ChunkStash: Speeding Up Storage Deduplication using Flash Memory

Biplob Debnath[+], Sudipta Sengupta[*], Jin Li[*]

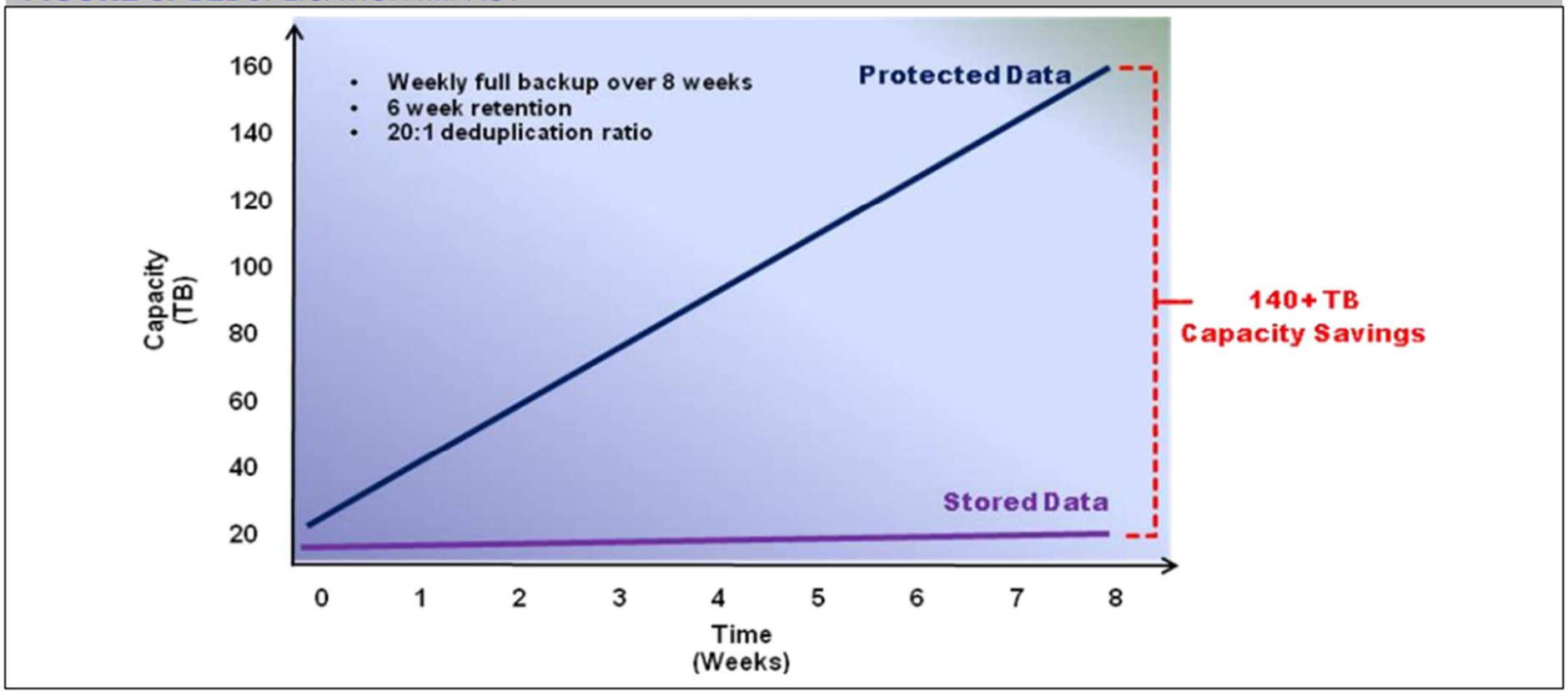
[*]Microsoft Research, Redmond (USA)
[+]Univ. of Minnesota, Twin Cities (USA)

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

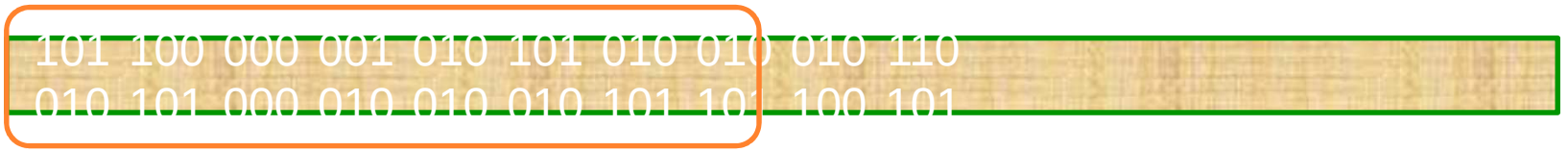
- Detect and remove duplicate data in storage systems
  - e.g., Across full backups
  - Storage space savings
  - Faster backup completion: Disk I/O and Network bandwidth savings
- Feature offering in many storage systems products
  - Data Domain, EMC, NetApp
- Backups need to complete over windows of 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)

