Speeding Up Cloud/Server Applications Using Flash Memory

Sudipta Sengupta
Microsoft Research, Redmond, WA, USA

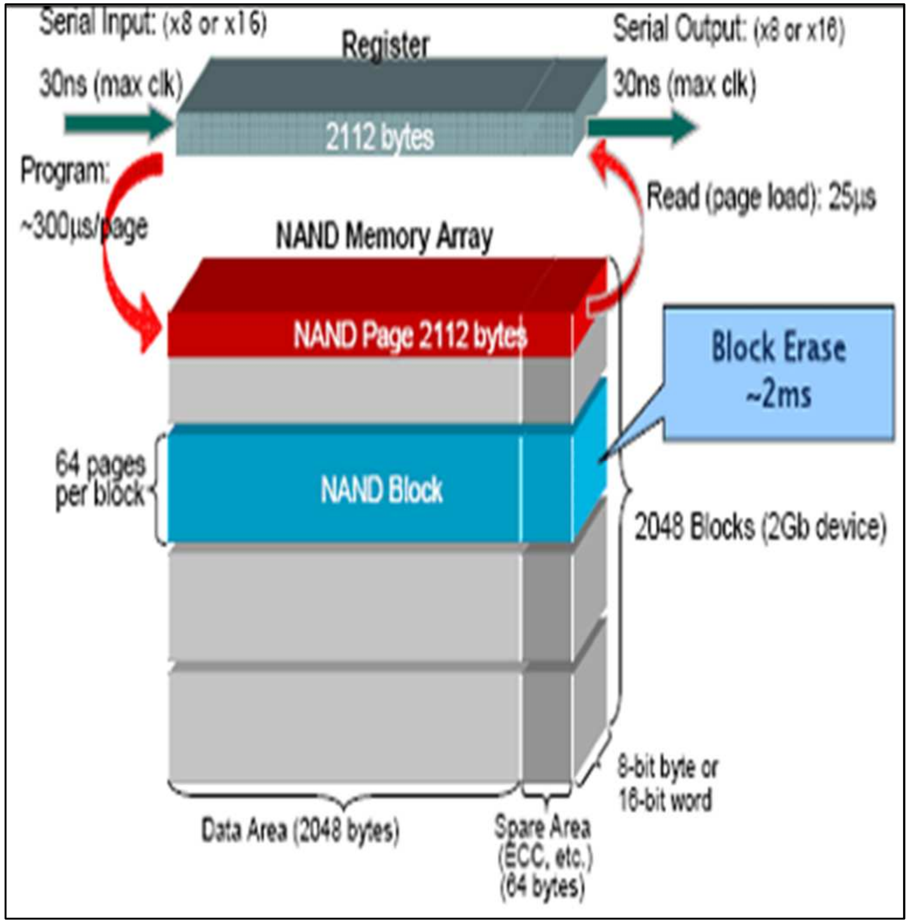
Contains work that is joint with B. Debnath (Univ. of Minnesota) and J. Li (Microsoft Research, Redmond)

Flash Memory

- Used for more than a decade in consumer device storage applications
- Very recent use in desktops and servers
  - New access patterns (e.g., random writes) pose new challenges for delivering sustained high throughput and low latency
  - Higher requirements in reliability, performance, data life
- Challenges being addressed at different layers of storage stack
  - Flash device vendors: device driver/ inside device
  - System builders: OS and application layers, e.g., Focus of this talk

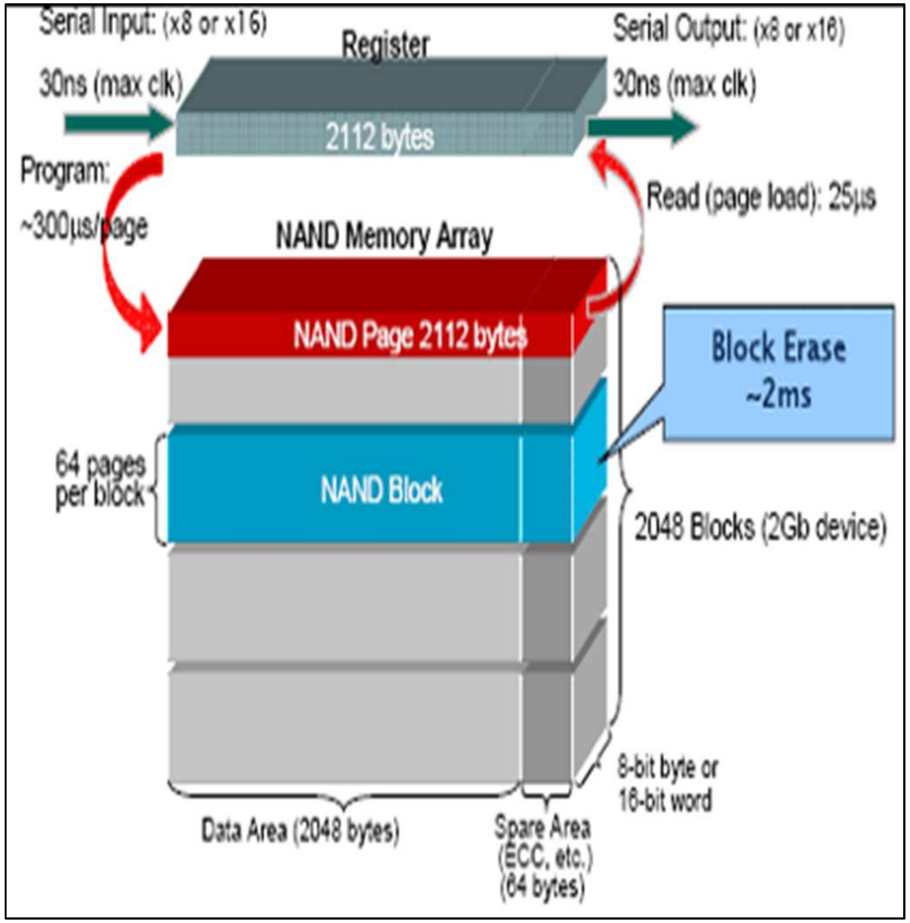
Flash Memory contd. …

- Performance and cost between RAM and HDD
  - 10-100 usec access times
  - About 10x cheaper than RAM and 10x more expensive than HDD
    - MLC: $3/GB, SLC: $10/GB
  - Can benefit applications that can find the sweet spot between cost and performance



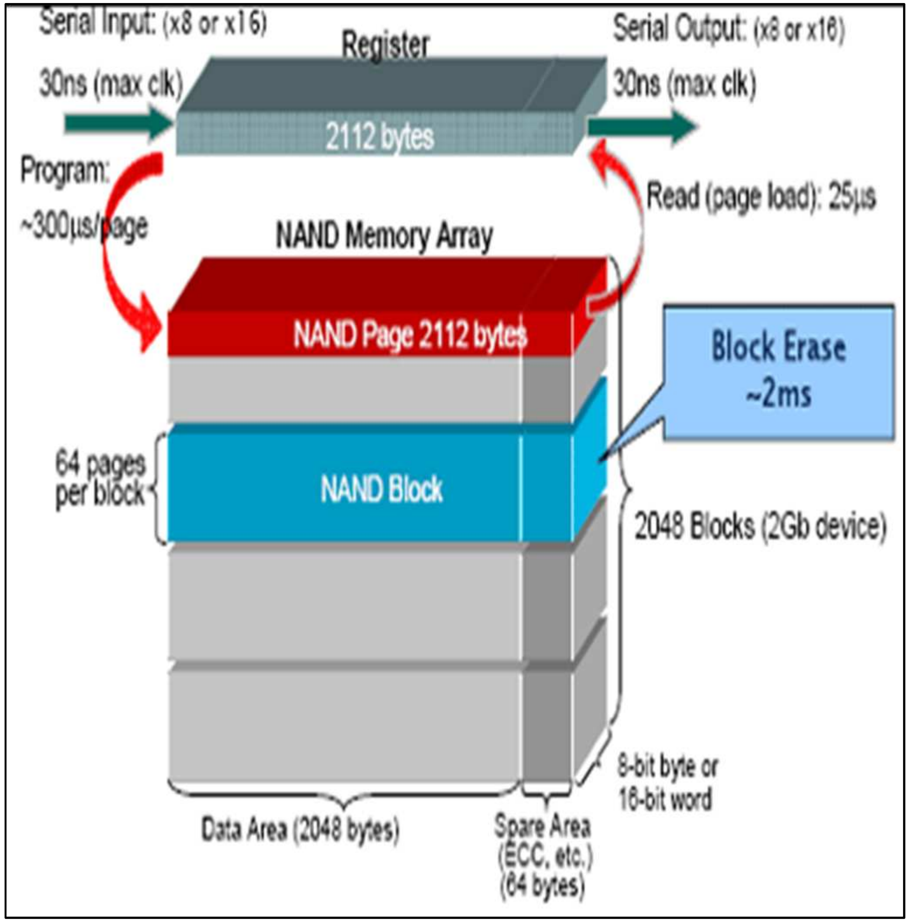
Background: NAND Flash Memory

- An array of flash blocks
  - No mechanical delay (vs. seek in HDD)
- Read/write in page units
  - Typical **page = 2K, block = 128K**
  - Must erase block before write
  - Random/sequential reads good (10-100 usec)
  - Sequential writes good
  - Small random writes perform poorly (leads to write amplification)
  - Block erase (1500 usec)
- Flash Translation Layer (FTL)
  - Translate from logical page # to physical page #
  - Block level mapping table (to reduce table size) + temporary page level mapping table



