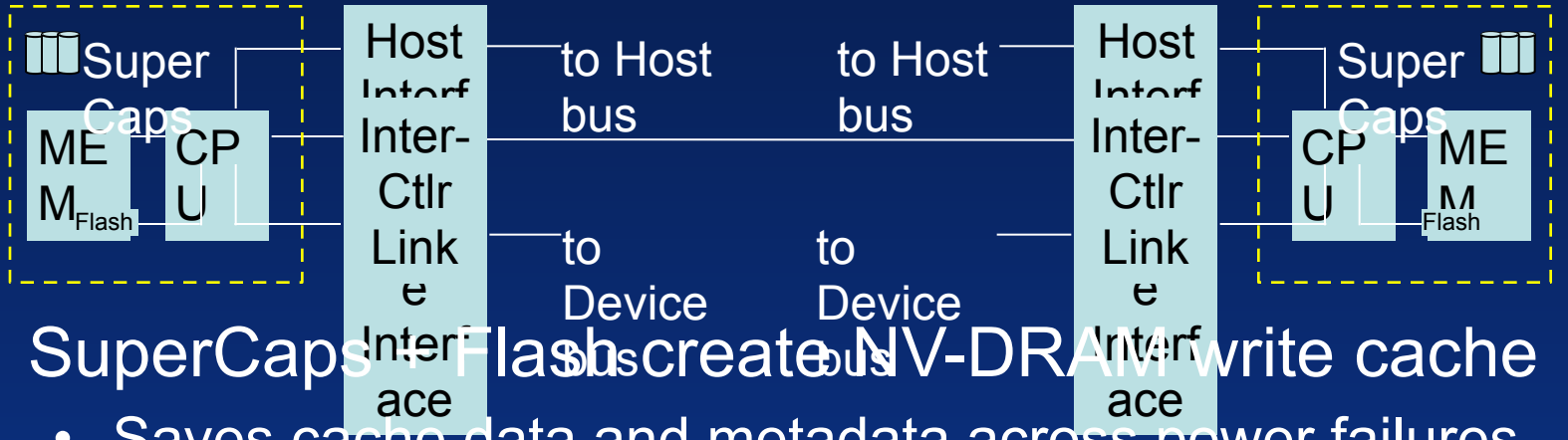
History of Storage Caching

- Storage Caching was introduced in the 1980's
  - IBM 3990 and DEC HSC70 were early caching storage controllers, using DRAM to cache magnetic disk data
  - Early storage caches were fully associative and write-through, but did incorporate read-ahead
  - Write-back storage caches introduced in late 1980's, with battery backup for cache DRAMs.
  - Small write-through (or unsafe write-back) caches implemented in magnetic disk drives in the late 1980's.

Read Cache Competition

- Host OS's all implement file-level read cache, and some implement (semi-unsafe) write cache as well.
- Storage controller read cache sits in the "shadow" of the host read caches, greatly reducing its effectiveness
- Controller read cache is still useful when multiple hosts access the same data (e.g. VDI "boot storms")
- Storage controller read-ahead is generally more aggressive than host read-ahead and is still effective

DRAM Caching in Storage Servers



- SuperCaps + Flash create NV-DRAM write cache
  - Saves cache data and metadata across power failures
- Inter-controller link allows cache mirroring (NSPF)
  - FW Mirrors cache data and metadata to other controller

DRAM Caching in Storage Servers

- DRAM read caching in a storage controller has "zero" overhead cost!
  - On a read miss, device data must be read into DRAM anyway for host transfer – only need to update metadata
  - Only overhead comes from read-ahead data that isn't used
- DRAM write caching in a storage controller has very small overhead cost
  - Write data must be loaded into DRAM anyway for device transfer – need to mirror it and update metadata

Flash Caching in Storage Systems

- Because of wear-leveling concerns, the simplest way to use flash in a storage system is as an SSD.
  - This will probably be true for 3D XPoint as well
- Low flash cost (~1/10th DRAM cost) allows read caches that are considerably larger than those in host systems
  - Flash cache metadata must be kept in DRAM to be searched
  - Lots of DRAM…
- Flash can be used as the only cache, or as a "second level" cache between DRAM and magnetic disk.

Flash Read Cache

- Flash read cache is the simplest form of Flash cache to implement
  - No need for fault tolerance or non-volatility of data and metadata
- Flash read cache does nothing to improve writes
  - It might add overhead to them, depending on write policy
- Performance improvement limited by read/write ratio
  - 50% write ratio → no more than 2:1 performance improvement
- Benefit of Read-ahead cache might be limited
  - Depending on what other techniques used to speed up writes

Flash Read Cache Overhead

- When a cache miss occurs in the Flash, data must be fetched from magnetic disk – this is not overhead
- Once the data is read into DRAM, it must be written out to the Flash cache – this is overhead
  - There are no effective criteria to determine which misses will never be accessed again, so all misses must be cached “on the come”
  - If the cached data is accessed again, then the overhead paid off in reduced I/O latency and increased performance – if not, it’s wasted overhead

