TBF: A Memory-Efficient Replacement Policy for Flash-based Caches

Biplob Debnath
NEC Laboratories America
HDD vs. Flash Memory

- Flash memory is semiconductor-based
  - Does not incur any seek cost due to lack of rotational parts

HDD

Flash Memory Based Storage

Figure Source: Internet
Flash Memory: Different Form Factors

Figure Source: Internet
Why Flash-based Cache?

- RAM is either too small or too expensive to cache the huge amount of on-disk data

- Flash is a good candidate for building a huge cache
  - 100x faster than disk
  - 10x cheaper than DRAM

Source: Flash Storage Today, ADAM Leventhal, ACM Queue, 2008
Caching Issues

- Metadata overhead in RAM to implement a caching policy
  - Flash cache size increases 1000X (GBs → TBs)
    • Metadata increases 10s of MB → 10s of GB
  - RAM consumption could limit the maximum cache size
Big Picture: Metadata Problem for Larger Cache

RAM (Gigabytes)  
- Cache Metadata  
- Cache Data  

Disk (Petabytes)
Big Picture: Metadata Problem for Larger Cache

Cache Metadata

Cache Data

RAM (Gigabytes)

Disk (Petabytes)

Flash (Terabytes)

Cache Data

RAM (Gigabytes)

Disk (Petabytes)
Big Picture: Metadata Problem for Larger Cache

Our Goal is to Reduce This Part

RAM (Gigabytes)  
Cache Metadata  
Cache Data

Disk (Petabytes)

RAM (Gigabytes)  
Flash (Terabytes)

Cache Data

Disk (Petabytes)
Outline

- Metadata for LRU Algorithm
- Related Work
- TBF Algorithm
- Experimental Results
- Summary
- References
Metadata for Least-Recently Used (LRU)

- **Index**
  - Locates an object and its related information
  - Typically, implemented as a hash table or a B-tree

- **Access Data Structure**
  - Maps an object to its recency information
    - Maintains temporal ordering
  - Helps to select eviction victims
  - Typically, implemented by doubly linked list
    - CLOCK uses one bit
Metadata for Least-Recently Used (LRU)

- **Index**
  - Locates an object and its related information
  - Typically, implemented as a hash table or a B-tree

- **Access Data Structure**
  - Maps an object to its recency information
    - Maintains temporal ordering
  - Helps to select eviction victims
  - Typically, implemented by doubly linked list
    - CLOCK uses one bit
Big Picture: Reducing Metadata

RAM

- Cache Metadata

Flash

- Cache Data

Disk

- Storage devices

2013 Storage Developer Conference. © NEC Laboratories America. All Rights Reserved.
Big Picture: Reducing Metadata

RAM

- Index
- Access Data Structure

Flash

Cache Data

Disk
Big Picture: Reducing Metadata

- Moving access data structure to flash would increase number of write operations
  - Bad for flash
Big Picture: Reducing Metadata

- We decouple index and access data structure
Big Picture: Reducing Metadata

- We move the access data structure and keep index on flash
Big Picture: Reducing Metadata

- Access data structure is implemented by Bloom filter
  - It tracks recency information

RAM
  - Bloom Filter

Flash
  - Index
  - Cache Data

Disk
Big Picture: Reducing Metadata

- We can use a key-value store for index and cache data storage
Big Picture: Reducing Metadata

- We can use a key-value store for index and cache data storage
  - Optimized for the physical properties of flash memory
Big Picture: Reducing Metadata

- **Bloom filter**
  - Tracks recency information

- **Key-Value Store**
  - Provides all functionalities of an index
  - It stores data as well
  - Operations
    - Get / Lookup
    - Set / Insert / Update
    - Delete
    - Scan
    - Replacement
      » We provide logic for it
Is It Necessary to Use a Key-Value Store?

- No, using a Key-Value store is not necessary
  - We need a mechanism to provide “lookup”, “delete” and “scan” operation
  - A flash-based hash table or B-Tree is suffice
    - Scan could be implemented by traversing the index in any order
    - Lot of excellent design choices exist in the literature (next slide)
### High-Level view of Recent Key-Value Stores