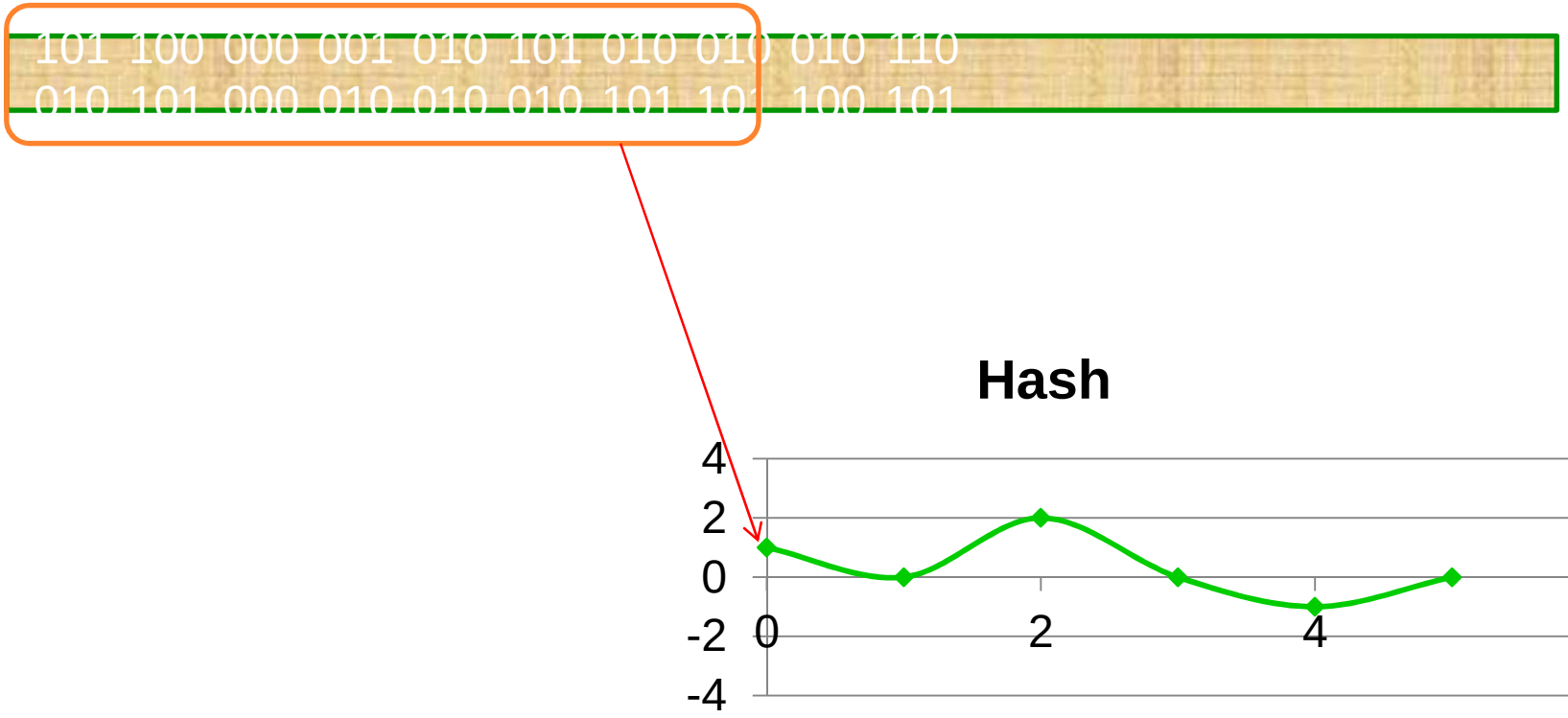
Content based Chunking

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



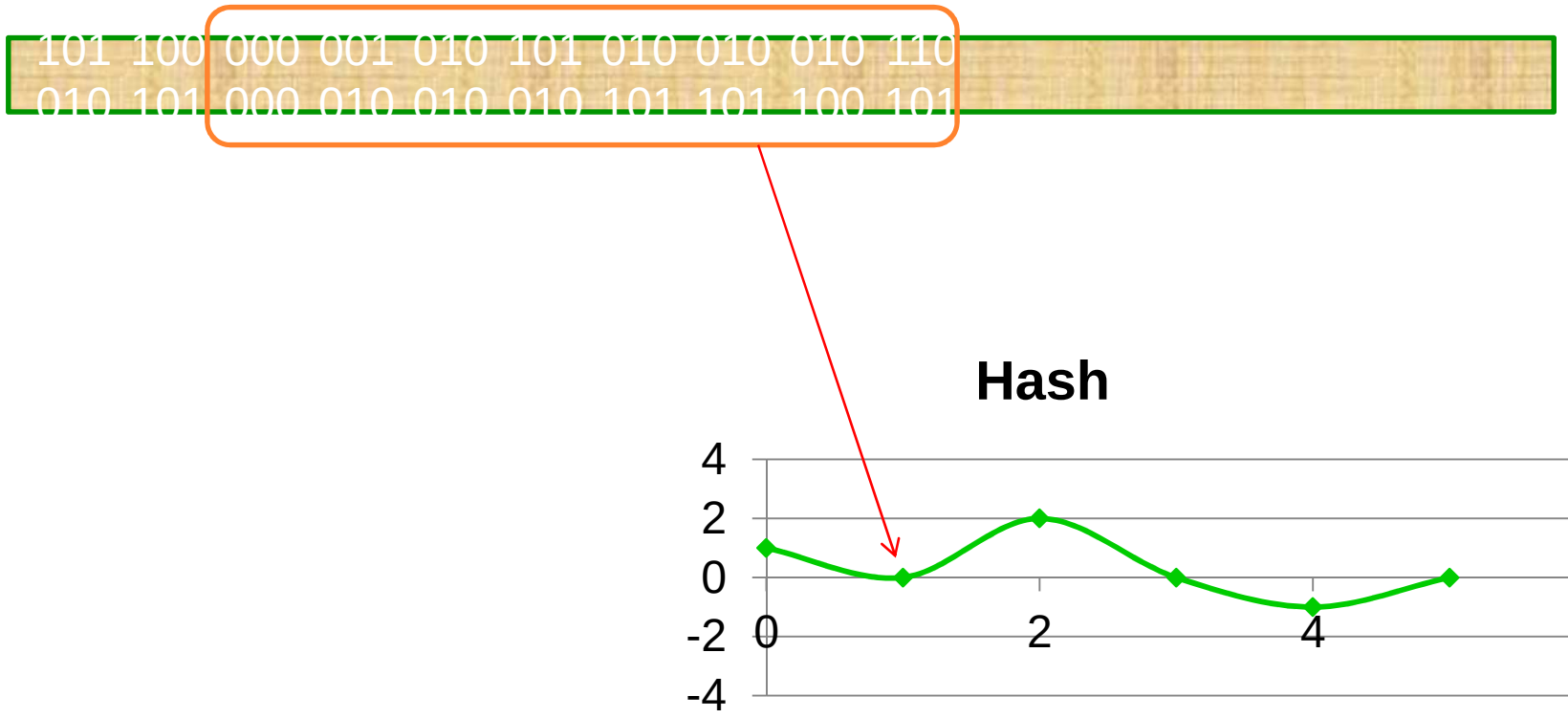
Content based Chunking

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



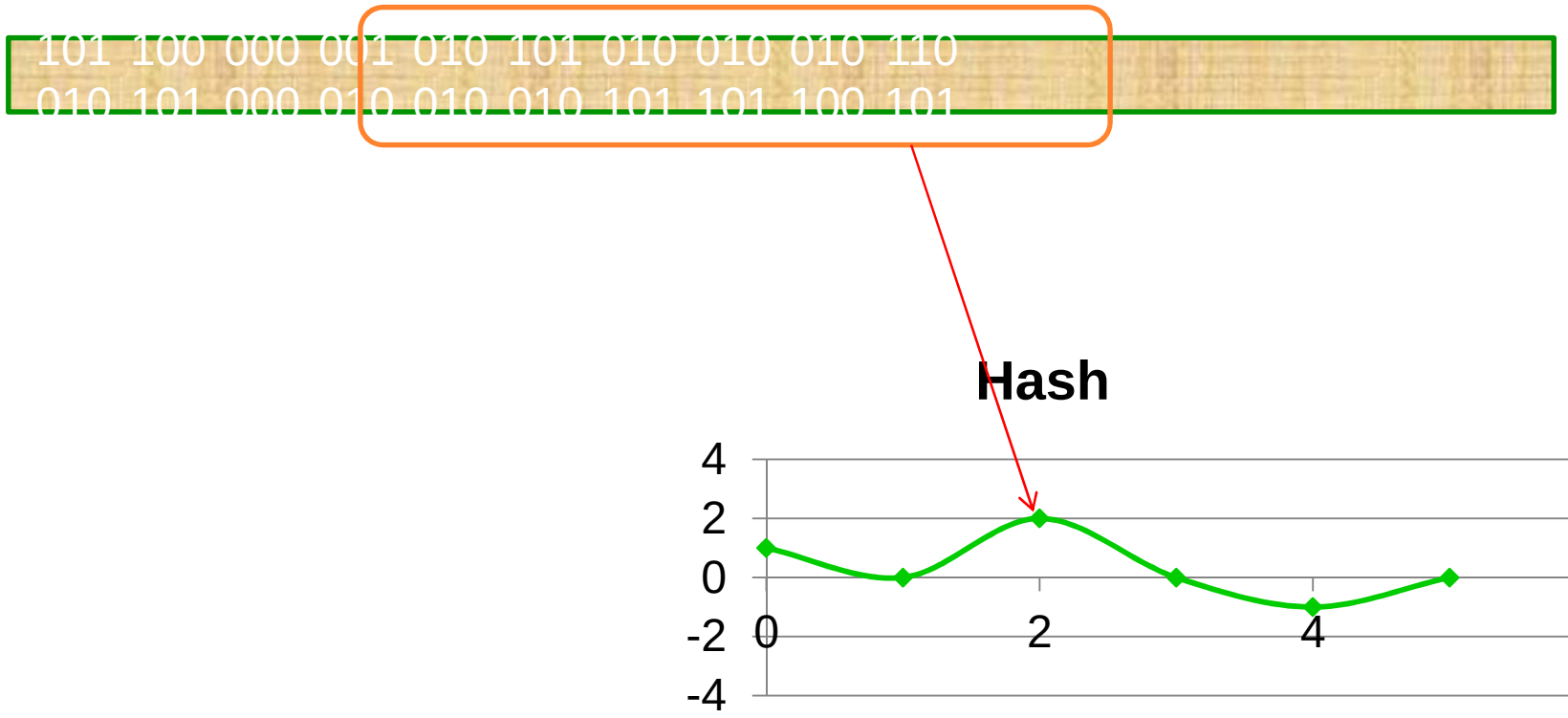
Content based Chunking

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



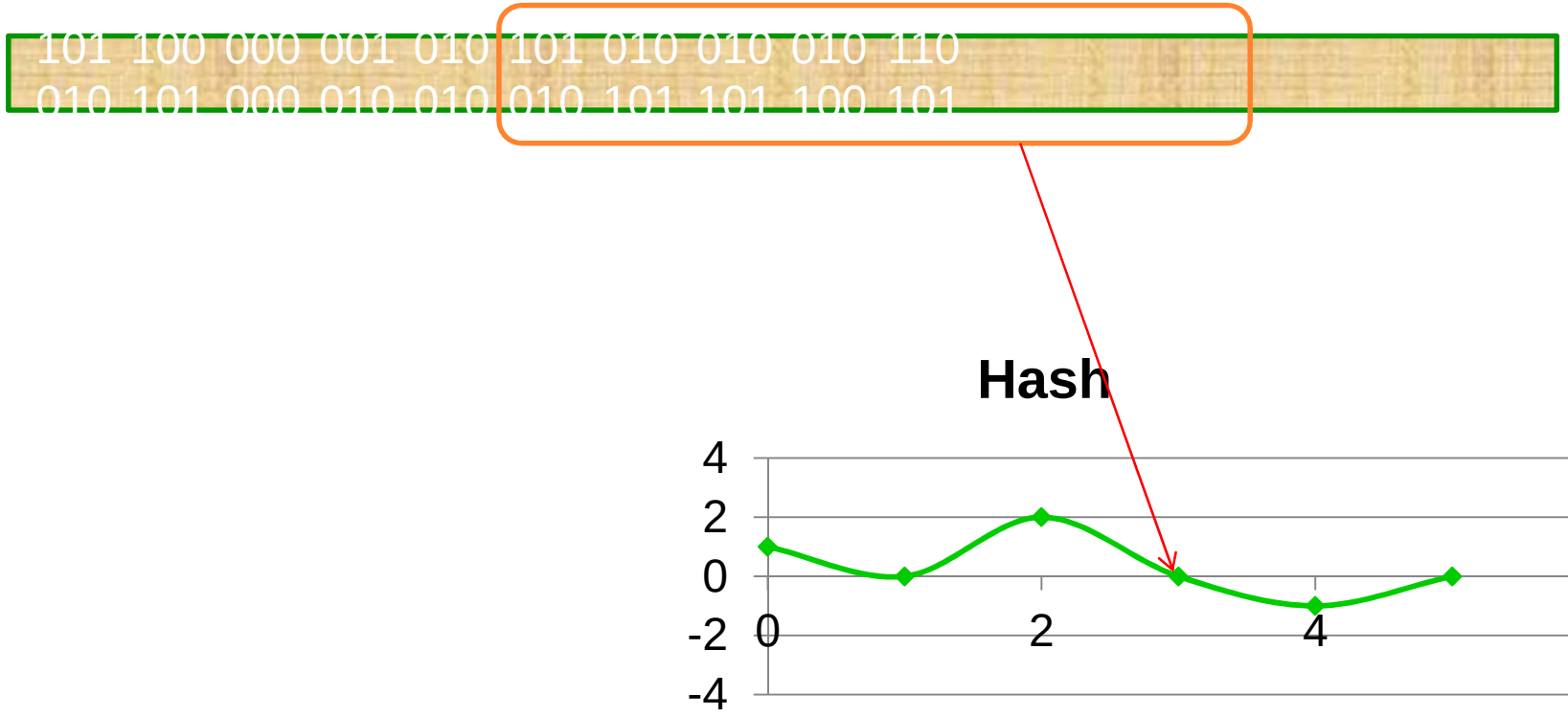
Content based Chunking

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



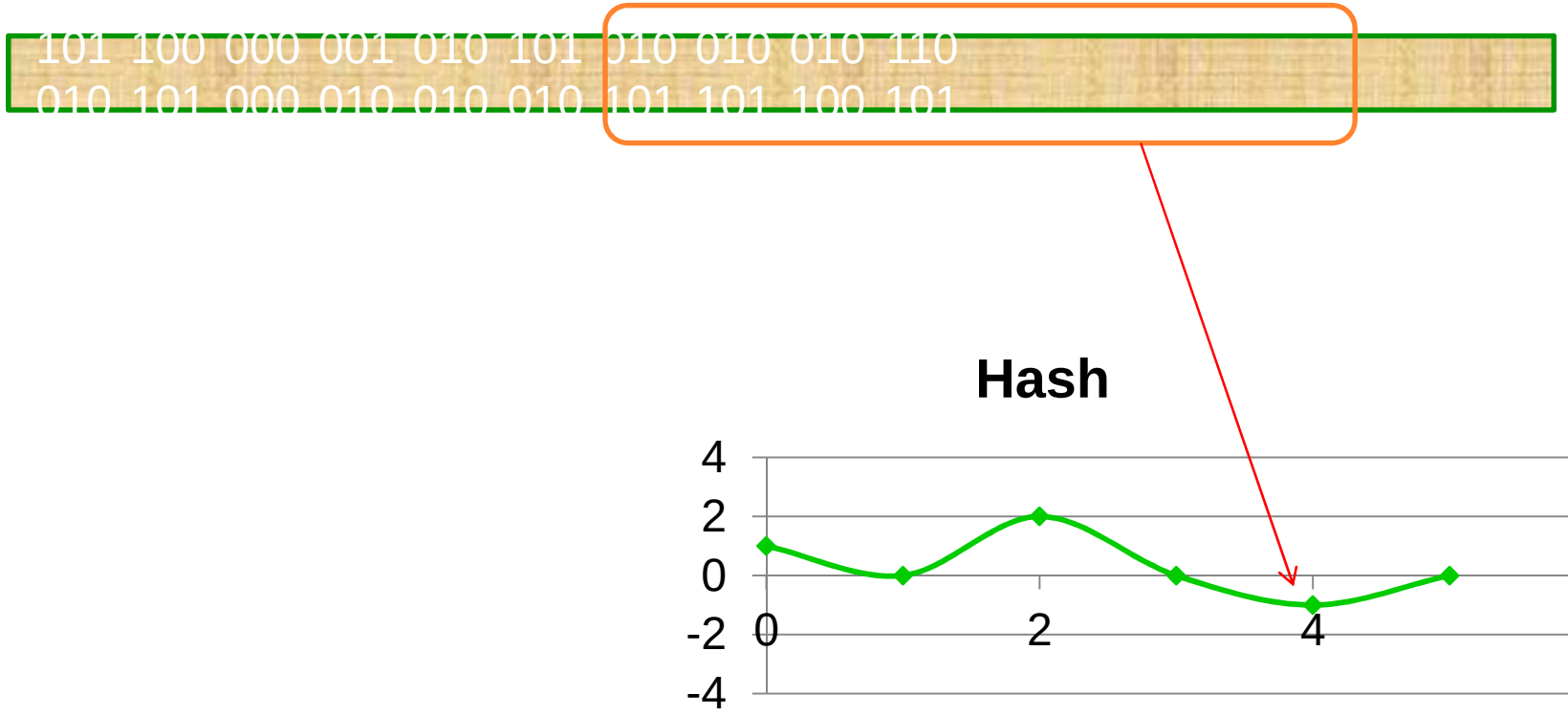
Content based Chunking

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



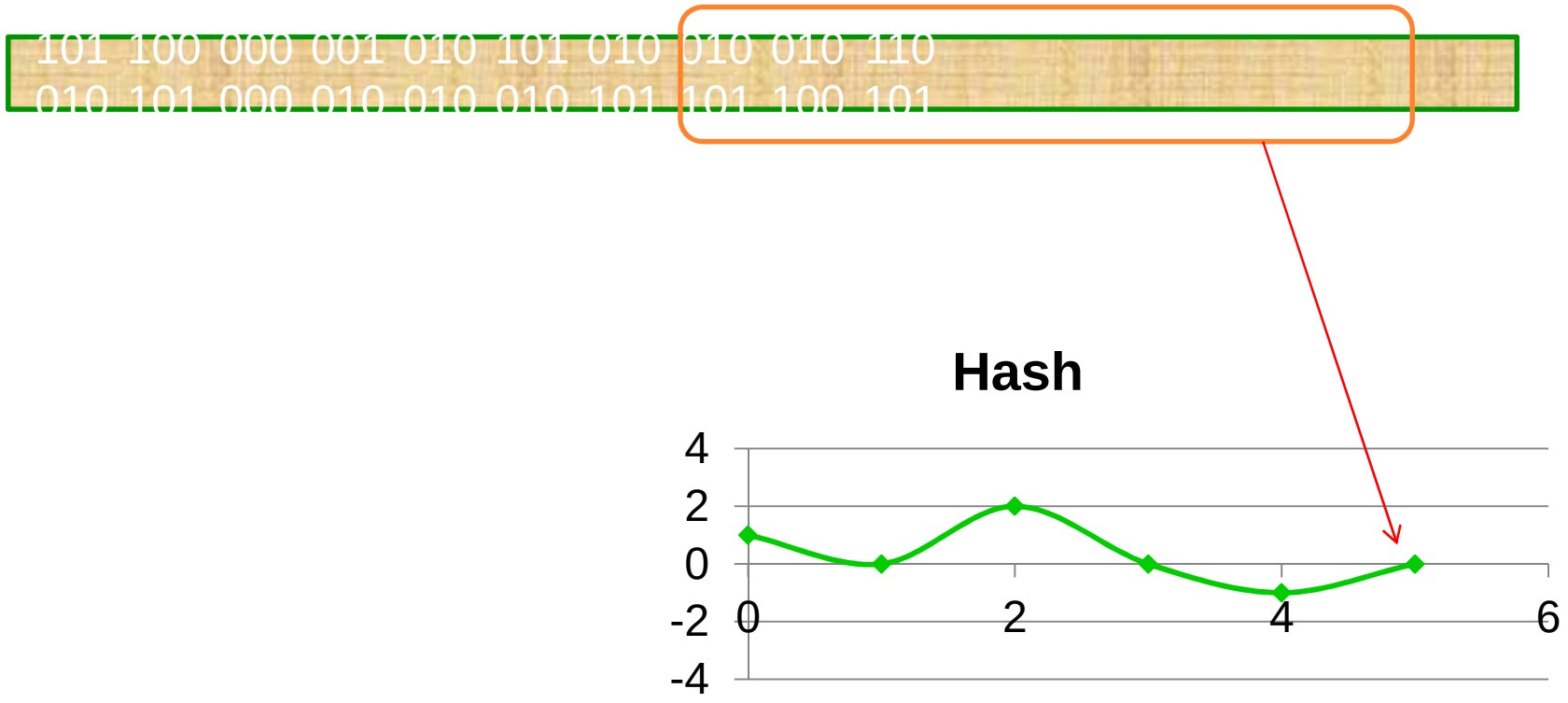
Content based Chunking

