Flash Memory Aware Software Architectures and Applications

Sudipta Sengupta and Jin Li
Microsoft Research, Redmond, WA, USA

Contains work that is joint with Biplob Debnath (Univ. of Minnesota)
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 Aware Applications

- **System builders:** Don’t just treat flash as disk replacement
  - Make the OS/application layer aware of flash
  - Exploit its benefits
  - Embrace its peculiarities and design around them
  - Identify applications that can exploit sweet spot between cost and performance

- **Device vendors:** You can help by exposing more APIs to the software layer for managing storage on flash
  - Can help to squeeze better performance out of flash with application knowledge
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]**
  - Efficient index design on flash for high throughput data deduplication

- **BloomFlash [ICDCS 2011]**
  - Bloom filter design for flash

- **SkimpyStash [ACM SIGMOD 2011]**
  - 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
  - SSD buffer pool extension in database server
  - SSD caching tier in cloud storage
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
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

RAM read cache for recently accessed key-value pairs

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

Recently unused key-value pairs destaged to HDD
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 $h_1(x), \ldots, h_K(x)$
  - In practice, two hash functions can simulate $K$ hash functions using $h_i(x) = g_1(x) + i\cdot g_2(x)$

- 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
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)

<table>
<thead>
<tr>
<th>CPU Type</th>
<th>Power</th>
<th>RAM Size</th>
<th>Flash (SSD) Type</th>
<th>Cost</th>
<th>Hard Disk (HDD) Type</th>
<th>Cost</th>
<th>Power</th>
<th>Chassis Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Core 2 Duo E8500 @3.16GHz</td>
<td>65W</td>
<td>4GB</td>
<td>fusionIO 80GB</td>
<td>$2200</td>
<td>Seagate Barracuda 250GB 7200rpm</td>
<td>$50</td>
<td>12W</td>
<td>$1150</td>
</tr>
</tbody>
</table>

- **Trace**
  - Xbox LIVE Primetime
  - Storage Deduplication

<table>
<thead>
<tr>
<th>Trace</th>
<th>Total get-set ops</th>
<th>get:set ratio</th>
<th>Avg. size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xbox</td>
<td>5.5 million</td>
<td>7.5:1</td>
<td>92 1200</td>
</tr>
<tr>
<td>Dedup</td>
<td>40 million</td>
<td>2.2:1</td>
<td>20 44</td>
</tr>
</tbody>
</table>
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)

- FlashStore: 42527 (5x) for Blue, 57271 (8x) for Dedup trace
- BerkeleyDB-SSD: 8395 (60x) for Blue, 7325 (24x) for Dedup trace
- BerkeleyDB-HDD: 700 for Blue, 2415 for Dedup trace
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
SkimpyStash: Ultra-Low RAM Footprint Key-Value Store on Flash
Aggressive Design Goal for RAM Usage

- Target ~1 byte of RAM usage per key-value pair on flash
  - Tradeoff with key access time (#flash reads per lookup)
- Preserve log-structured storage organization on flash
SkimpyStash: Base Design

- Resolve hash table collisions using linear chaining
  - Multiple keys resolving to a given hash table bucket are chained in a linked list
- Storing the linked lists on flash itself
  - Preserve log-structured organization with later inserted keys pointing to earlier keys in the log
  - Each hash table bucket in RAM contains a pointer to the beginning of the linked list on flash
Hash table directory

RAM

Flash Memory

Sequential log

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
<th>null</th>
</tr>
</thead>
<tbody>
<tr>
<td>key</td>
<td>value</td>
<td>null</td>
</tr>
<tr>
<td>key</td>
<td>value</td>
<td>null</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>key</td>
<td>value</td>
<td></td>
</tr>
<tr>
<td>key</td>
<td>value</td>
<td></td>
</tr>
</tbody>
</table>

Keys ordered by write time in log
Logical pages are formed by linking together records on possibly different physical pages

- Hash buckets do *not* correspond to whole physical pages on flash but to logical pages
- Physical pages on flash contain records from multiple hash buckets

Exploits random access nature of flash media
- No disk-like seek overhead in reading records in a hash bucket spread across multiple physical pages on flash
Base Design: RAM Space Usage

- \( k = \) average \#keys per bucket
  - Critical design parameter
- \( \frac{4}{k} \) bytes of RAM per k-v pair
  - Pointer to chain on flash (4 bytes) per slot
- Example: \( k=10 \)
  - Average of 5 flash reads per lookup = \(~50\) usec
  - 0.5 bytes in RAM per k-v pair on flash
The Tradeoff Curve

![Graph showing the tradeoff between RAM bytes per key-value pair and average keys per bucket (k). The graph compares the base design and enhanced design.]
Base Design: Room for Improvement?