<table>
<thead>
<tr>
<th>Store</th>
<th>Cache</th>
<th>Data Store</th>
<th>Memory per object (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SkimpyStash [SIGMOD 2011]</td>
<td>X</td>
<td>√</td>
<td>1 (±0.5)</td>
</tr>
<tr>
<td>SILT [SOSP 2011]</td>
<td>X</td>
<td>√</td>
<td>0.7</td>
</tr>
<tr>
<td>FAWN-DS [SOSP 2009]</td>
<td>X</td>
<td>√</td>
<td>12</td>
</tr>
<tr>
<td>FlashStore [VLDB 2010]</td>
<td>√</td>
<td>√</td>
<td>6</td>
</tr>
<tr>
<td>BufferHash [NSDI 2010]</td>
<td>√</td>
<td>√</td>
<td>4</td>
</tr>
<tr>
<td>HashCache [NSDI 2009]</td>
<td>√</td>
<td>X</td>
<td>7</td>
</tr>
<tr>
<td>FlashCache [facebook 2010]</td>
<td>√</td>
<td>X</td>
<td>24</td>
</tr>
</tbody>
</table>

- All of these key-value stores are (or could be) optimized to cope with the physical properties of flash memory.
## High-Level view of Recent Key-Value Stores

<table>
<thead>
<tr>
<th></th>
<th>Cache</th>
<th>Data Store</th>
<th>Memory per object (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SkimpyStash [SIGMOD 2011]</td>
<td>X</td>
<td>√</td>
<td>1 (±0.5)</td>
</tr>
<tr>
<td>SILT [SOSP 2011]</td>
<td>X</td>
<td>√</td>
<td>0.7</td>
</tr>
<tr>
<td>FAWN-DS [SOSP 2009]</td>
<td>X</td>
<td>√</td>
<td>12</td>
</tr>
<tr>
<td>FlashStore [VLDB 2010]</td>
<td>√</td>
<td>√</td>
<td>6</td>
</tr>
</tbody>
</table>

Our goal is to provide a memory-efficient mechanism to implement an LRU-like algorithm based on key-value stores.
Overview of Our Caching Policy

- **Bloom filter**
  - Tracks cache hits
  - Takes one byte per object

- **Key-Value Store**
  - Determines cache hit or miss

- **Replacement Decision**
  - CLOCK-like
  - scan (key-value store) + Bloom filter + delete (key-value store)
CLOCK Algorithm (Overview)

- CLOCK-hand moves in a single direction
  - If recency bit is one, it is reset to zero
- Keep on moving until an object with recency bit zero is found
CLOCK-hand Movement

- CLOCK-hand resets recency bit to forget past information
- CLOCK-hand needs a full traversal to return its current position
CLOCK Algorithm (Overview) Contd.

- New object is inserted just behind the CLOCK-hand
- Two variants exist: Initial value of recency bit could be set to either 0 or 1
- Recency bits is set to ‘1’ whenever an object gets hit
Bloom Filter (overview)

- A probabilistic data structure for testing set membership
  - Provides a compact representation of a set of elements
Bloom Filter (overview)

- A probabilistic data structure for testing set membership
  - Provides a compact representation of a set of elements
- Consists of an array of bits
  - Initially all bits are set to ‘0’
  - Multiple independent hash functions are used to calculate bit positions
  - Insertions: All corresponding bits positions are set to ‘1’s
  - Lookup: Check if all corresponding bits are ‘1’s
Bloom Filter: Bit Collisions

- Due to bit collisions
  - Suffers from false positive problem
  - Nonexistent keys may appear to be present
  - Deletion is problematic
  - Introduces false negative

- Array of bits

h1(k3)

h2(k3)
k3 (lookup)

an array of bits
Bloom Filter: False Negative Example

- **Due to bit collisions**
  - Suffers from false positive problem
    - Nonexistent keys may appear to be present
  - Deletion is problematic
    - Introduces false negative
    - Removing k1 (or k2) might also remove k2 (or k1)
Our Replacement Decision

- **scan** operation (key-value store) can emulate CLOCK-hand movement
  - For every candidate object
    - Lookup recency information in Bloom filter
  - Absent (i.e., not recently accessed)
    - Select as a victim
    - **delete** from key–value store
  - Present
    - Keep in cache
    - “delete” recency information from Bloom filter (i.e., reset recency info)
    - Skip to the next candidate object
How to Forget Past Recency Information?

- By deleting bits in the Bloom filter (BFD)
  - Explained in the paper
  - Suffers from false-negative problem
  - Recently accessed key-value pairs might be evicted
    - Decreases hit ratio
    - Hurts performance
  - Counting Bloom filter could be used
    - Increases memory consumption
How to Forget Past Recency Information? Contd.