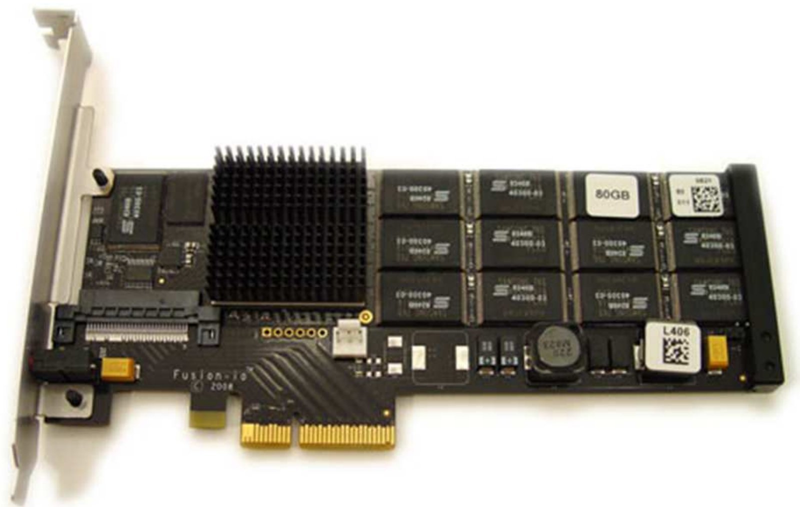
Background: NAND Flash Memory

- Limited number of erases per block
  - **100K** for **SLC**; **10K** for **MLC**
  - Use wear-leveling to distribute erases across blocks
- Lower power consumption
  - ~ 1/3 of 7200 HDD
  - ~ 1/6 15k HDD
- Higher ruggedness (100x in active state)

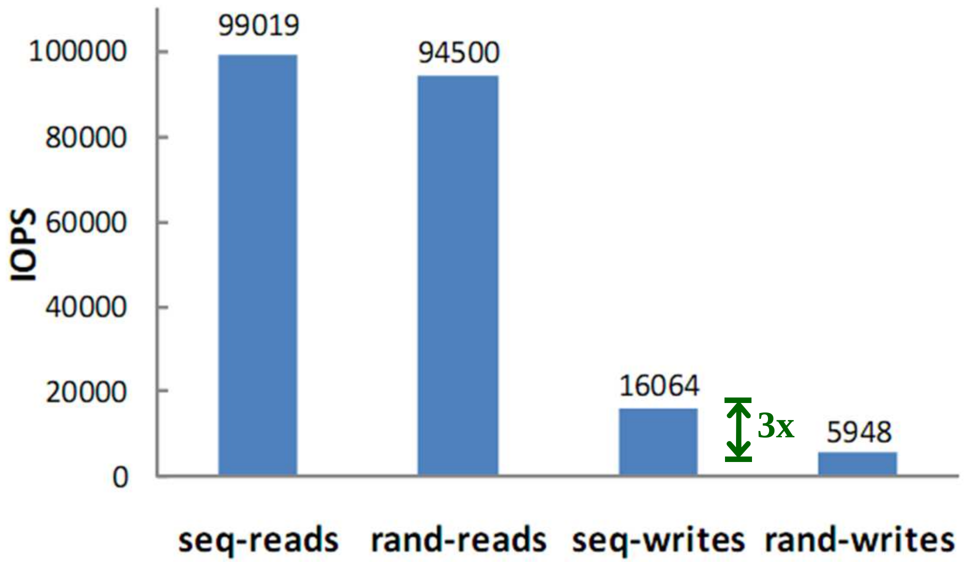


Flash Memory: Random Writes

- Need to optimize the storage stack for making best use of flash
  - Random writes not efficient on flash media
  - Flash Translation Layer (FTL) cannot hide or abstract away device constraints



FusionIO 160GB ioDrive



Flash for Speeding Up Cloud/Server Applications

- FlashStore [VLDB 2010]
  - High throughput, low latency persistent key-value store using flash as cache above HDD
- ChunkStash [USENIX ATC 2010]
  - Speeding up storage deduplication using flash memory
- BloomFlash
  - Bloom filter on flash with low RAM footprint
- SkimpyStash
  - Key-value store with ultra-low RAM footprint at about 1-byte per k-v pair
- Flash as block level cache above HDD
  - Either application managed or OS managed
  - Lower cost design point with acceptable performance for applications that do not need super blazing RAM cache speeds

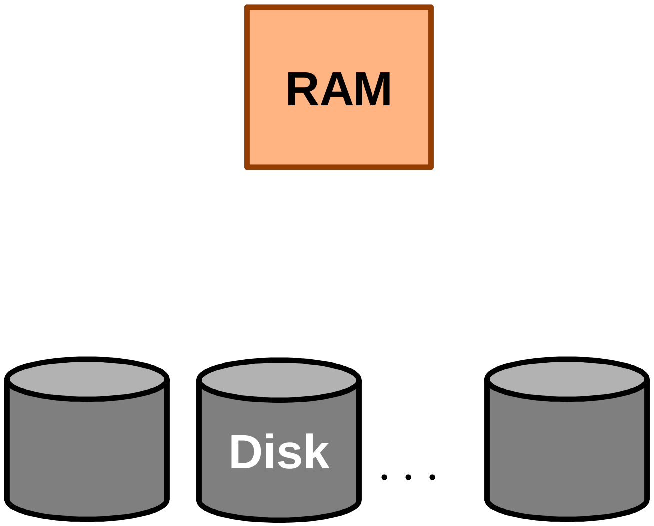
FlashStore: High Throughput Persistent Key-Value Store

Design Goals and Guidelines

- Support low latency, high throughput operations as a key-value store
- Exploit flash memory properties and work around its constraints
  - Fast random (and sequential) reads
  - Reduce random writes
  - Non-volatile property
- Low RAM footprint per key independent of key-value pair size

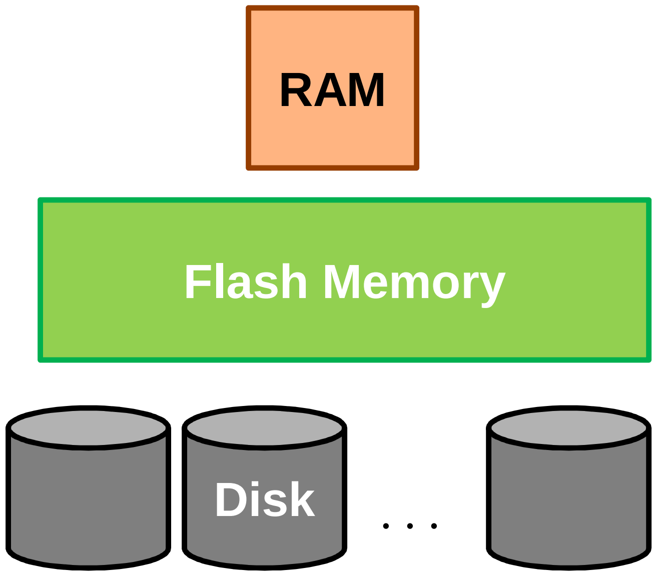
FlashStore Design: Flash as Cache

- Low-latency, high throughput operations
- Use flash memory as cache between RAM and hard disk



Current

(bottlenecked by hard disk seek times ~ 10msec)

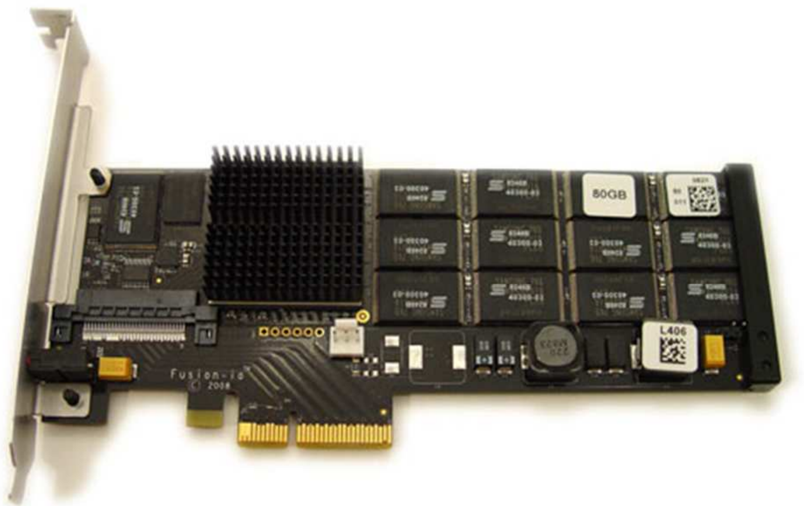


FlashStore

(flash access times are of the order of 10 -100 μsec)

FlashStore Design: Flash Awareness

- Flash aware data structures and algorithms
  - Random writes, in-place updates are expensive on flash memory
    - Flash Translation Layer (FTL) cannot hide or abstract away device constraints
  - Sequential writes, Random/Sequential reads great!
- Use flash in a log-structured manner

