Speeding Up Cloud/Server Applications Using Flash Memory

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/firmware
  - 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
  - E.g., Trim(), newly proposed ptrim(), exists() from fusionIO
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

![Image with bar chart showing IOPS for different types of reads and writes.](image-url)
Flash for Speeding Up Cloud/Server Applications

- **FlashStore [VLDB 2010]**
  - High throughput, low latency persistent key-value store using flash as cache above HDD, RAM footprint at about 6 bytes per k-v pair
- **SkimpyStash [ACM SIGMOD 2011]**
  - Key-value store with ultra-low RAM footprint at about 1-byte per k-v pair
- **ChunkStash [USENIX ATC 2010]**
  - Efficient index design on flash for high throughput data deduplication
- **BloomFlash [ICDCS 2011]**
  - Bloom filter design for flash
- **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
Key-Value Store Applications

- Low latency backend key-value store for cloud services
  - Online multiplayer gaming (Xbox LIVE)
  - Online data serving
  - SSD caching tier in cloud storage

- Buffer pool extension for database server
  - Reduce RAM buffer pool size
  - Better performance/cost ratios

- Chunk hash index for data deduplication
  - Mitigate index lookups bottlenecks to improve throughput
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 k-v pair size
  - Target order of few bytes of RAM per k-v pair on flash
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
FlashStore Architecture

RAM write buffer for aggregating writes into flash

RAM read cache for recently accessed key-value pairs

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

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

Recently unused key-value pairs destaged to HDD
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</th>
<th>Power</th>
<th>RAM</th>
<th>Flash (SSD)</th>
<th>Hard Disk (HDD)</th>
<th>Chassis Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Core 2</td>
<td>65W</td>
<td>4GB</td>
<td>3.5W fusionIO</td>
<td>15W Seagate Barracuda</td>
<td>$1150</td>
</tr>
<tr>
<td>E8500 @3.16GHz</td>
<td></td>
<td>3.5W fusionIO 80GB</td>
<td>250GB 7200rpm</td>
<td></td>
<td></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></td>
<td></td>
<td></td>
<td>Key</td>
</tr>
<tr>
<td>Xbox</td>
<td>5.5 million</td>
<td>7.5:1</td>
<td>92</td>
</tr>
<tr>
<td>Dedup</td>
<td>40 million</td>
<td>2.2:1</td>
<td>20</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)

Blue

Xbox trace

Dedup trace

FlashStore
BerkeleyDB-SSD
BerkeleyDB-HDD

avg. ops/sec (log scale)

42527
8395
700
5x
60x
8x

57271
7325
2415
8x
24x
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

- FlashStore uses 6 bytes in RAM per key-value pair on flash.
- 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

Sequential log

Flash Memory

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

- $(4/k)$ 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

- Base design
- Enhanced Design

RAM bytes per key-value pair vs. Avg. keys per bucket (k)
Comapctation 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 = ⌊k/2m⌋

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

- Detect and remove duplicate data in storage systems
  - Primary data or backup data
  - Storage space savings
  - Disk I/O and Network bandwidth savings
- Feature offering in many storage systems products
  - Data Domain, EMC, NetApp
- Deduplication needs to complete over short time windows
  - 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

Source: Data Domain white paper
Calculate Rabin fingerprint hash for each sliding window (16 byte)
Content based Chunking

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

1. Check Metadata Cache (RAM)
2. Check Write Buffer (RAM)
3. Check Chunk Index on Flash

- Chunk hit in Metadata Cache (RAM)
- Chunk hit in Write Buffer (RAM)
- Chunk hit in Chunk Index on Flash
- Chunk miss in Metadata Cache (RAM)
- Chunk miss in Write Buffer (RAM)
- Chunk miss in Chunk Index on Flash

- This is a new chunk
- Add to the Container
- Add chunk metadata to Write Buffer

- Container is Full?
  - yes
    - Flush Write Buffer to HDD
    - Update HT Index
    - Flush Container to Container Store
  - no

- Prefetch all chunks in that Container from HDD to Metadata Cache

- This is a duplicate chunk
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**
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

Backup Throughput (MB/sec)

Dataset 2

1st Full Backup

2nd Full Backup

- Zhu08-BDB-HDD
- Zhu08-BDB-SSD
- ChunkStash-SSD
- ChunkStash-HDD

- Backup Throughput:
  - 1st Full Backup:
    - Zhu08-BDB-HDD: 3 MB/sec
    - Zhu08-BDB-SSD: 55 MB/sec
    - ChunkStash-SSD: 195 MB/sec
    - ChunkStash-HDD: 136 MB/sec
  - 2nd Full Backup:
    - Zhu08-BDB-HDD: 17 MB/sec
    - Zhu08-BDB-SSD: 109 MB/sec
    - ChunkStash-SSD: 432 MB/sec
    - ChunkStash-HDD: 432 MB/sec

- Performance Improvements:
  - 1st Full Backup:
    - Zhu08-BDB-HDD: 65x
    - Zhu08-BDB-SSD: 3.5x
    - ChunkStash-SSD: 1.8x
  - 2nd Full Backup:
    - Zhu08-BDB-HDD: 25x
    - Zhu08-BDB-SSD: 3x
    - ChunkStash-SSD: 1.2x
Performance Evaluation – Disk IOPS

- Dataset 1: 636x
- Dataset 2: 895x
- Dataset 3: 974x

Disk I/Os in BerkeleyDB relative to ChunkStash (log scale)
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

- **Flash awareness for cloud/server applications**
  - Exploit its benefits
  - Embrace its peculiarities and design around them
  - Identify applications that can exploit sweet spot between cost and performance

- **Flash controllers can be simpler (but provide same/better application level performance)**
  - Lower cost, lower power

- **Higher device endurance by mitigating random writes (reduced write amplification)**
  - Less expensive MLC flash could become feasible for more scenarios
Thank You!

Email: {sudipta, jinl}@microsoft.com