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



Content based Chunking

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



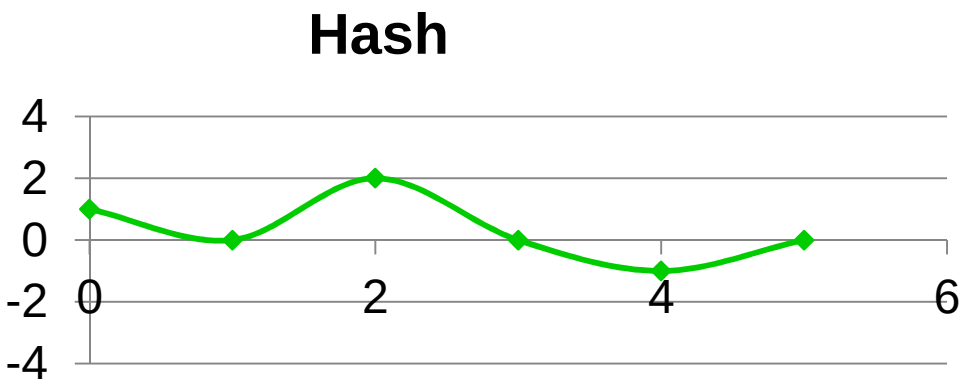
Content based Chunking

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

101 100 000 001 010 101 010 010 010 110
010 101 000 010 010 010 101 101 100 101

If Hash matches a particular pattern,

Declare a chunk boundary

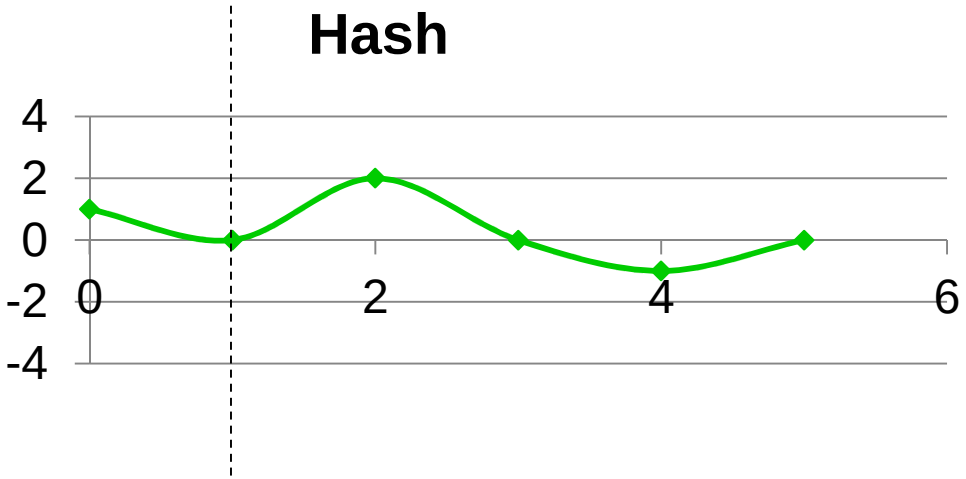


Content based Chunking

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

If Hash matches a particular pattern,

Declare a chunk boundary

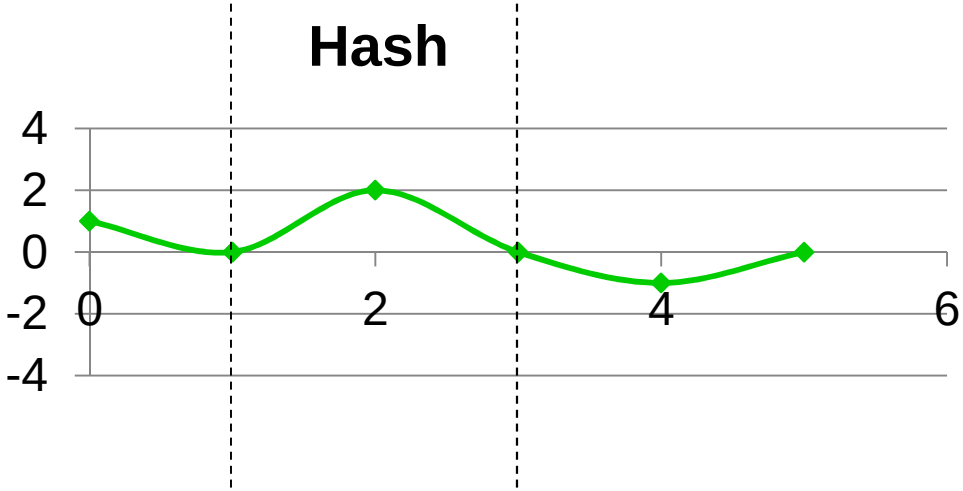


Content based Chunking

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