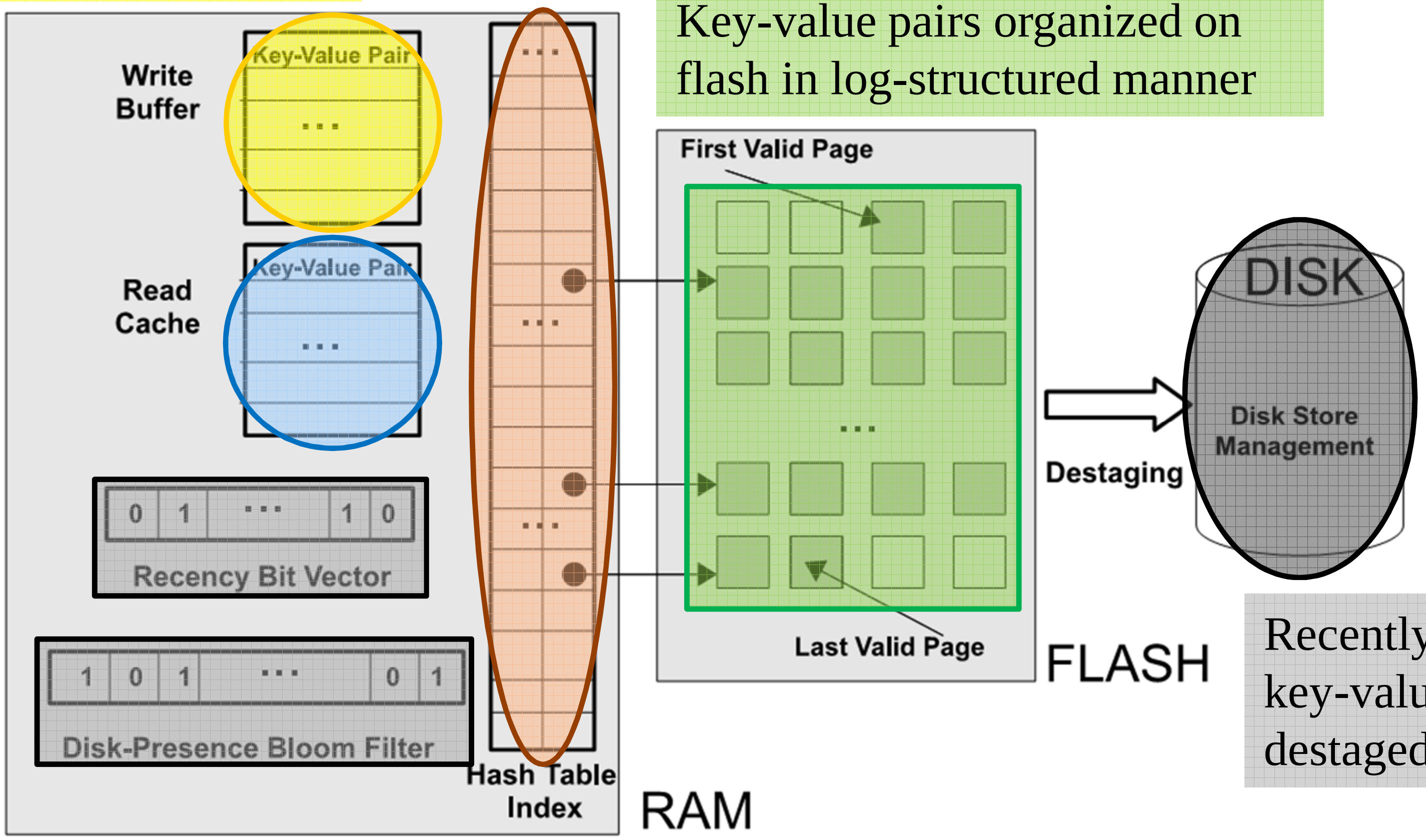


FusionIO 160GB ioDrive

FlashStore Architecture

RAM write buffer for aggregating writes into flash

Key-value pairs organized on flash in log-structured manner

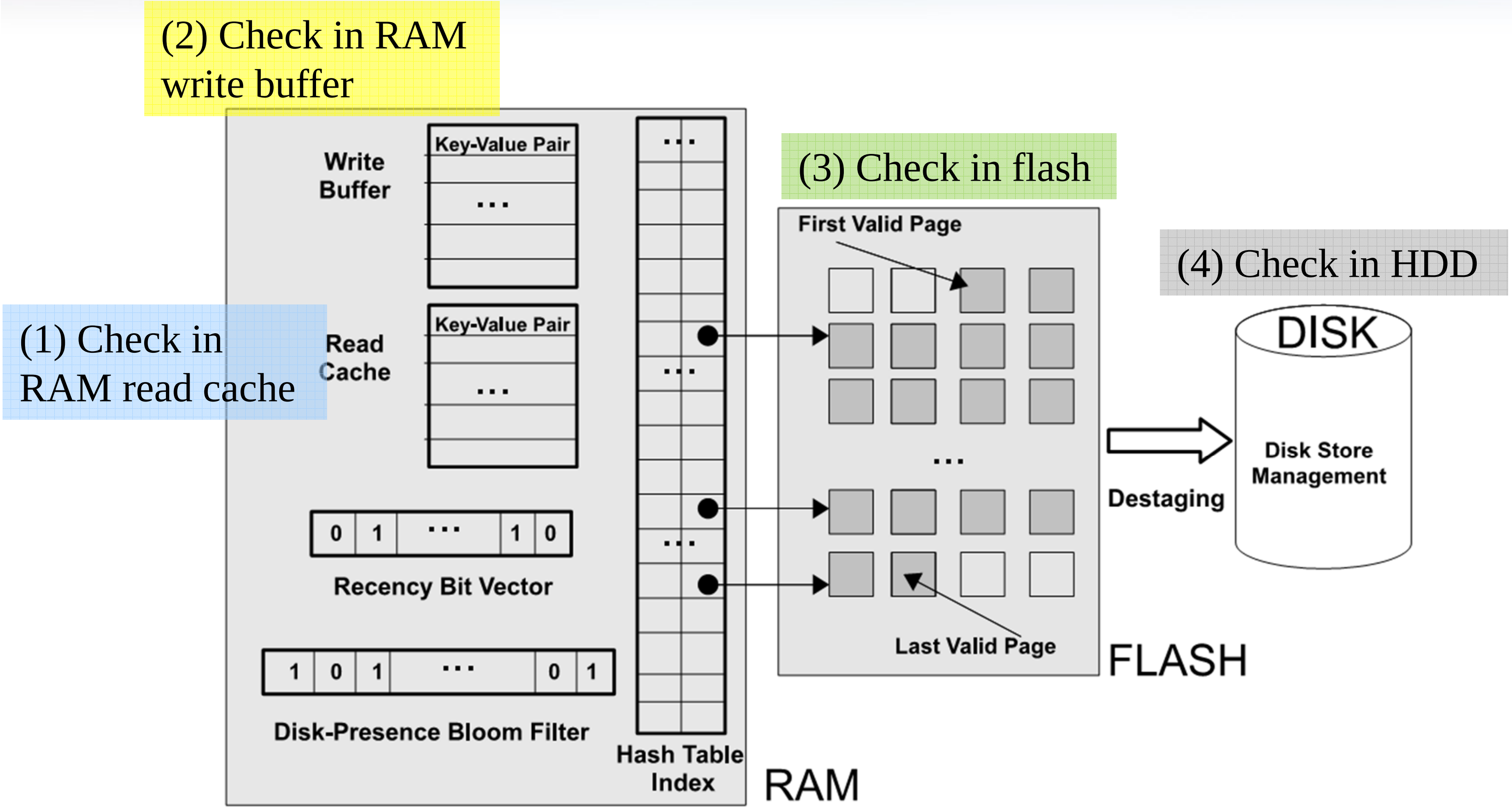


Recently unused key-value pairs destaged to HDD

RAM read cache for recently accessed key-value pairs

Key-value pairs on flash indexed in RAM using a specialized space efficient hash table