Flash Cache Read Overhead (2)

- Cache miss overhead reduces the cache performance
  - Not a big problem for SSD caching HDD, because the performance ratio is high
  - Would be a problem if ratio is low – i.e. 3D XPoint caching Flash

"The smaller the performance ratio between tiers, the more thought you have to apply to move decisions"

Flash Write Cache

- Acts like a Flash read cache on reads
- All writes are first written to Flash SSD
  - After being read into controller DRAM from host
- Cache Replacement Policy causes dirty data to be flushed from Flash SSD to HDD
  - Using controller DRAM as a buffer for the I/O operation
- Data and metadata must be non-volatile & fault-tolerant

Flash Write Cache Overhead

- Read overhead is same as for Flash Read Cache
- Write overhead is significantly worse
  - Dirty data must be written to at least 2 SSDs for redundancy
  - Metadata must also be written out (2X) in recoverable form
  - Writes must be transactional (i.e. atomic)
  - Data must eventually be read back to send to HDD
  - Write overhead is >4X read overhead
- This is why SSD/HDD hybrid systems generally don't use SSD as a write cache

Improving Flash Cache overheads in an SSD/HDD hybrid

- Step 1 – make Flash a middle hierarchy level between a DRAM write-back cache and HDD
  - Improves performance in general when data hits in the DRAM
  - No need to write Flash metadata to Flash (DRAM is safe)
  - NV DRAM provides transactional control for SSD writes
- Step 2 – implement Flash cache as a victim cache for DRAM, instead of as a cache for HDD
  - Allows more time to collect information on access patterns to decide whether storing data in Flash is justified

History of Tiering

- Tiering was originally done manually in the 1960's in the High Performance Computing community
  - Load data from tape to disk, run monster job, write modified data and results onto (new) tape
- IBM DFHSM (1970's): tiering between disk and tape
  - IBM's user community (SHARE/GUIDE) helped define it
  - Moved files or groups of files between disk and tape
  - Semi-automatic, based on user-defined policies and scripts
  - Eventually morphed into Archiving facility

Online Tiering in HDD-only Systems

- The advent of high-speed (7200-15000 RPM) disk drives in the 1990's drove a need to place online data on media appropriate to its access demands
- BUT – the performance ratio of "fast & pricey" disks to "cheap & slow" disks was only in the 2-3:1 range
  - Therefore a lot of thought had to go into deciding what to move
- In the same timeframe, "storage virtualization" became a popular way to reduce storage management burdens
  - The fusion of these ideas led to the first "auto-tiering" systems

Tiering Design: Virtualization

- Tiering algorithms are piggybacked on an existing page-based storage virtualization architecture
  - Fixed-size pages mapped from virtual to logical devices using multi-level page tables, a la virtual memory in CPUs
  - Storage page sizes generally larger than CPU pages – anywhere from 64KB to 8MB – for performance reasons
- The “virtual storage page” is the basic unit of storage that is moved between hierarchy members by tiering

Tiering Design: Basic Algorithm

- For some period of time, count the number of I/O operations to every storage page in the system
- At the end of that period:
  - Move slow media pages with highest I/O counts to fast media
  - Need fast media space? Move fast media pages with lowest I/O counts to slow media
  - In steady state, “swaps” fast pages with slow pages
- Repeat as necessary
- Simple, eh?

Tiering Design: ROI-based Decisions

- How many pages to move each period?
  - Move too few pages – unresponsive to workload changes
  - Move too many pages – too much move overhead
- Take a hint from business – use ROI methodology
  - Migrating a page has a performance cost, incurred up front
  - Migration "pays off" over time in improved performance of that page on fast media vs slow media
  - Knowing historic I/O rate and move cost, calculate ROI
  - Apply "ROI Hurdle" (tunable parameter) to make move decision

Tiering Design: the Noise Problem

"The smaller the ratio in performance between tiers, the more thought you have to apply to move decisions"

Consider a 2TB slow HDD virtualized with 1MB pages

- There are 2 million pages in the HDD
- The HDD can, flat-out, do 125 operations per second
- Average page will see .0000625 ops/sec = 5.4 op/***day***
  - A "hot" page will see > 10X this rate
- A simple load of a Word document will produce 50 or more read ops to the same page in a few milliseconds