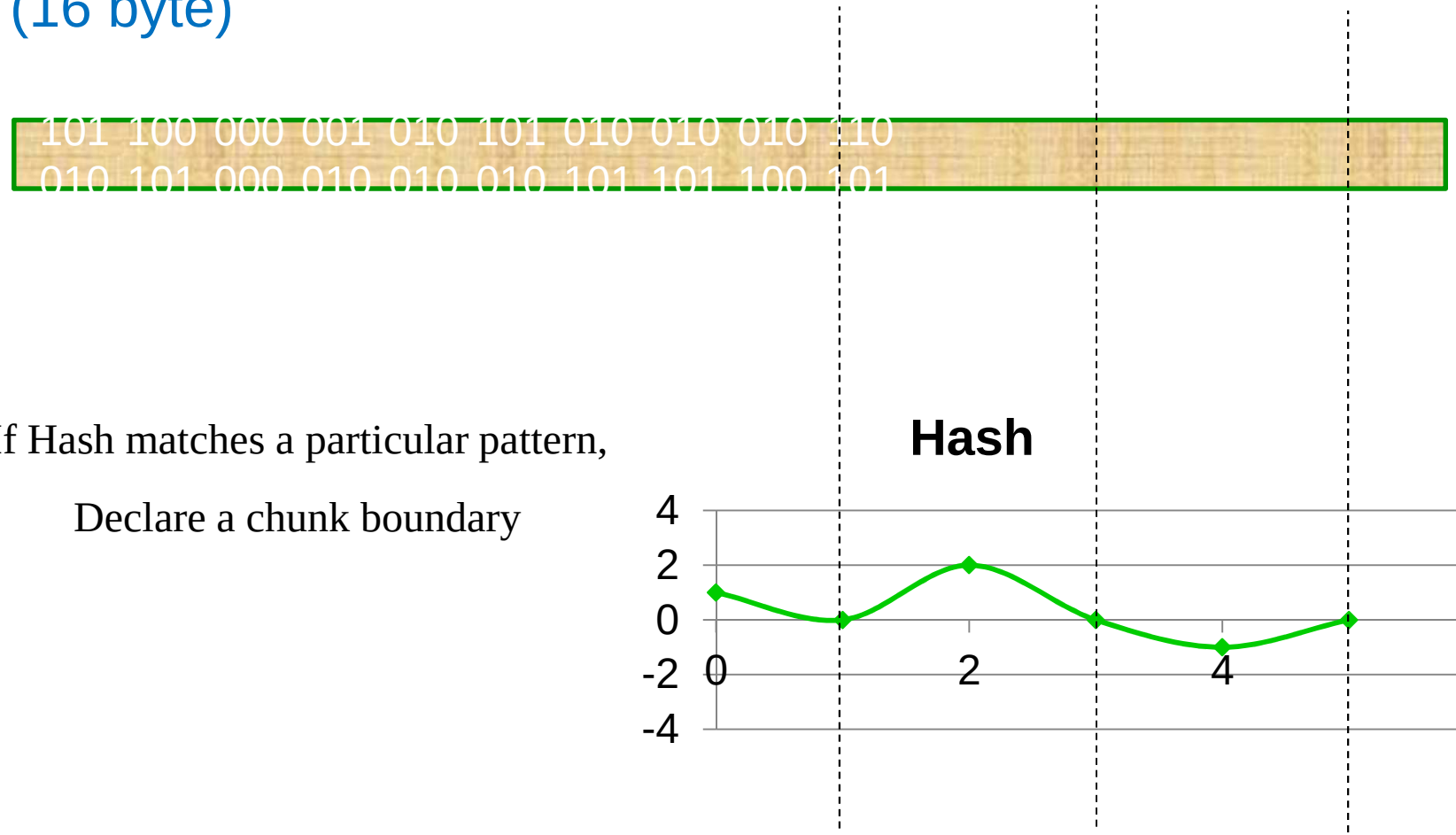
If Hash matches a particular pattern,

Declare a chunk boundary



Content based Chunking

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



If Hash matches a particular pattern,

Declare a chunk boundary

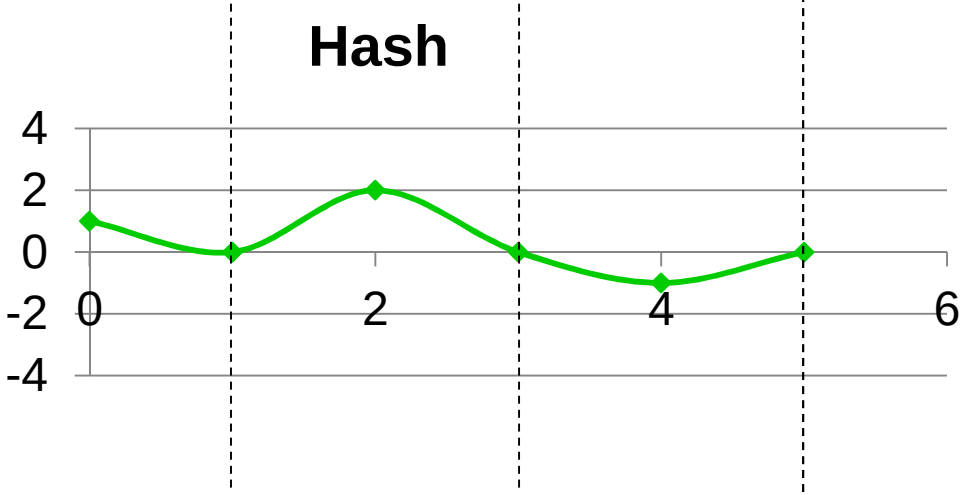
Content based Chunking

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

3 Chunks

If Hash matches a particular pattern,

Declare a chunk boundary

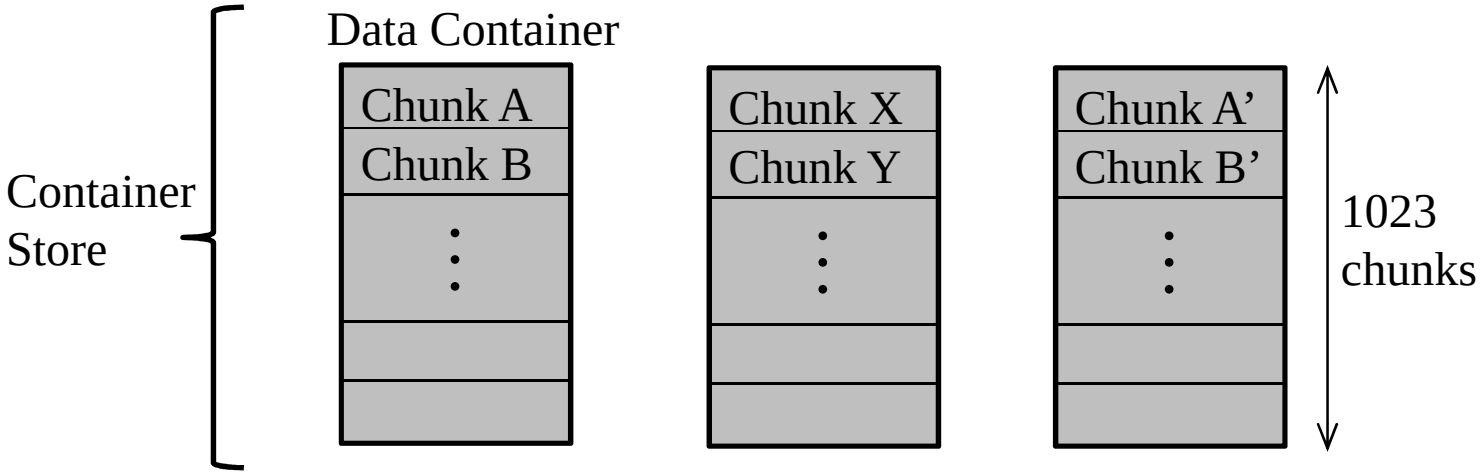


How to Obtain Chunk Boundaries?

- 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

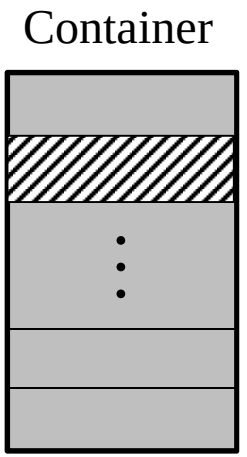


Index for Detecting Duplicate Chunks