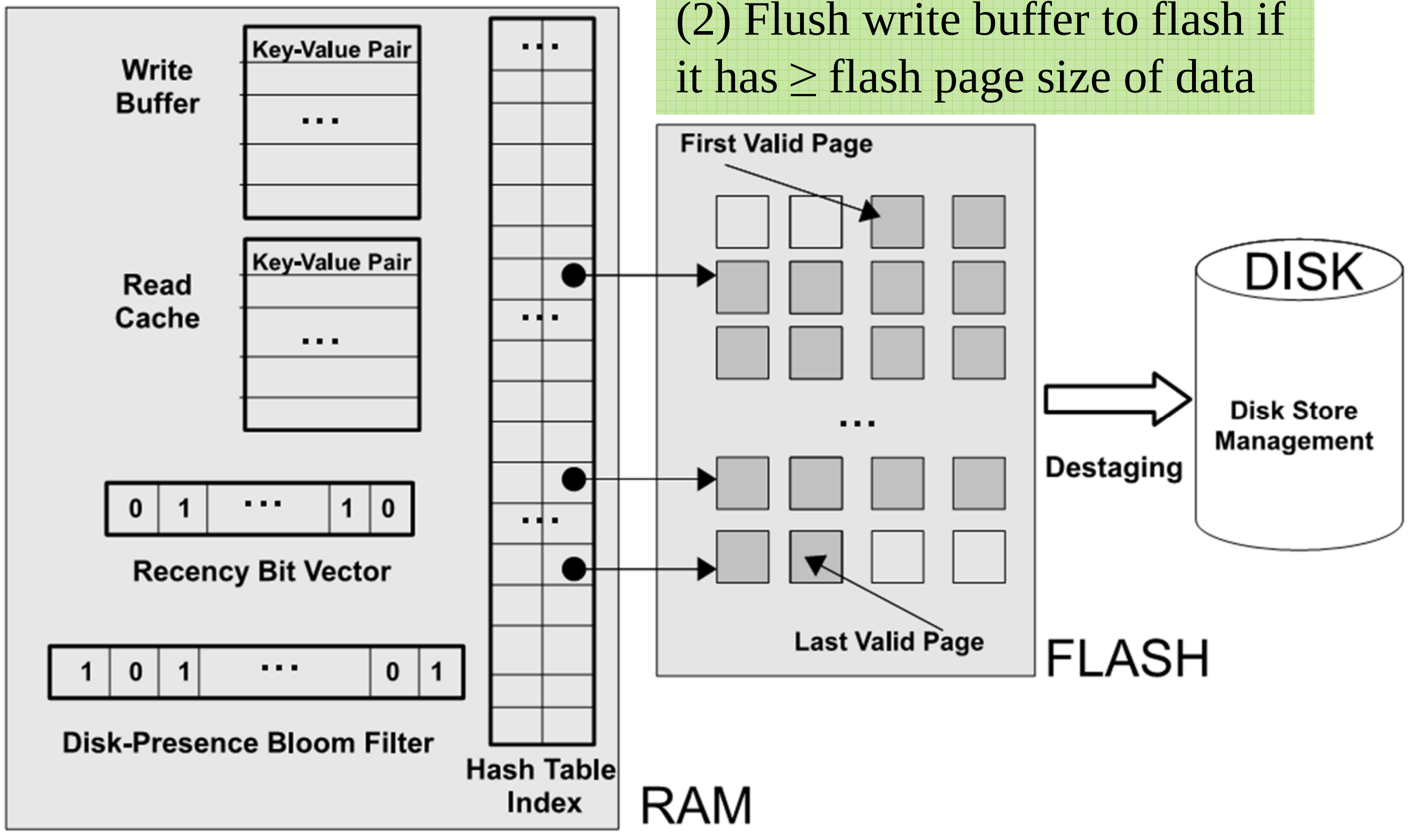
FlashStore: Get Operation



FlashStore: Set Operation

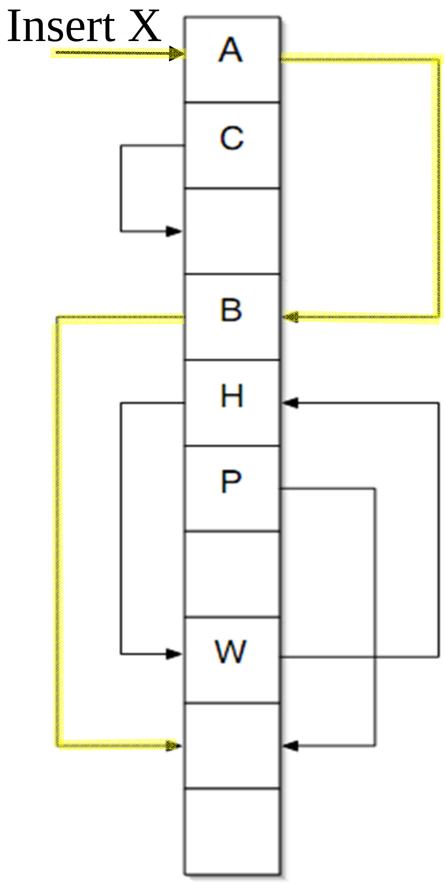
(1) Write key-value pair to RAM write buffer

(2) Flush write buffer to flash if it has ≥ flash page size of data



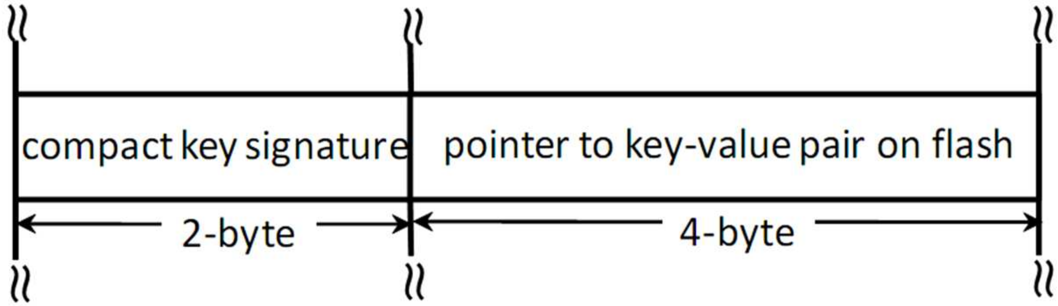
FlashStore Design: Low RAM Usage

- High hash table load factors while keeping lookup times fast
  - Collisions resolved using cuckoo hashing
  - Key can be in one of K candidate positions
  - Later inserted keys can relocate earlier keys to their other candidate positions
  - K candidate positions for key x obtained using K hash functions $h_1(x)$, …, $h_K(x)$
  - In practice, two hash functions can simulate K hash functions using $h_i(x) = g_1(x) + i*g_2(x)$
- System uses value of K=16 and targets 90% hash table load factor



Low RAM Usage: Compact Key Signatures

- Compact key signatures stored in hash table
  - 2-byte key signature (vs. key length size bytes)
  - Key x stored at its candidate position i derives its signature from $h_i(x)$
  - False flash read probability < 0.01%
- Total 6-10 bytes per entry (4-8 byte flash pointer)



- Related work on key-value stores on flash media
  - MicroHash, FlashDB, FAWN, BufferHash

RAM and Flash Capacity Considerations

- Whether RAM or flash size becomes bottleneck for cache size on flash depends on key-value pair size
- Example: 4GB of RAM
  - 716 million key-value pairs @6 bytes of RAM per entry
  - At 64-byte per key-value pair, these occupy 42GB on flash
    - => RAM is bottleneck for key-value pair capacity on flash
  - At 1024-byte per key-value pair, these occupy 672GB on flash
    - => Flash is bottleneck for key-value pair capacity on flash, need multiple attached flash drives

Flash Recycling

- Arising from use of flash in log-structured manner
- Recycle page by page in oldest written order (starting from head of log)
  - Triggered by configurable threshold of flash usage
- Three cases for key-value pairs on a recycled flash page
  - Some entries are obsolete
  - Some entries are frequently accessed
    - Should remain in flash memory
    - Reinserted into write buffer and subsequently to tail of log
  - Some entries are not frequently accessed
    - Destaged to hard disk

FlashStore Performance Evaluation

Hardware Platform

- Intel Processor, 4GB RAM, 7200 RPM Disk, fusionIO SSD
- Cost without flash = $1200
- Cost of fusionIO 80GB SLC SSD = $2200 (circa 2009)

CPU		RAM		Flash (SSD)			Hard Disk (HDD)			Chassis
Type	Power	Size	Power	Type	Cost	Power	Type	Cost	Power	Cost
Intel Core 2 Duo E8500 @3.16GHz	65W	4GB	3.5W	fusionIO 80GB	$2200	15W	Seagate Barracuda 250GB 7200rpm	$50	12W	$1150

Trace

- Xbox LIVE Primetime
- Storage Deduplication

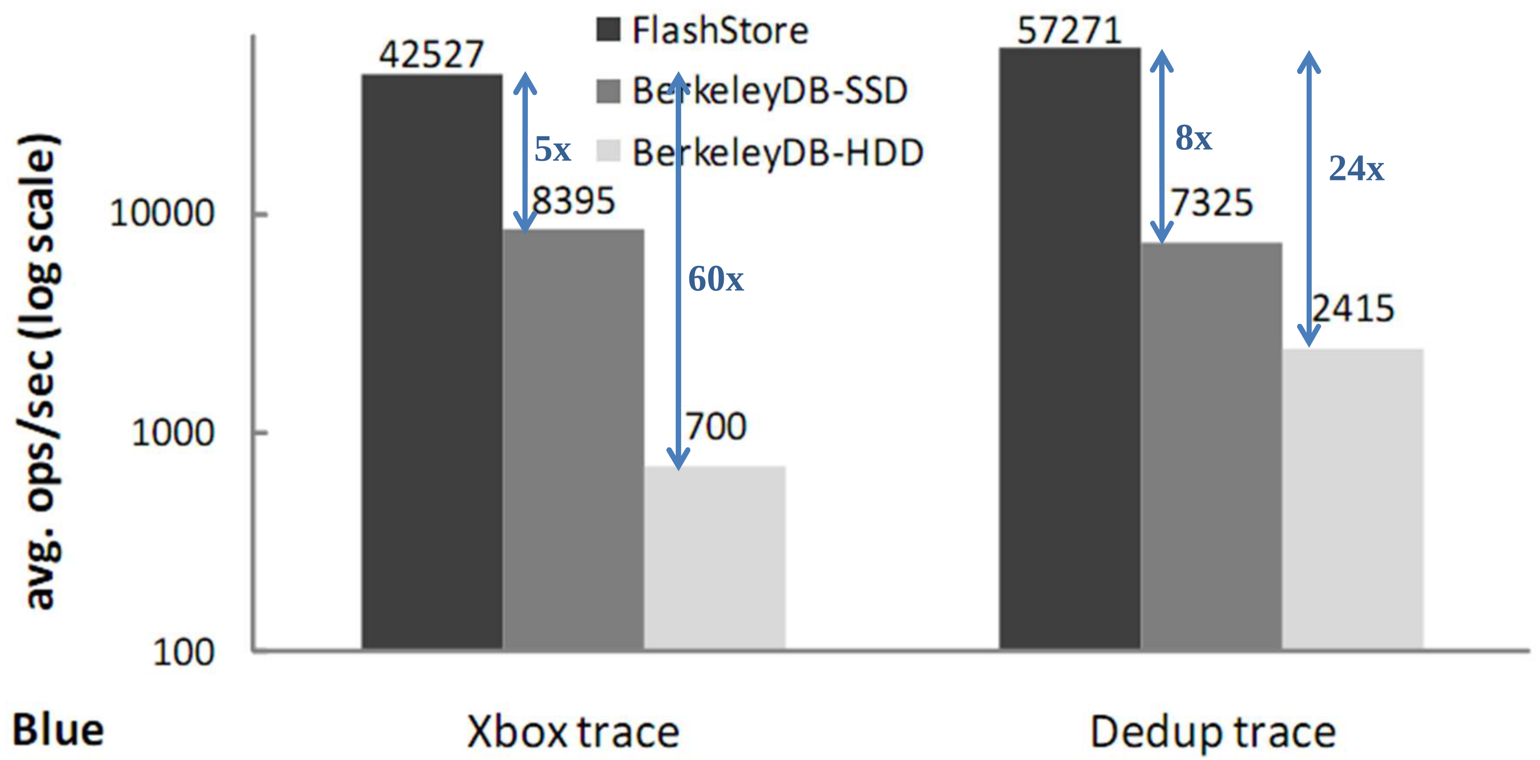
Trace	Total get-set ops	get:set ratio	Avg. size (bytes)	
			Key	Value
Xbox	5.5 million	7.5:1	92	1200
Dedup	40 million	2.2:1	20	44

