Cheap and Large CAMs for High Performance Data-Intensive Networked Systems

Ashok Anand, Chitra Muthukrishnan, Steven Kappes, and Aditya Akella
University of Wisconsin-Madison

Suman Nath
Microsoft Research
New data-intensive networked systems

Large hash tables (10s to 100s of GBs)
New data-intensive networked systems

- Object store (~4 TB)
- Hashtables (~32 GB)
- Large hash tables (32 GB)
- High speed (~10K/sec) lookups for 500 Mbps link
- High speed (~10 K/sec) inserts and evictions

Key (20 B) → Chunk pointer

Data center

Branch office

WAN optimizers

WAN

Object

Chunks (4 KB)

Look up

Object store (~4 TB)
New data-intensive networked systems

• Other systems
  – De-duplication in storage systems (e.g., Datadomain)
  – CCN cache (Jacobson et al., CONEXT 2009)
  – DONA directory lookup (Koponen et al., SIGCOMM 2006)

Cost-effective large hash tables
Cheap Large cAMs
Candidate options

<table>
<thead>
<tr>
<th></th>
<th>Random reads/sec</th>
<th>Random writes/sec</th>
<th>Cost (128 GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disk</td>
<td>250</td>
<td>250</td>
<td>$30\textsuperscript{+}</td>
</tr>
<tr>
<td>DRAM</td>
<td>300K</td>
<td>300K</td>
<td>$120K\textsuperscript{+}</td>
</tr>
<tr>
<td>Flash-SSD</td>
<td>10K\textsuperscript{*}</td>
<td>5K\textsuperscript{*}</td>
<td>$225\textsuperscript{+}</td>
</tr>
</tbody>
</table>

\textsuperscript{+}Price statistics from 2008-09

* Derived from latencies on Intel M-18 SSD in experiments

How to deal with slow writes of Flash SSD
Our CLAM design

• New data structure “BufferHash” + Flash
• Key features
  – Avoid random writes, and perform sequential writes in a batch
    • Sequential writes are 2X faster than random writes (Intel SSD)
    • Batched writes reduce the number of writes going to Flash
  – Bloom filters for optimizing lookups

BufferHash performs orders of magnitude better than DRAM based traditional hash tables in ops/sec/$
Outline

• Background and motivation

• CLAM design
  – Key operations (insert, lookup, update)
  – Eviction
  – Latency analysis and performance tuning

• Evaluation
Flash/SSD primer

• Random writes are expensive
  Avoid random page writes

• Reads and writes happen at the granularity of a flash page
  I/O smaller than page should be avoided, if possible
Conventional hash table on Flash/SSD

Keys are likely to hash to random locations

SSDs: FTL handles random writes to some extent; But garbage collection overhead is high

~200 lookups/sec and ~200 inserts/sec with WAN optimizer workload, << 10 K/s and 5 K/s
Conventional hash table on Flash/SSD

Can’t assume locality in requests – DRAM as cache won’t work
Our approach: Buffering insertions

- Control the impact of random writes
- Maintain small hash table (buffer) in memory
- As in-memory buffer gets full, write it to flash
  - We call in-flash buffer, incarnation of buffer

Buffer: In-memory hash table

Incarnation: In-flash hash table
Two-level memory hierarchy

Net hash table is: buffer + all incarnations
Lookups are impacted due to buffers

Multiple in-flash lookups. Can we limit to only one?
Bloom filters for optimizing lookups

In-memory lookups
False positive!

Configure carefully!

2 GB Bloom filters for 32 GB Flash for false positive rate < 0.01!
Update: naïve approach

Update key

Expensive random writes

Discard this naïve approach
Lazy updates

Update key

Insert key

Key, new value

Key, old value

Lookups check latest incarnations first
Eviction for streaming apps

- Eviction policies may depend on application
  - LRU, FIFO, Priority based eviction, etc.
- Two BufferHash primitives
  - Full Discard: evict all items
    - Naturally implements FIFO
  - Partial Discard: retain few items
    - Priority based eviction by retaining high priority items
- BufferHash best suited for FIFO
  - Incarnations arranged by age
  - Other useful policies at some additional cost
- Details in paper
Issues with using one buffer

- Single buffer in DRAM
  - All operations and eviction policies

- High worst case insert latency
  - Few seconds for 1 GB buffer
  - New lookups stall

![Diagram showing DRAM, Bloom filters, Flash, and Incarnation table.]
Partitioning buffers