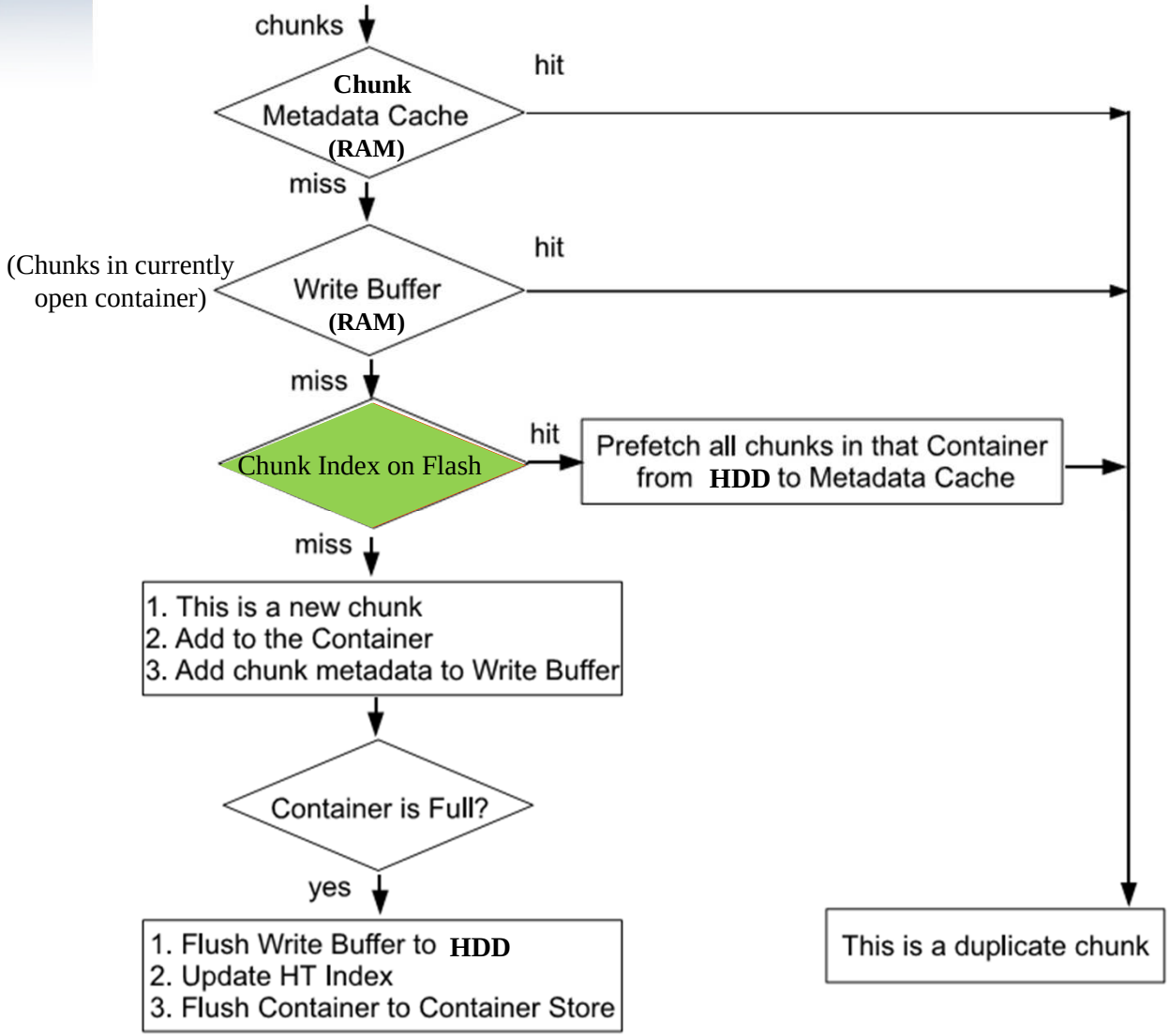
- Chunk hash index for identifying duplicate chunks
  - Key = 20-byte SHA-1 hash (or, 32-byte SHA-256)
  - Value = chunk metadata, e.g., length, location on disk
  - Key + Value ➔ 64 bytes
- Essential Operations
  - Lookup (Get)
  - Insert (Set)
- Need a high performance indexing scheme
  - Chunk metadata 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

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

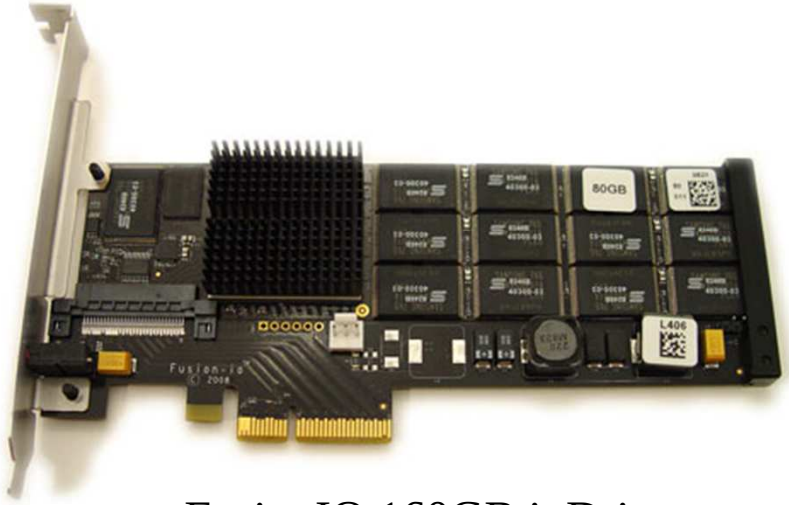
- Average lookup time with on-flash index

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

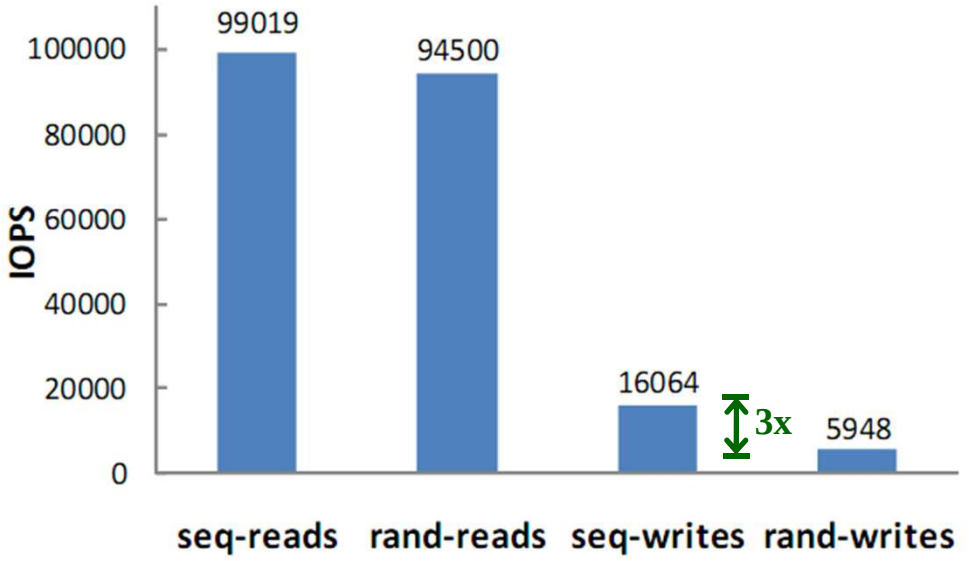
- *Potential of up to 50x speedup with index lookups served from flash*