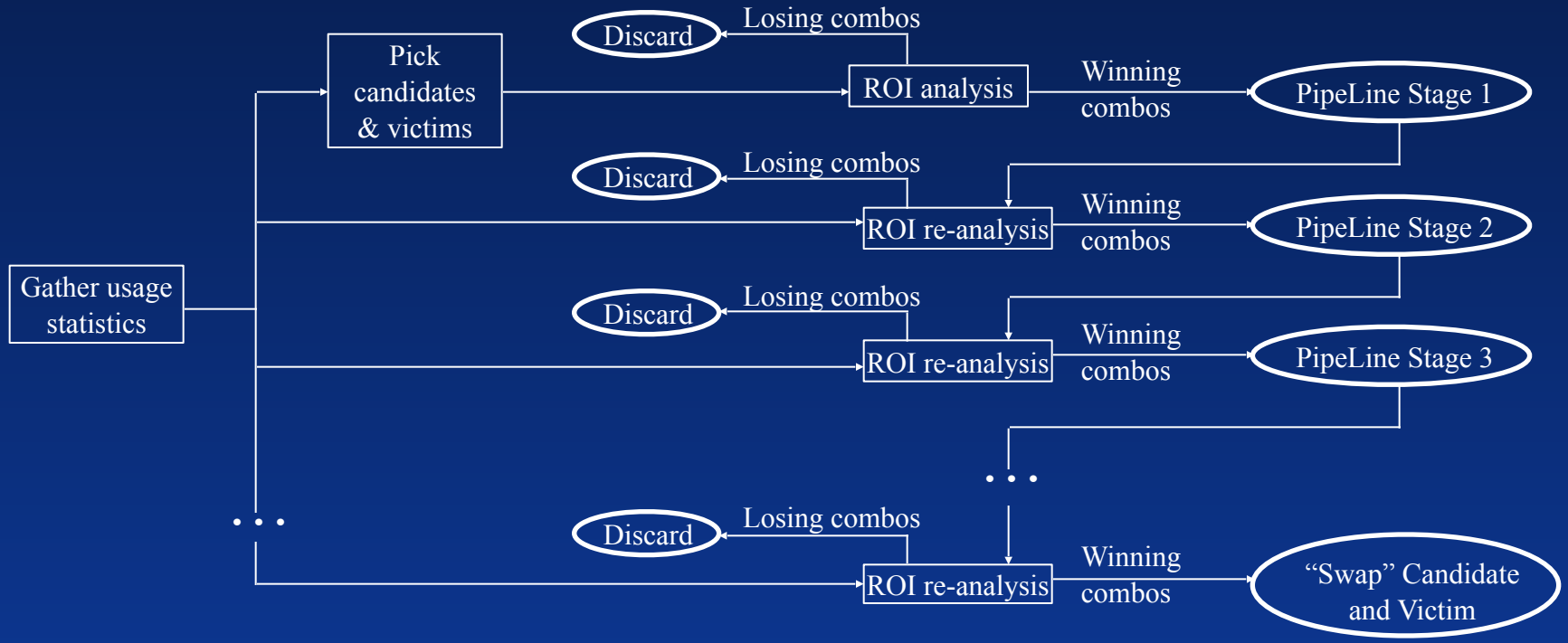
Tiering: Dealing with Noise

- Collect data for loooong periods before making decision
  - That's what many HDD-HDD tiering systems did: period = 1 day
  - Efficient but not dynamic – doesn't adapt to changing workload
- Smaller number of larger pages
  - Reduces noise, but you move larger chunks of data
  - Fortunately, HDDs are efficient for large data moves
- Flash-based tiering has better performance ratio
  - More tolerance for bad decisions, but bad decisions are still bad
- Improved decision making algorithms

Tiering Design: Decision Pipeline

- Shorten decision period by factor of N
  - Increases effect of noise
- Make a tentative decision based on ROI criteria
- Ratify decision in each of N consecutive periods
- Act on decision if ratified in M out of those N periods
- Adds “consistency” criteria to decision process
  - Only pages demonstrating consistent I/O activity get moved
  - Decreases effect of noise compared to using the original period

Pipelined Decision Flowchart



Comparing Caching and Tiering

- Tiering moves bigger chunks of data than Caching, but moves data less often
  - Moving data in big chunks is more efficient on HDD
- Tiering allows full utilization of all media – no copies
  - Becomes significant when fast tier is > 10% of total storage
- Caching is more responsive to app changes than tiering
  - Responds in milliseconds rather than 10s of seconds
- Tiering requires much less DRAM metadata than caching
  - Typically less than 1/100th as much

Combining Caching and Tiering

DRAM caching and Flash tiering work very well together!

- Cache responds instantaneously to I/O pattern changes
- DRAM Cache captures the small “red-hot” areas of activity, so tiering can work well with large page sizes
- Tiering adds Flash capacity to disk capacity, amortizing cost

3D XPoint, Caching and Tiering

- 3D XPoint can easily substitute for Flash in any caching or tiering hierarchy
- Not clear whether 3D XPoint in SSD form is useful as an additional hierarchy level between DRAM and Flash
  - Performance ratio is on the low side
- Best use of 3D Xpoint would be to integrate its need for wear-leveling into a caching/tiering algorithm, so it can be used as a cheaper, slower, non-volatile DRAM rather than as a faster SSD

3D XPoint, Caching and Tiering (2)

- An interesting use of 3D XPoint technology would be to use tiering not for performance, but for endurance
  - 3D XPoint has ~1000X the endurance of Flash
  - Use tiering algorithms between 3D XPoint and TLC/QLC Flash, but only count Write operations to determine what to move
  - Heavily written data areas wind up on 3D Xpoint
  - Fewer writes go to Flash
  - Flash endurance is extended!

Questions?