FlashStore Performance Evaluation

- How much better than simple hard disk replacement with flash?
  - Impact of flash aware data structures and algorithms in FlashStore
- Comparison with flash unaware key-value store
  - FlashStore-SSD
  - BerkeleyDB-HDD
  - BerkeleyDB-SSD
  - FlashStore-SSD-HDD (evaluate impact of flash recycling activity)

BerkeleyDB used as the flash unaware index on HDD/SSD

Throughput (get-set ops/sec)

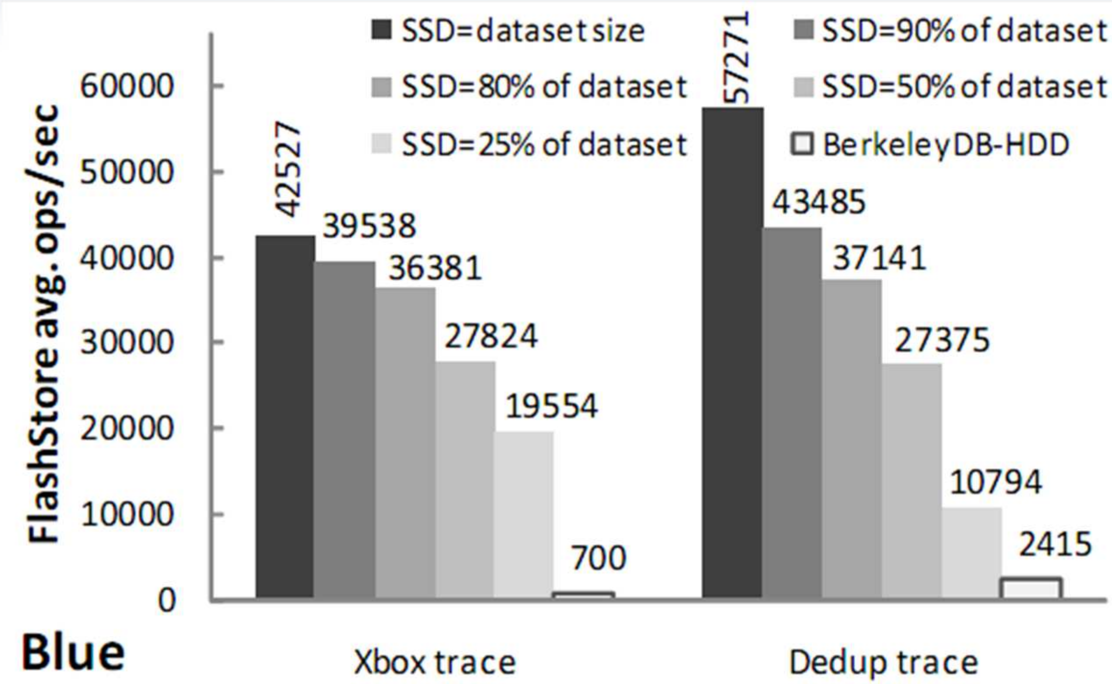


Performance per Dollar

- From BerkeleyDB-HDD to FlashStore-SSD
  - Throughput improvement of ~ 40x
  - Flash investment = 50% of HDD capacity (example)

    = 5x of HDD cost (assuming flash costs 10x per GB)
  - Throughput/dollar improvement of about 40/6 ~ 7x

Impact of Flash Recycling Activity



- Graceful degradation in throughput as flash capacity is reduced
- Performance on Xbox trace drops less sharply vs. for dedup trace
- Even at SSD size = 25% of dataset, FlashStore throughput >> BerkeleyDB-HDD

Summary

- Designed FlashStore to be used as a high-throughput persistent key-value storage layer
  - Flash as cache above hard disk
  - Log structured organization on flash
  - Specialized low RAM footprint hash table to index flash
- Evaluation on real-world data center applications
  - Xbox LIVE Primetime online gaming
  - Storage deduplication
- Significant performance improvements
  - Vs. BerkeleyDB running on hard disk or flash separately
  - Of 1-2 orders of magnitude on the metric of throughput (ops/sec) and 1 order of magnitude on cost efficiency (ops/sec/dollar)
  - For both applications

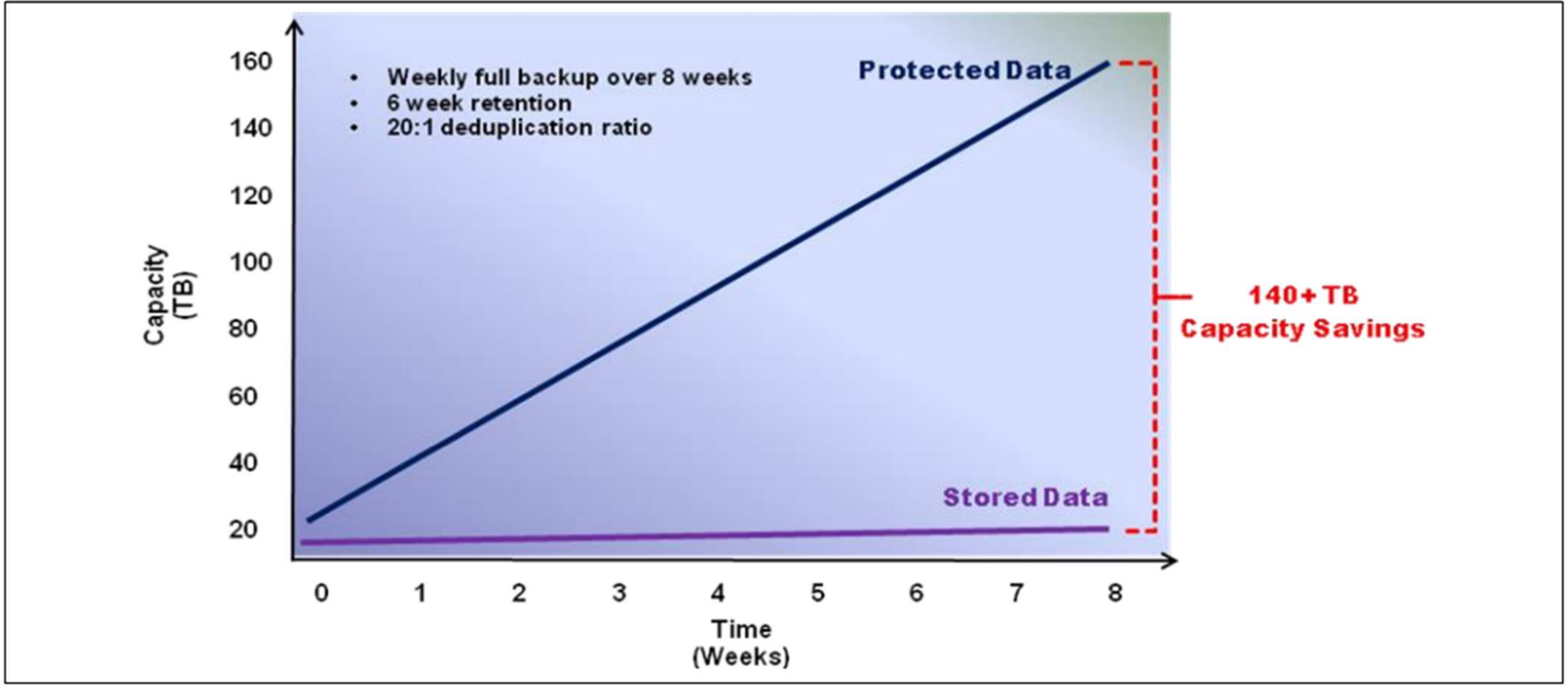
ChunkStash: Speeding Up Storage Deduplication using Flash Memory

Deduplication of Storage

- Detect and remove duplicate data in storage systems
  - e.g., Across multiple full backups
  - Storage space savings
  - Faster backup completion: Disk I/O and Network bandwidth savings
- Feature offering in many storage systems products
  - Data Domain, EMC, NetApp
- Backups need to complete over windows of few hours
  - Throughput (MB/sec) important performance metric
- High-level techniques
  - Content based chunking, detect/store unique chunks only
  - Object/File level, Differential encoding

Impact of Dedup Savings Across Full Backups

FIGURE 3. DEDUPLICATION IMPACT



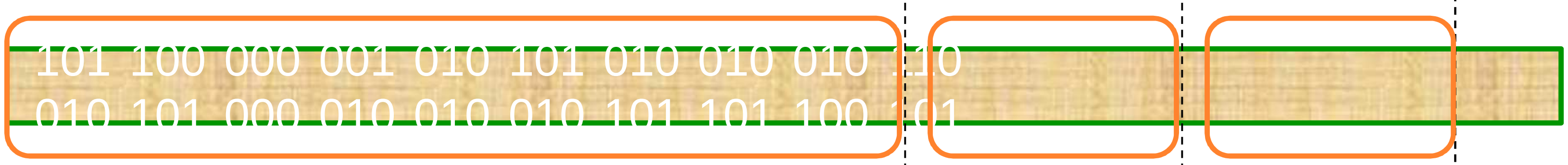
Source: Data Domain white paper

Deduplication of Storage

- Detect and remove duplicate data in storage systems
  - e.g., Across full backups
  - Storage space savings
  - Faster backup completion: Disk I/O and Network bandwidth savings
- Feature offering in many storage systems products
  - Data Domain, EMC, NetApp
- Backups need to complete over windows of few hours
  - Throughput (MB/sec) important performance metric
- High-level techniques
  - Content based chunking, detect/store unique chunks only
  - Object/File level, Differential encoding