ChunkStash: Chunk Metadata Store on Flash

- Flash aware data structures and algorithms
  - Random writes, in-place updates are expensive on flash memory
  - Sequential writes, Random/Sequential reads great!
  - Use flash in a log-structured manner
- Low RAM footprint
  - Order of few bytes in RAM for each key-value pair stored on flash



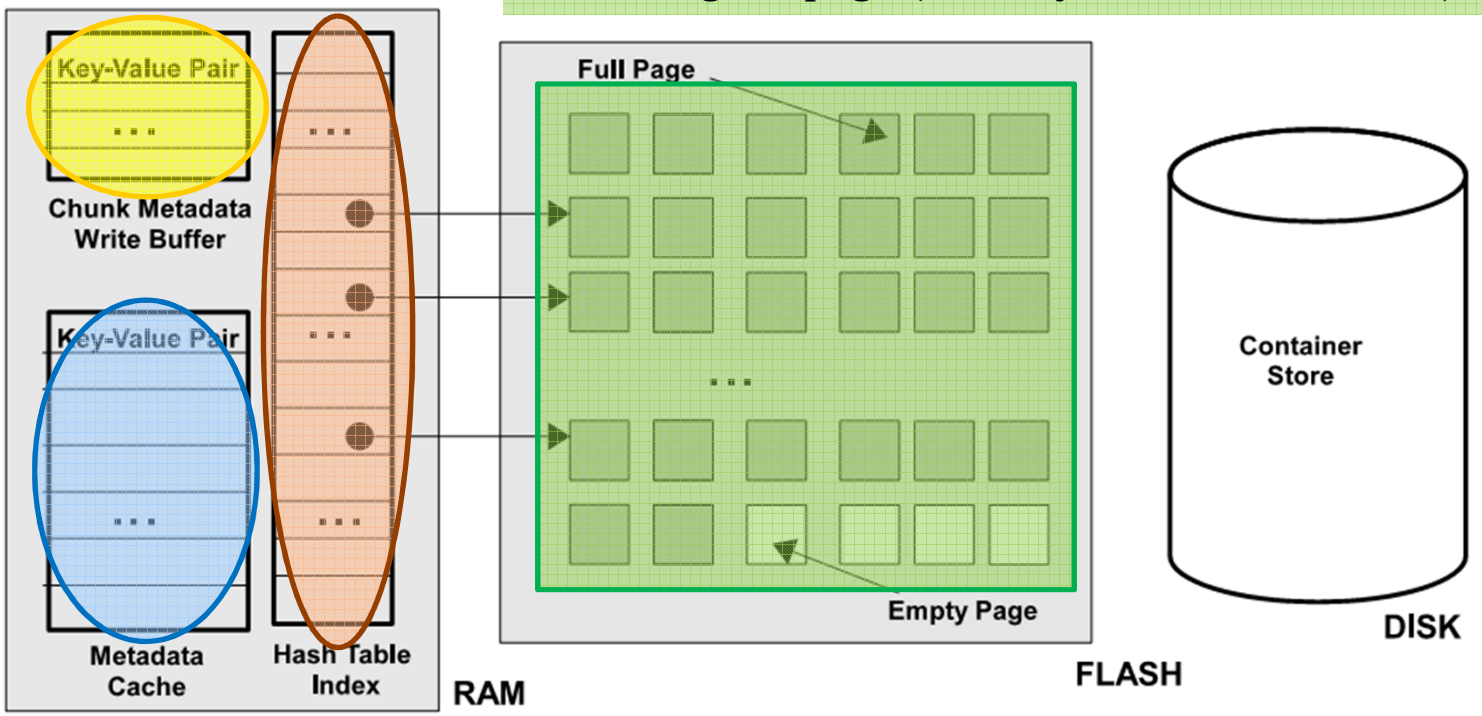
FusionIO 160GB ioDrive



ChunkStash Architecture

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)

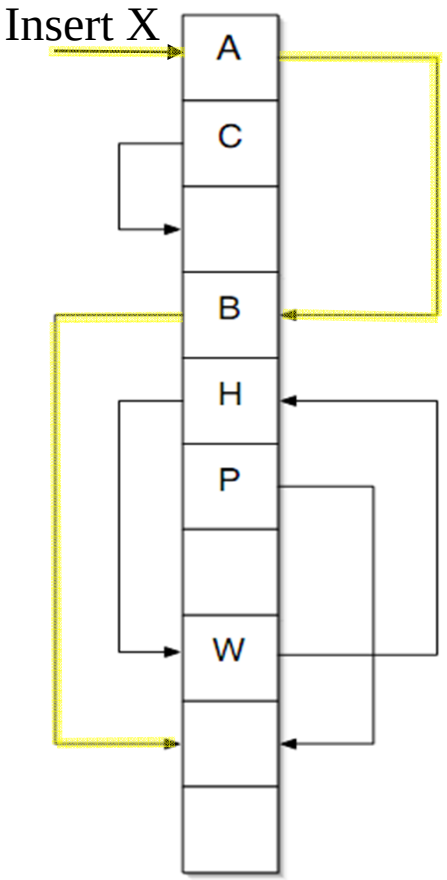


Prefetch cache for chunk metadata in RAM for sequential predictability of chunk lookups

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

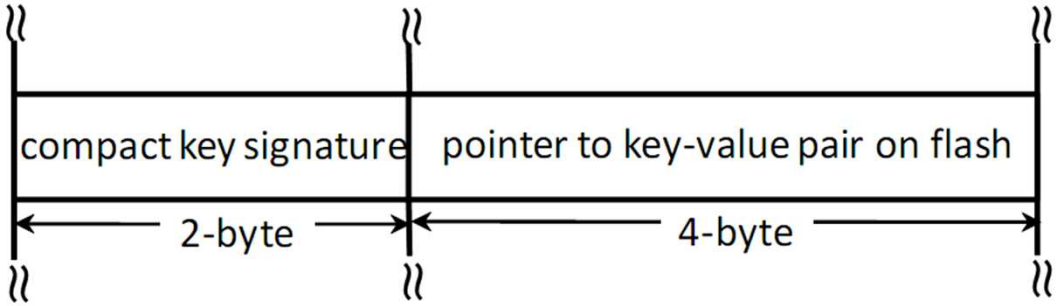
Low RAM Usage: Cuckoo Hashing

- 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. 20-byte SHA-1 hash)
  - 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 of flash size becomes bottleneck for store capacity depends on key-value size
  - At 64 bytes per key-value pair, RAM is the bottleneck
- Example 4GB of RAM
  - 716 million key-value pairs (chunks) @6 bytes of RAM per entry
  - At 8KB average chunk size, this corresponds to 6TB of deduplicated data
  - At 64 bytes of metadata per chunk on flash, this uses 45GB of flash
  - Larger chunk sizes => larger datasets for same amount of RAM and flash (but may tradeoff with dedup quality)

Further Reducing RAM Usage in ChunkStash

- Approach 1: Reduce the RAM requirements of the key-value store (work in progress)
- 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 chunk 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!

Compare with Sparse Indexing Scheme

- Sparse indexing scheme (FAST 2009)
  - Chop incoming stream into multi-MB segments, select chunk hooks in each segment using random sampling
  - Use these hooks to find few segments seen in the *recent past* that share many chunks
- How does ChunkStash differ?
  - Uniform interval sampling
  - No concept of segment; all incoming chunks looked up in index
  - Match incoming chunks with sampled chunks in *all containers* stored in the system, not just those see in recent past