- Two Bloom filters (TBF)
  - False-negative free
  - Only one of filters is active at any point of time
    - Cache hits are recorded in the active filter
  - Both filters are consulted to select a victim
    - If recency information is found in any of the two filters, not selected for eviction
- Bulk Delete
  - Periodically all the bits of passive filter are reset, and filters are swapped
Illustration: Two Bloom Filters (TBF)

Two RAM Bloom Filters

On-flash Key-value Store

active

passive

scan pointer

1 1

1 1 1

C

A

B

D

1 1 1

1
TBF: How To Size Bloom filters?

Two RAM Bloom Filters

No. of bits per object: 4
Hash functions: 3
Total: 4*2 = 8 bits (1 byte) per object
TBF: When to Swap Filters?

- Our logic is derived from CLOCK
  - CLOCK-hand takes a full traversal to come to its initial position

- Filters are swapped when clock-pointer has examined a fixed number of elements
  - This fixed number is proportional to the cache capacity
What is the Cost of Replacement?

- To select a victim, we need to traverse on-flash key-value store
  - Incurs flash read operations
  - We use CLOCK variant with initial value of receney bit is set to ‘0’
    - We set Bloom filter bits only on cache hit
      » That is why, we need less bit per object compared to standard Bloom filter implementation
    - Reduces extra flash read operations
  - We also limit traversal length
    - Based on the ratio of flash and disk access latency
What are the Differences from CLOCK?

- False positive in the access data structure
  - Arises from the use of Bloom filter
  - Cold objects could be incorrectly considered as recently accessed

- To select victims, we need to traverse on-flash key-value store
  - Incurs flash read operations

- Position of a new key-value pair depends on the key-value store implementation
  - Unlike just behind the CLOCK-hand
Experimental Evaluation

- Simulations: On paper
- Real-Prototype Setup:

YCSB Workload Generator

![](diagram)

BerkeleyDB (150M 256B kv pairs)

Flash

fusionIO

Key-Value Store (Cache)

Cache Adapter (RAM)

Key-Value Store (Database)

BerkeleyDB (1500M 256B kv pairs)

DISK

RAID0 (2 Disks)
Results: IOPs and Memory Usage

<table>
<thead>
<tr>
<th>Policy</th>
<th>Bytes Per Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBF</td>
<td>1</td>
</tr>
<tr>
<td>CLOCK</td>
<td>9</td>
</tr>
<tr>
<td>LRU</td>
<td>20</td>
</tr>
<tr>
<td>RANDOM</td>
<td>0</td>
</tr>
<tr>
<td>NONE</td>
<td>0</td>
</tr>
</tbody>
</table>
Results: Hit Ratio

<table>
<thead>
<tr>
<th>Workload</th>
<th>Cache Hit Rate</th>
<th>Avg. No of Key Traversal by TBF</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TBF</td>
<td>CLOCK</td>
</tr>
<tr>
<td>YCSB-Latest</td>
<td>84.9</td>
<td>84.8</td>
</tr>
<tr>
<td>YCSB-Zipf</td>
<td>77.7</td>
<td>77.6</td>
</tr>
<tr>
<td>YCSB-Uniform</td>
<td>10.0</td>
<td>10.0</td>
</tr>
</tbody>
</table>
Summary

- TBF introduces a memory-efficient mechanism to implement an LRU-like algorithm
  - Generic Solution
    - Could be applied to other new technologies (e.g. PCM)
    - Suitable for enhancing existing key-value stores to use as caches
      - Agnostic to key-value store implementation

- TBF requires one byte of additional memory per object
  - Thus, suitable for implementing very large cache

- TBF provides performance similar to LRU and CLOCK
References

- **SkimpyStash: A RAM Space Skimpy Key-Value Store on Flash-based Storage.** Biplob Debnath, Sudipta Sengupta, Jin Li. In ACM SIGMOD 2011 Conference.
- **FlashStore: High Throughput Persistent Key-Value Store.** Biplob Debnath, Sudipta Sengupta, Jin Li. In VLDB 2010.
- **SILT: A Memory-Efficient, High-Performance Key-Value Store.** Hyeontaek Lim, Bin Fan, David Andersen, Michael Kaminsky. In SOSP 2011.
- **Cheap and Large CAMs for High-performance Data-intensive Networked Systems.** Ashok Anand, Chitra Muthukrishnan, Steven Kappes, Aditya Akella and Suman Nath. In NSDI 2010
- **FlashCache:** [https://github.com/facebook/flashcache/](https://github.com/facebook/flashcache/)

biplob@nec-labs.com

Thank You!