Content based Chunking

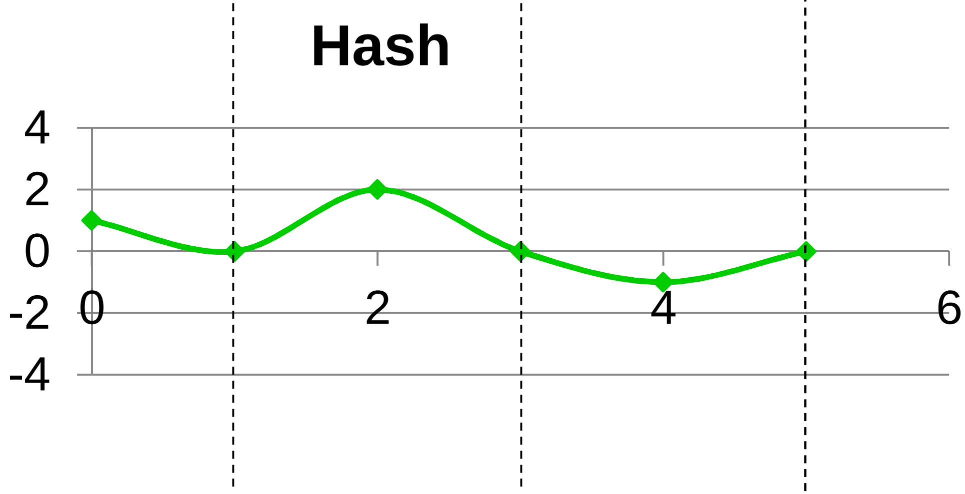
- Calculate Rabin fingerprint hash for each sliding window (16 byte)



3 Chunks

If Hash matches a particular pattern,

Declare a chunk boundary

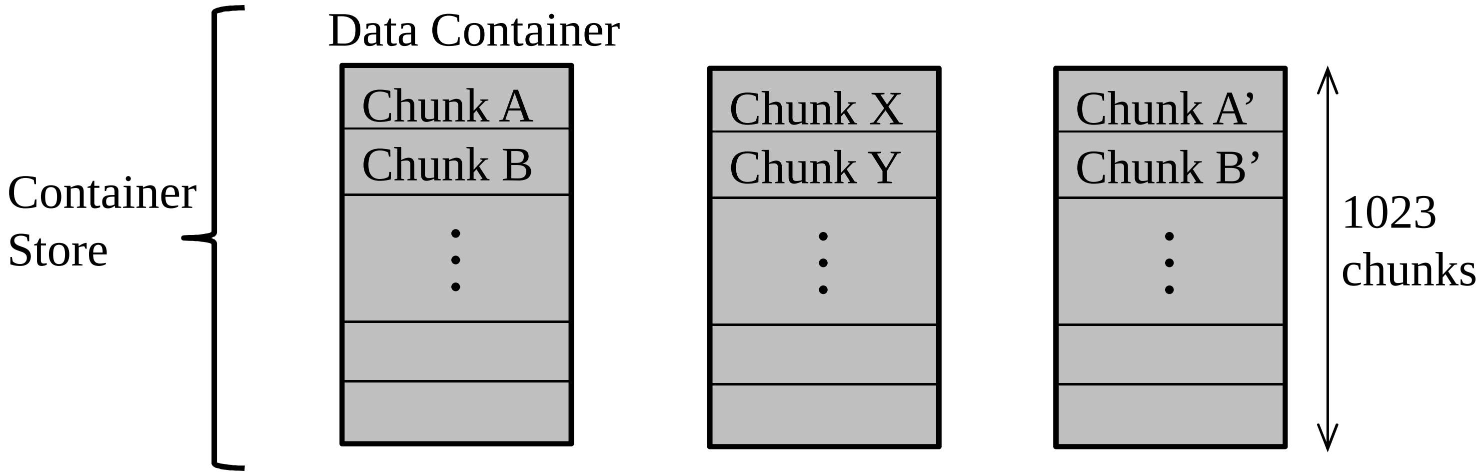


How to Obtain Chunk Boundaries?

- Content dependent chunking
  - When last n bits of Rabin hash = 0, declare chunk boundary
  - Average chunk size = 2^n bytes
  - When data changes over time, new chunks correspond to new data regions only
- Compare with fixed size chunks (e.g., disk blocks)
  - Even unchanged data could be detected as new because of shifting
- How are chunks compared for equality?
  - 20-byte SHA-1 hash (or, 32-byte SHA-256)
  - Probability of collisions is less than that of hardware error by many orders of magnitude

Container Store and Chunk Parameters

- Chunks are written to disk in groups of containers
  - Each container contains 1023 chunks
  - New chunks added into currently open container, which is sealed when full
  - Average chunk size = 8KB, Typical chunk compression ratio of 2:1
  - Average container size ≈ 4MB

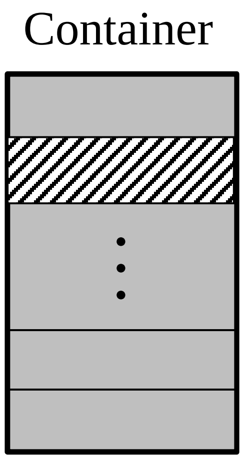


Index for Detecting Duplicate Chunks

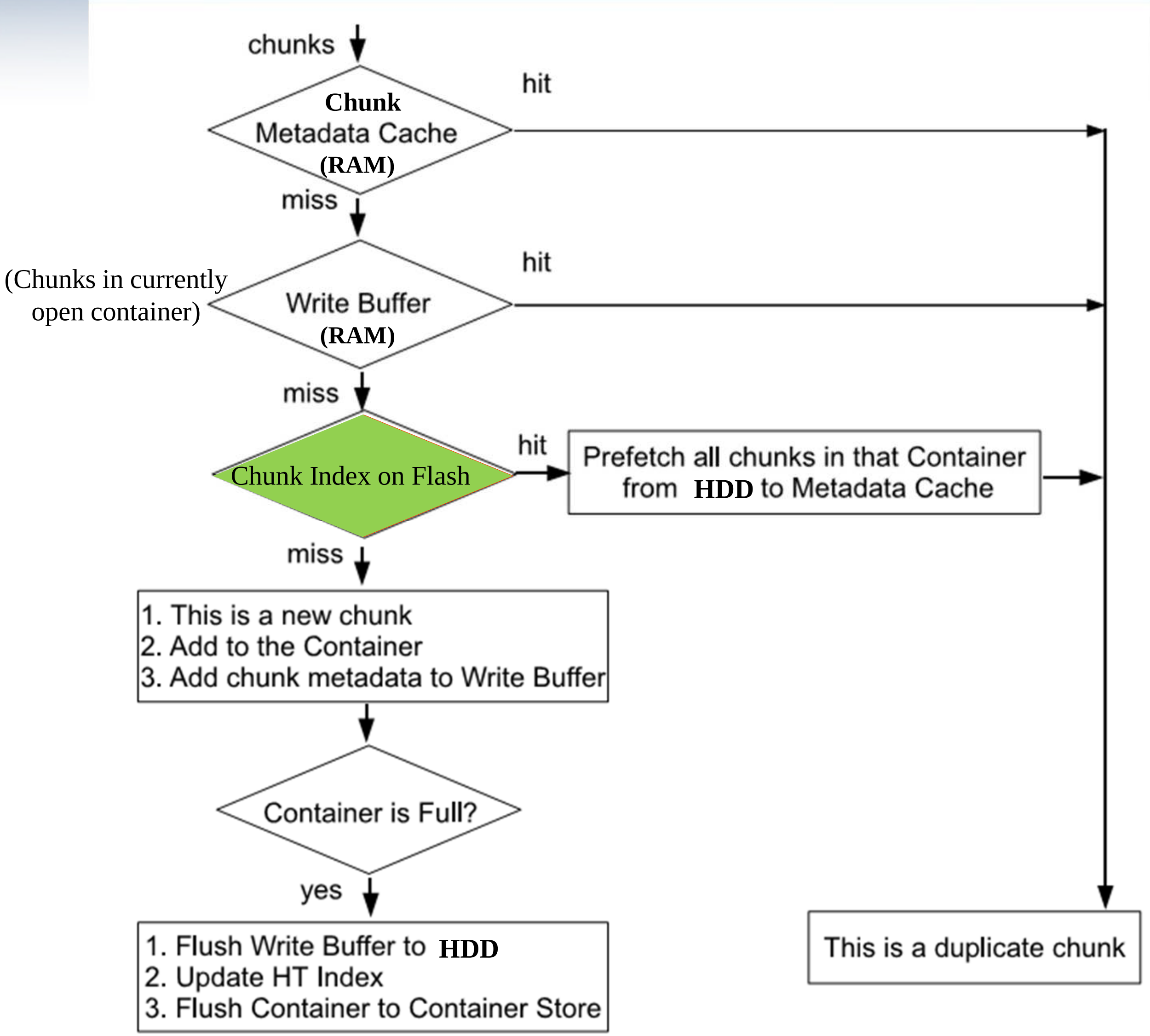
- Chunk hash index for identifying duplicate chunks
  - Key = 20-byte SHA-1 hash (or, 32-byte SHA-256)
  - Value = chunk metadata, e.g., length, location on disk
  - Key + Value ➔ 64 bytes
- Essential Operations
  - Lookup (Get)
  - Insert (Set)
- Need a high performance indexing scheme
  - Chunk metadata too big to fit in RAM
  - Disk IOPS is a bottleneck for disk-based index
  - Duplicate chunk detection bottlenecked by hard disk seek times (~10 msec)