Performance Evaluation

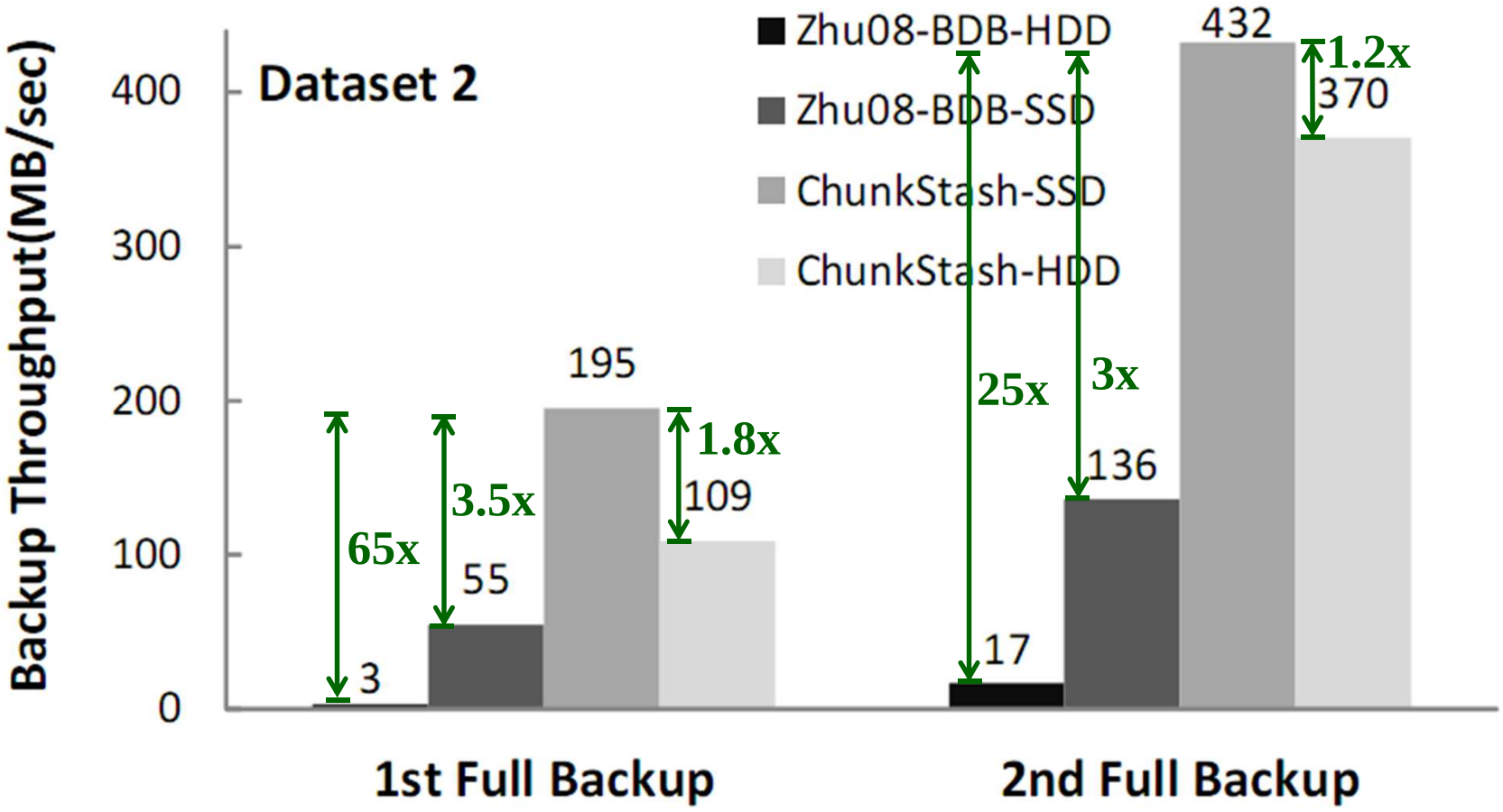
- Comparison with disk index based system
  - Disk based index (Zhu08-BDB-HDD)
  - SSD replacement (Zhu08-BDB-SSD)

  } BerkeleyDB used as the index on HDD/SSD

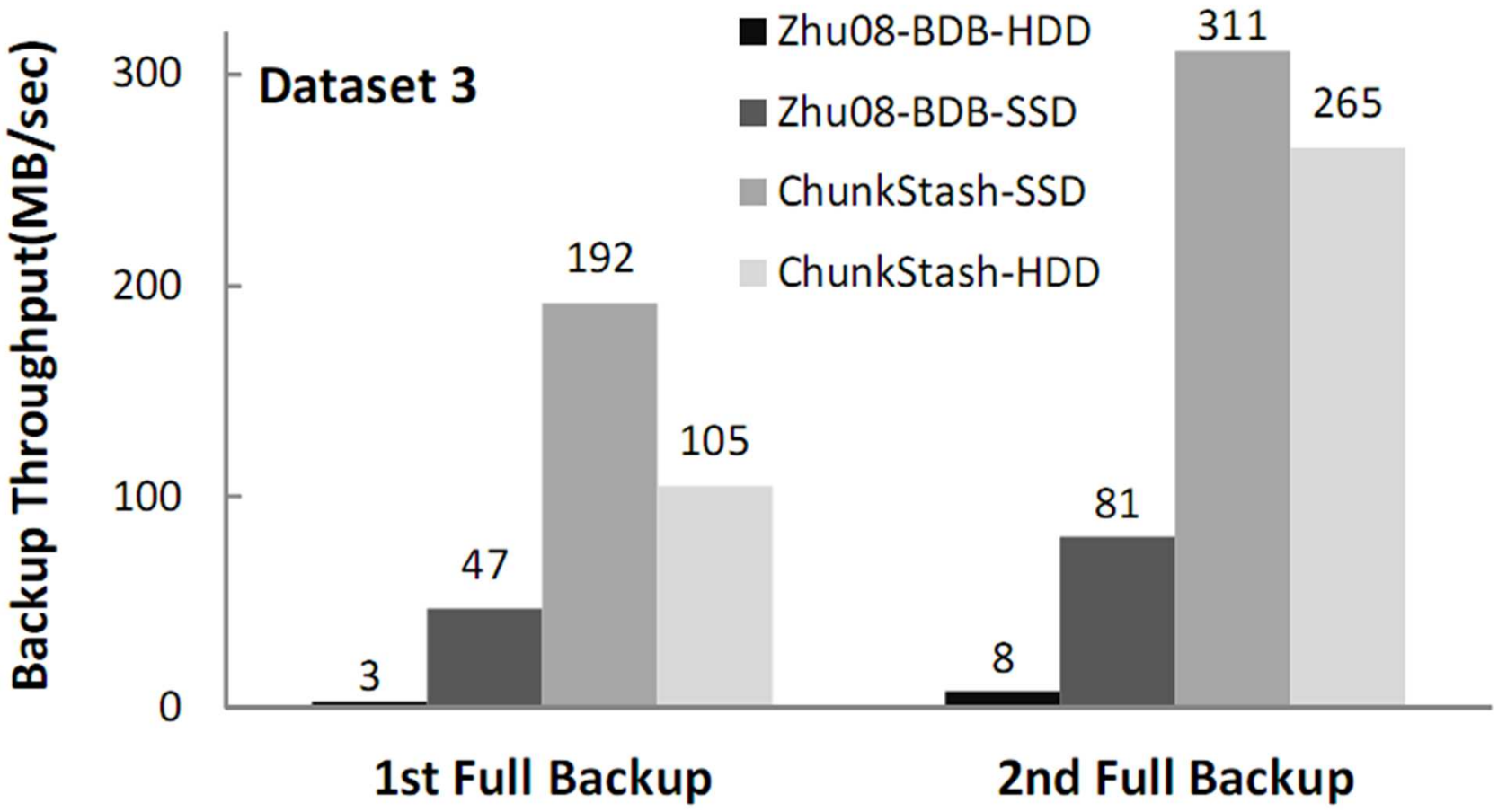
  - SSD replacement + ChunkStash (ChunkStash-SSD)
  - ChunkStash on hard disk (ChunkStash-HDD)
- Prefetching of chunk metadata in all systems
- Three datasets, 2 full backups for each