- Large variations in average lookup times across buckets
  - Skewed distribution in number of keys in each bucket chain
- Lookups on non-existing keys
  - Require entire bucket (linked list) to be searched on flash
Improvement Idea 1: Load Balancing across Buckets

- Two-choice based load balancing across buckets
  - Hash each key to two buckets and insert in least-loaded bucket
  - 1-byte counter per bucket
- Lookup times double
  - Need to search both buckets during lookup
  - Fix?
Improvement Idea 2: Bloom Filter per Bucket

- **Bloom Filter per Bucket**
  - Lookup checks BF before searching linked list on flash
  - Sized for ~k keys => k-bytes per hash table directory slot

- **Other benefits**
  - Lookups on non-existing keys faster (almost always no flash access)

- **Benefits from load balancing**
  - Balanced chains help to improve BF accuracy (false positives)
  - Symbiotic relationship!
Enhanced Design: RAM Space Usage

- \( k \) = average number of keys per bucket
- \((1 + 5/k)\) bytes of RAM per k-v pair
  - Pointer to chain on flash (4 bytes)
  - Bucket size (1 byte)
  - Bloom filter (k bytes)

- Example: \( k = 10 \)
  - Average of 5 flash reads per lookup = \( \sim 50 \) usec
  - 1.5 bytes in RAM per k-v pair on flash
Compaction to Improve Read Performance

- When enough records accumulate in a bucket to fill a flash page
  - Place them contiguously on one or more flash pages (m records per page)
  - Average #flash reads per lookup = \( \lceil k/2m \rceil \)

- Garbage created in the log
  - Compaction
  - Updated or deleted records
ChunkStash: Speeding Up Storage Deduplication using Flash Memory
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**

- Weekly full backup over 8 weeks
- 6 week retention
- 20:1 deduplication ratio

Protected Data

140+ TB
Capacity Savings

Stored Data

Source: Data Domain white paper
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)

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)

If Hash matches a particular pattern,

Declare a chunk boundary

3 Chunks
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 \(\rightarrow\) 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 ($2.5 \times 10^9$ 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

- chunks
  - Chunk Metadata Cache (RAM)
    - hit
    - Write Buffer (RAM)
      - hit
      - chunk Index on Flash
        - hit
        - Prefetch all chunks in that Container from HDD to Metadata Cache
        - miss
          - 1. This is a new chunk
            2. Add to the Container
            3. Add chunk metadata to Write Buffer
        - Container is Full?
          - yes
            1. Flush Write Buffer to HDD
            2. Update HT Index
            3. Flush Container to Container Store
        - This is a duplicate chunk
ChunkStash: Chunk Metadata Store on Flash

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

RAM write buffer for chunk mappings in currently open container

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
Performance Evaluation

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

<table>
<thead>
<tr>
<th>Trace</th>
<th>Size (GB)</th>
<th>Total Chunks</th>
<th>#Full Backups</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dataset 1</td>
<td>8GB</td>
<td>1.1 million</td>
<td>2</td>
</tr>
<tr>
<td>Dataset 2</td>
<td>32GB</td>
<td>4.1 million</td>
<td>2</td>
</tr>
<tr>
<td>Dataset 3</td>
<td>126GB</td>
<td>15.4 million</td>
<td>2</td>
</tr>
</tbody>
</table>
Performance Evaluation – Dataset 2

**Dataset 2**

Backup Throughput (MB/sec)

- **1st Full Backup**
  - Zhu08-BDB-HDD: 3
  - Zhu08-BDB-SSD: 55
  - ChunkStash-SSD: 109
  - ChunkStash-HDD: 17

- **2nd Full Backup**
  - Zhu08-BDB-HDD: 432
  - Zhu08-BDB-SSD: 136
  - ChunkStash-SSD: 25x
  - ChunkStash-HDD: 370

Backup Throughput Multiplier:
- 1st Full Backup: 65x, 3.5x, 1.8x
- 2nd Full Backup: 1.2x, 3x
Performance Evaluation – Disk IOPS

- Dataset 1: 7x
- Dataset 2: 5x
- Dataset 3: 3x
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

- System builders: Don’t just treat flash as disk replacement
  - Make the OS/application layer aware of flash
  - Exploit its benefits
  - Embrace its peculiarities and design around them
  - Identify applications that can exploit sweet spot between cost and performance

- Device vendors can help by exposing more APIs to the software layer for managing storage on flash
  - Can help to squeeze better performance out of flash with application knowledge
  - E.g., Trim(), newly proposed ptrim(), exists() from fusionIO
Thank You!

Email: {sudipta, jinl}@microsoft.com