Disk Bottleneck for Identifying Duplicate Chunks

- 20 TB of unique data, average 8 KB chunk size
  - 160 GB of storage for full index (2.5×10^9 unique chunks @64 bytes per chunk metadata)
- Not cost effective to keep all of this huge index in RAM
- Backup throughput limited by disk seek times for index lookups
  - 10ms seek time => 100 chunk lookups per second
    => 800 KB/sec backup throughput
  - No locality in the key space for chunk hash lookups
  - Prefetching into RAM index mappings for entire container to exploit sequential predictability of lookups during 2nd and subsequent full backups (Zhu et al., FAST 2008)



Storage Deduplication Process Schematic



Speedup Potential of a Flash based Index

- RAM hit ratio of 99% (using chunk metadata prefetching techniques)
- Average lookup time with on-disk index

$$t_r + (1 - h_r) * t_d = 1\mu\text{sec} + 0.01 * 10\text{msec} = 101\mu\text{sec}$$

- Average lookup time with on-flash index

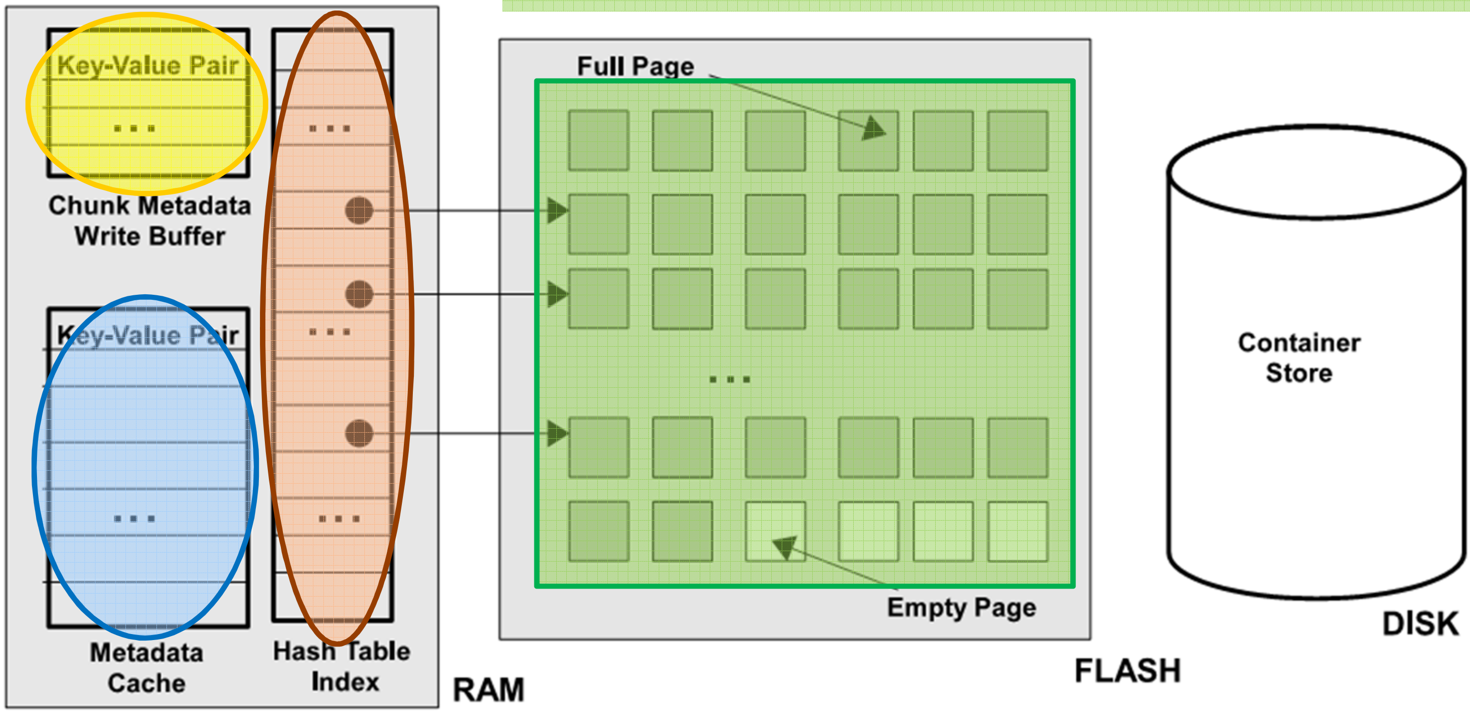
$$t_r + (1 - h_r) * t_f = 1\mu\text{sec} + 0.01 * 100\mu\text{sec} = 2\mu\text{sec}$$

- *Potential of up to 50x speedup with index lookups served from flash*

ChunkStash: Chunk Metadata Store on Flash

RAM write buffer for chunk mappings in currently open container

Chunk metadata organized on flash in log-structured manner in groups of 1023 chunks => 64 KB logical page (@64-byte metadata/ chunk)



Prefetch cache for chunk metadata in RAM for sequential predictability of chunk lookups

Chunk metadata indexed in RAM using a specialized space efficient hash table

Further Reducing RAM Usage in ChunkStash

- Approach 1: Reduce the RAM requirements of the key-value store (work in progress, SkimpyStash)
- Approach 2: Deduplication application specific
  - Index in RAM only a small fraction of the chunks in each container (sample and index every i-th chunk)
    - Flash still holds the metadata for ***all*** chunks in the system
    - Prefetch full container metadata into RAM as before
  - Incur some loss in deduplication quality
  - Fraction of chunks indexed is a powerful knob for tradeoff between RAM usage and dedup quality
    - Index 10% chunks => 90% reduction in RAM usage => less than 1-byte of RAM usage per chunk metadata stored on flash
    - And negligible loss in dedup quality!

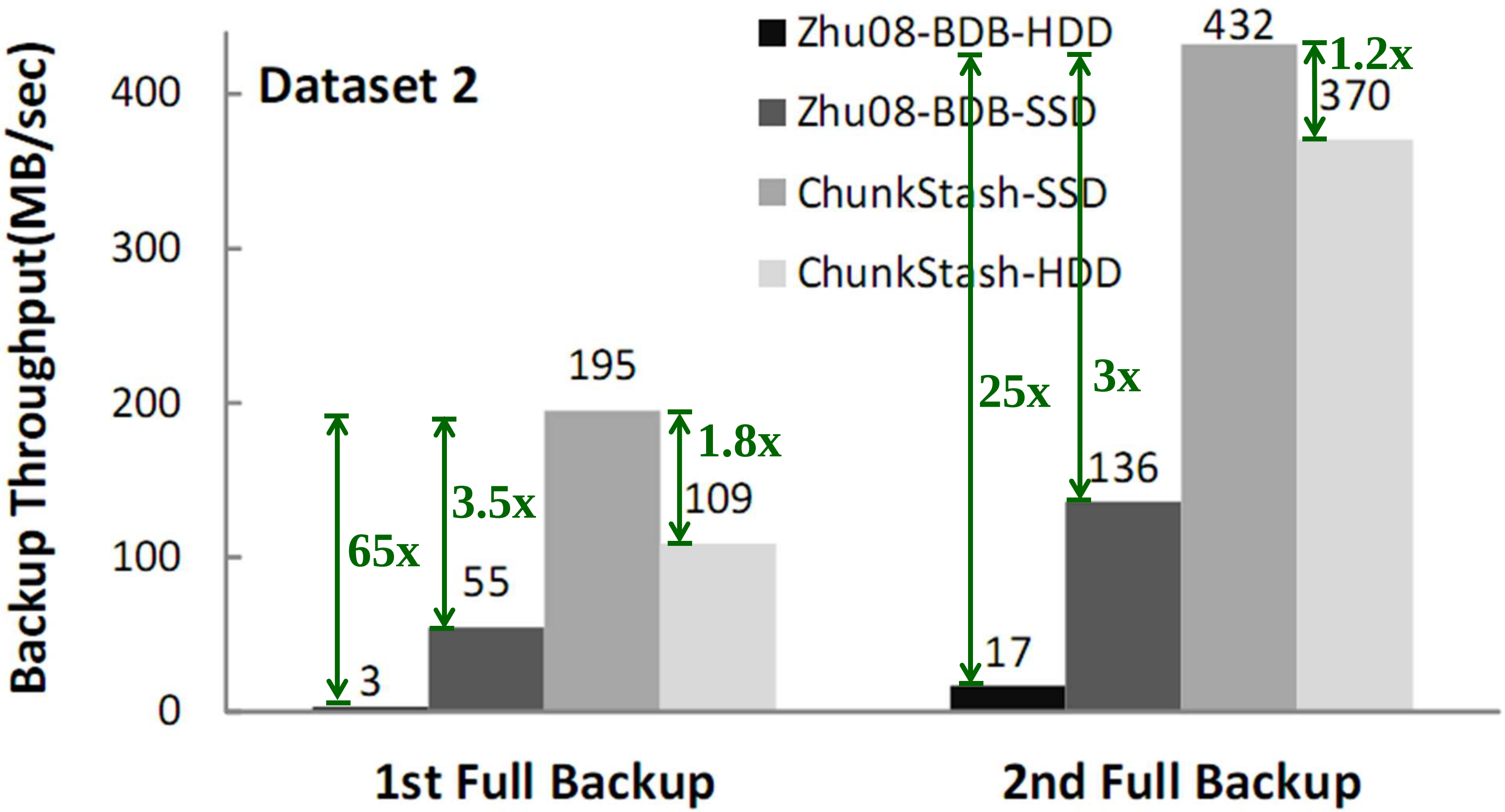
Performance Evaluation

- Comparison with disk index based system
  - Disk based index (Zhu08-BDB-HDD)
  - SSD replacement (Zhu08-BDB-SSD)