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

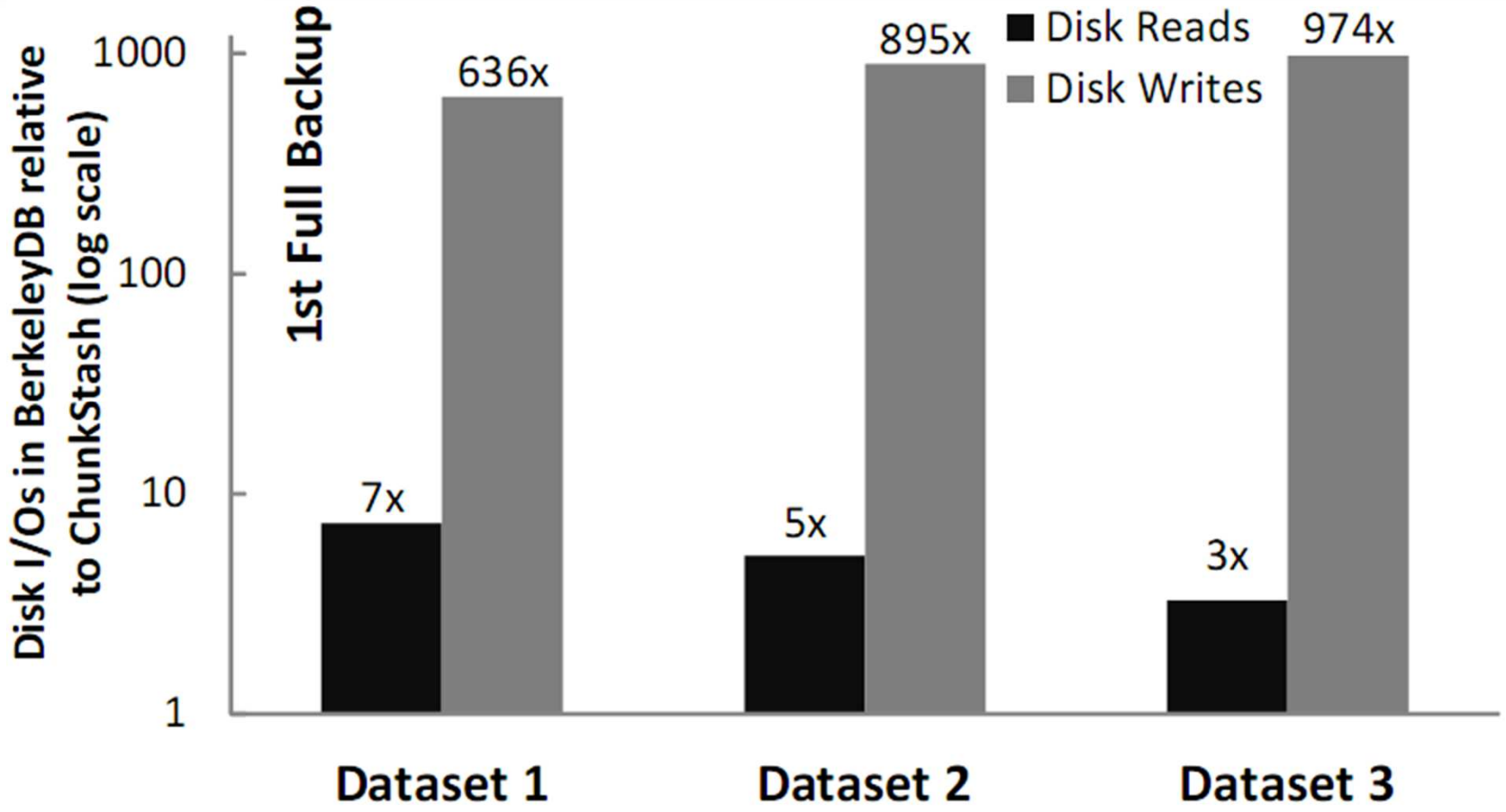
Performance Evaluation – Dataset 2



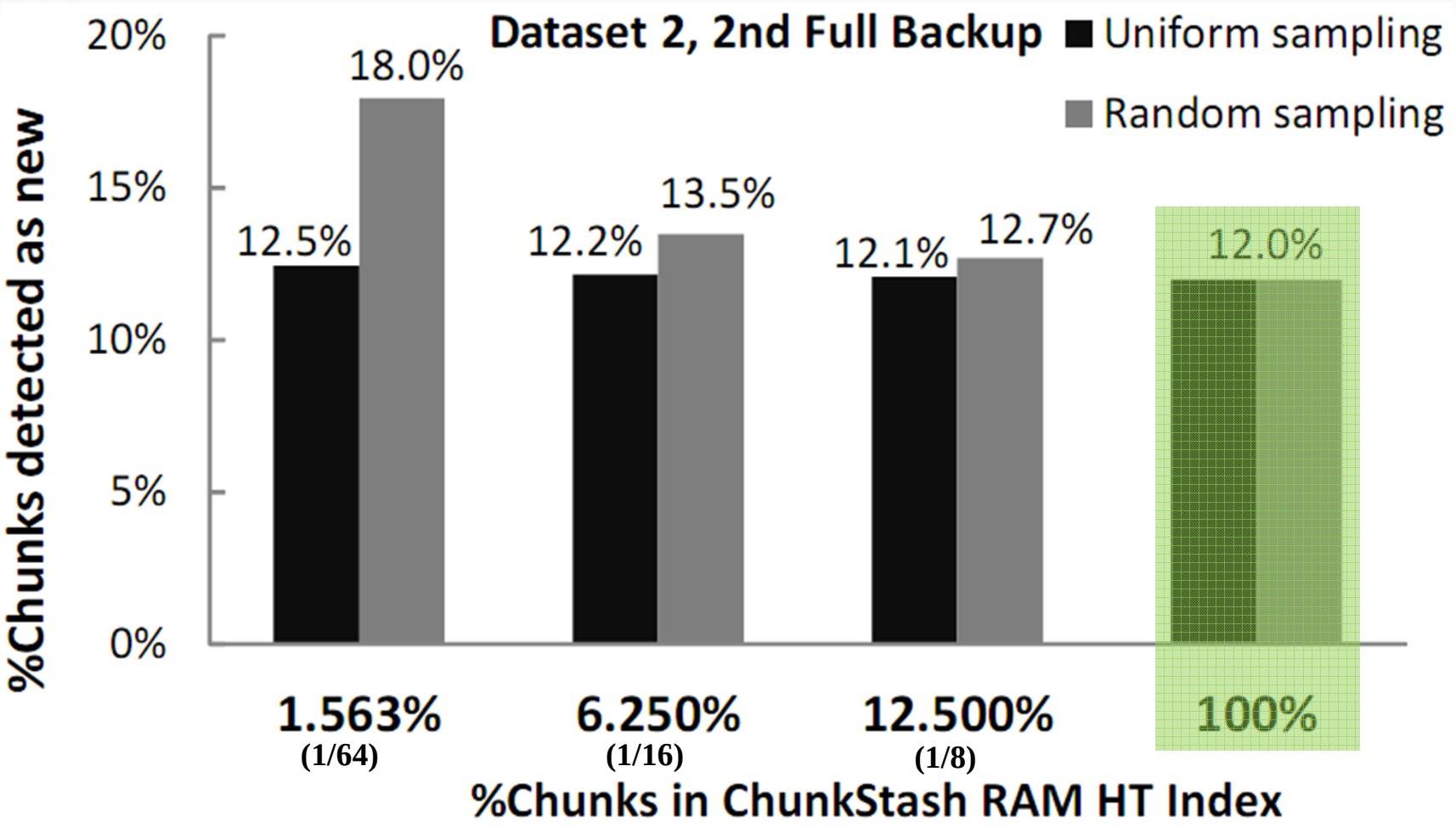
Performance Evaluation – Dataset 3



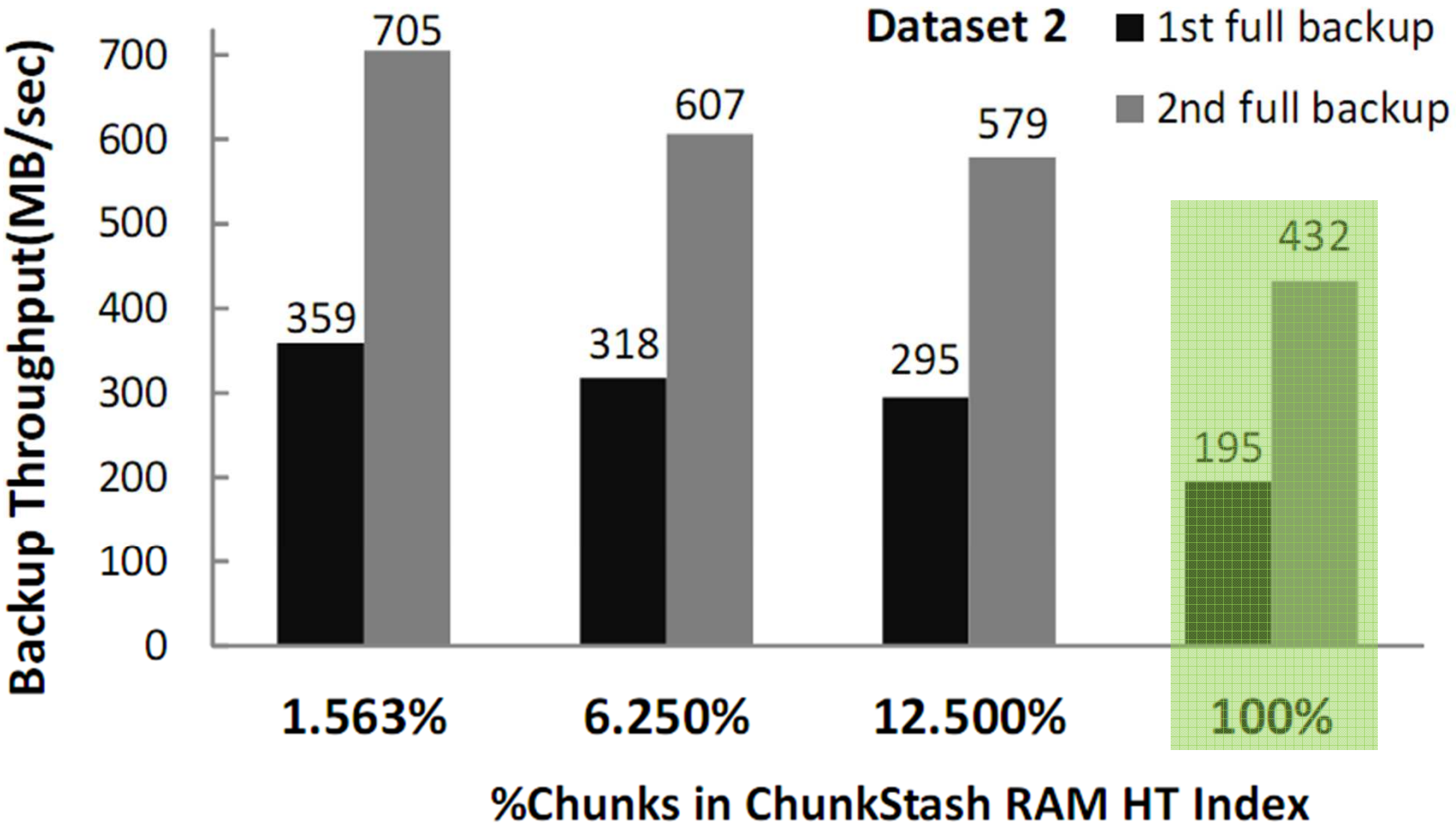
Performance Evaluation – Disk IOPS



Indexing Chunk Samples in ChunkStash: Deduplication Quality



Indexing Chunk Samples in ChunkStash: Backup Throughput



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 occupies 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