- Partition buffers
  - Based on first few bits of key space
  - Size > page
    - Avoid i/o less than page
  - Size >= block
    - Avoid random page writes
- Reduces worst case latency
- Eviction policies apply per buffer

![Diagram of partitioning buffers]
BufferHash: Putting it all together

- Multiple buffers in memory
- Multiple incarnations per buffer in flash
- One in-memory bloom filter per incarnation

Net hash table = all buffers + all incarnations
Outline

• Background and motivation

• *Our CLAM design*
  – Key operations (insert, lookup, update)
  – Eviction
    – *Latency analysis and performance tuning*

• Evaluation
Latency analysis

• Insertion latency
  – Worst case \( \propto \) size of buffer
  – Average case is constant for buffer > block size

• Lookup latency
  – Average case \( \propto \) Number of incarnations
  – Average case \( \propto \) False positive rate of bloom filter
Parameter tuning: Total size of Buffers

Total size of buffers = B1 + B2 + ... + BN

Given fixed DRAM, how much allocated to buffers

Total bloom filter size = DRAM – total size of buffers

Too small is not optimal
Too large is not optimal either
Optimal = 2 * SSD/entry
Parameter tuning: Per-buffer size

What should be size of a partitioned buffer (e.g. B1)?

Affects worst case insertion

Adjusted according to application requirement (128 KB – 1 block)
Outline

• Background and motivation

• Our CLAM design
  – Key operations (insert, lookup, update)
  – Eviction
  – Latency analysis and performance tuning

• Evaluation
Evaluation

• Configuration
  – 4 GB DRAM, 32 GB Intel SSD, Transcend SSD
  – 2 GB buffers, 2 GB bloom filters, 0.01 false positive rate
  – FIFO eviction policy
BufferHash performance

• WAN optimizer workload
  – Random key lookups followed by inserts
  – Hit rate (40%)
  – Used workload from real packet traces also

• Comparison with BerkeleyDB (traditional hash table) on Intel SSD

<table>
<thead>
<tr>
<th>Average latency</th>
<th>BufferHash</th>
<th>BerkeleyDB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Look up (ms)</td>
<td>0.06</td>
<td>4.6</td>
</tr>
<tr>
<td>Insert (ms)</td>
<td>0.006</td>
<td>4.8</td>
</tr>
</tbody>
</table>

Better lookups!
Better inserts!
**Insert performance**

- **Bufferhash**
  - 99% inserts < 0.1 ms

- **BerkeleyDB**
  - 40% of inserts > 5 ms!

**Buffering effect!**

**Random writes are slow!**
Lookup performance

99% of lookups < 0.2ms
40% of lookups > 5 ms
Garbage collection overhead due to writes!

0.15 ms Intel SSD latency
Performance in Ops/sec/$

• 16K lookups/sec and 160K inserts/sec

• Overall cost of $400

• 42 lookups/sec/$ and 420 inserts/sec/$
  – Orders of magnitude better than 2.5 ops/sec/$ of DRAM based hash tables
Other workloads

• Varying fractions of lookups
• Results on Trancend SSD

<table>
<thead>
<tr>
<th>Lookup fraction</th>
<th>BufferHash</th>
<th>BerkeleyDB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.007 ms</td>
<td>18.4 ms</td>
</tr>
<tr>
<td>0.5</td>
<td>0.09 ms</td>
<td>10.3 ms</td>
</tr>
<tr>
<td>1</td>
<td>0.12 ms</td>
<td>0.3 ms</td>
</tr>
</tbody>
</table>

• BufferHash ideally suited for write intensive workloads
Evaluation summary

• BufferHash performs orders of magnitude better in ops/sec/$ compared to traditional hashtables on DRAM (and disks)

• BufferHash is best suited for FIFO eviction policy
  – Other policies can be supported at additional cost, details in paper

• WAN optimizer using BufferHash can operate optimally at 200 Mbps, much better than 10 Mbps with BerkeleyDB
  – Details in paper
Related Work

• FAWN (Vasudevan et al., SOSP 2009)
  – Cluster of wimpy nodes with flash storage
  – Each wimpy node has its hash table in DRAM
  – We target...
    • *Hash table much bigger than DRAM*
    • *Low latency as well as high throughput systems*

• HashCache (Badam et al., NSDI 2009)
  – In-memory hash table for objects stored on disk
Conclusion

• We have designed a new data structure BufferHash for building CLAMs.

• Our CLAM on Intel SSD achieves high ops/sec/$ for today’s data-intensive systems.

• Our CLAM can support useful eviction policies.

• Dramatically improves performance of WAN optimizers.
Thank you