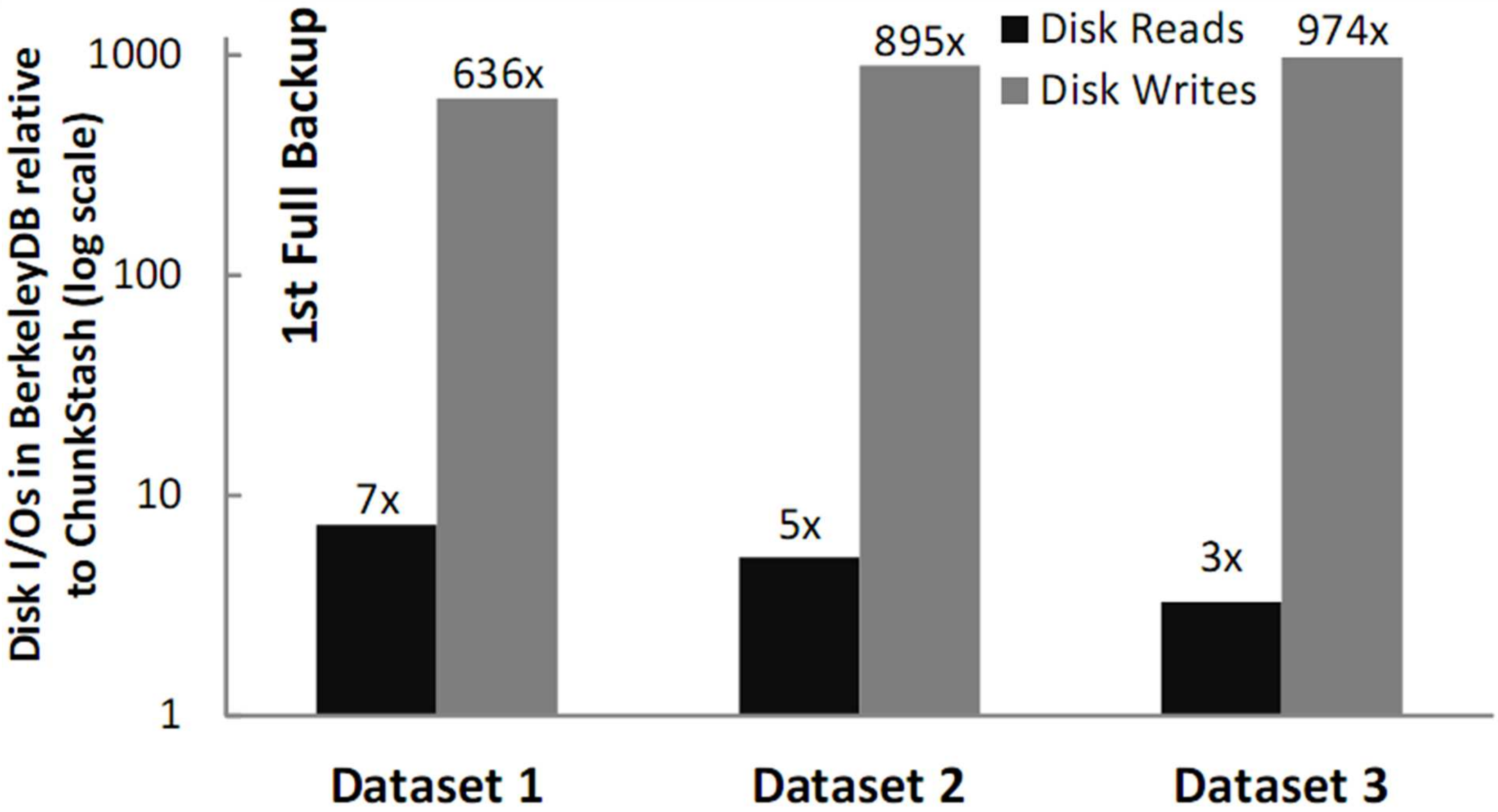
  BerkeleyDB used as the index on HDD/SSD
  - SSD replacement + ChunkStash (ChunkStash-SSD)
  - ChunkStash on hard disk (ChunkStash-HDD)
- Prefetching of chunk metadata in all systems
- Three datasets, 2 full backups for each

Trace	Size (GB)	Total Chunks	#Full Backups
Dataset 1	8GB	1.1 million	2
Dataset 2	32GB	4.1 million	2
Dataset 3	126GB	15.4 million	2

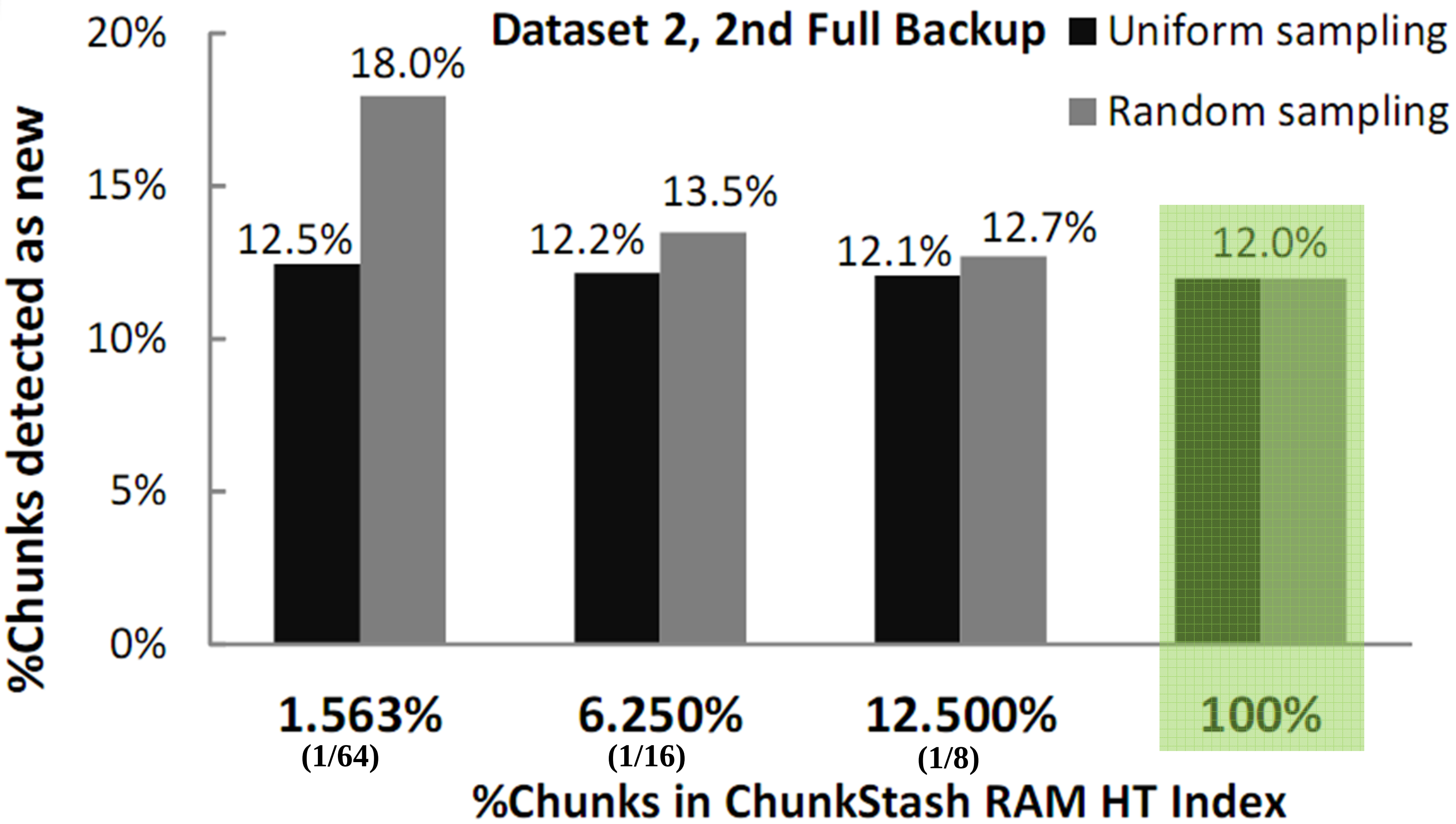
Performance Evaluation – Dataset 2



Performance Evaluation – Disk IOPS



Indexing Chunk Samples in ChunkStash: Deduplication Quality



Flash Memory Cost Considerations

- Chunks occupy an average of 4KB on hard disk
  - Store compressed chunks on hard disk
  - Typical compression ratio of 2:1
- Flash storage is 1/64-th of hard disk storage
  - 64-byte metadata on flash per 4KB occupied space on hard disk
- Flash investment is about 16% of hard disk cost
  - 1/64-th additional storage @10x/GB cost = 16% additional cost
- Performance/dollar improvement of 22x
  - 25x performance at 1.16x cost
- Further cost reduction by amortizing flash across datasets
  - Store chunk metadata on HDD and preload to flash

Summary

- Backup throughput in inline deduplication systems limited by chunk hash index lookups
- Flash-assisted storage deduplication system
  - Chunk metadata store on flash
  - Flash aware data structures and algorithms
  - Low RAM footprint
- Significant backup throughput improvements
  - 7x-60x over over HDD index based system (BerkeleyDB)
  - 2x-4x over flash index based (but flash unaware) system (BerkeleyDB)
  - Performance/dollar improvement of 22x (over HDD index)
- Reduce RAM usage further by 90-99%
  - Index small fraction of chunks in each container
  - Negligible to marginal loss in deduplication quality

Thank You!

Email: sudipta@microsoft.com