Memshare: a Dynamic Multi-tenant Key-value Cache

ASA F CIDON*, DANIEL RUSHTON†, STEPHEN M. RUMBLE‡, RYAN STUTSMAN†

*STANFORD UNIVERSITY, †UNIVERSITY OF UTAH, ‡GOOGLE INC.
Cache is 100X Faster Than Database

- Web Server
- MySQL
- memCached

100 us
10 ms
Cache Hit Rate Drives Cloud Performance

• Small improvements to cache hit rate make big difference:

• At 98% cache hit rate:
  • +1% hit rate $\Rightarrow$ 35% speedup
  • Facebook study [Atikoglu ’12]
Static Partitioning → Low Hit Rates

• Cache providers statically partition their memory among applications

• Examples:
  • Facebook
  • Amazon Elasticache
  • Memcachier
Partitioned Memory Over Time

Static Partition

No Partition

Time (Hours)

Cache Occupancy (MB)

App A  App B  App C
## Partitioned vs No Partition Hit Rates

<table>
<thead>
<tr>
<th>Application</th>
<th>Hit Rate Partitioned</th>
<th>Hit Rate No Partition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Combined</td>
<td>87.8%</td>
<td>88.8%</td>
</tr>
<tr>
<td>A</td>
<td>97.6%</td>
<td>96.6%</td>
</tr>
<tr>
<td>B</td>
<td>98.8%</td>
<td>99.1%</td>
</tr>
<tr>
<td>C</td>
<td>30.1%</td>
<td>39.2%</td>
</tr>
</tbody>
</table>
Partitioned Memory: Pros and Cons

• Disadvantages:
  • Lower hit rate due to low utilization
  • Higher TCO

• Advantages:
  • Isolated performance and predictable hit rate
  • “Fairness”: customers get what they pay for
Memshare: the Best of Both Worlds

- Optimize memory allocation to maximize overall hit rate
- While providing minimal guaranteed memory allocation and performance isolation
Multi-tenant Cache Design Challenges

1. Decide application memory allocation to optimize hit rate
2. Enforce memory allocation among applications
Estimate Hit Rate Curve Gradient to Optimize Hit Rate

Workload 1

Hit Rate

Cache Allocation

Workload 2

Hit Rate

Cleaning Enforces Allocations and Gives High Utilization

Keep items from Workload 1

∇ < ∇ →
Estimate Hit Rate Curve Gradient to Optimize Hit Rate

\[ \nabla w_1 < \nabla w_2 \rightarrow \text{Keep items from } w_2 \]
Estimating Hit Rate Gradient

- Track access frequency to recently evicted objects to determine gradient at working point
- Can be further improved with full hit rate curve estimation
  - SHARDS [Waldspurger 2015, 2017]
  - AET [Hu 2016]
Multi-tenant Cache Design Challenges

1. Decide application memory allocation to optimize hit rate
2. Enforce memory allocation among applications
Multi-tenant Cache Design Challenges

1. Decide application memory allocation to optimize hit rate

2. Enforce memory allocation among applications

Not so simple
Slab Allocation Primer
Slab Allocation Primer
Slab Allocation Primer

Memcached Server

LRU Queues

App 1  App 2
Goal: Move 4KB from App 2 to App 1
Goal: Move 4KB from App 2 to App 1

- Problems:
  - Need to evict 1MB
  - Contains many small objects, some are hot
  - App 1 can only use extra space for objects of certain size
Goal: Move 4KB from App 2 to App 1

- Problems:
  - Need to evict 1MB
  - Contains many small objects, some are hot
  - App 1 can only use extra space for objects of certain size

Problematic even for one application, see Cliffhanger [Cidon 2016]
Instead of Slabs: Log-structured Memory
Instead of Slabs: Log-structured Memory
Instead of Slabs: Log-structured Memory

Log segments

Log Head
Applications are Physically Intermixed

Log segments

Log Head

App 1  App 2
Memshare’s Sharing Model

- Reserved Memory: guaranteed static memory
- Pooled Memory: application’s share of pooled memory
- Target Memory = Reserved Memory + Pooled Memory
Cleaning Priority Determines Eviction Priority

- **Q:** When does Memshare evict?
- **A:** Newly written objects evict old objects, but not in critical path
  - Cleaner keeps 1% of cache empty
  - Cleaner tries to enforce actual memory allocation to be equal to Target Memory
Cleaner Pass

n candidate segments (n = 2)

n - 1 survivor segments (n = 2)

App 1  App 2
Cleaner Pass

n candidate segments (n = 2)

n - 1 survivor segments (n = 2)
Cleaner Pass

n candidate segments (n = 2)

n - 1 survivor segments (n = 2)

App 1  App 2
Cleaner Pass

n candidate segments (n = 2)

n - 1 survivor segments (n = 2)

App 1  App 2
Cleaner Pass

n candidate segments (n = 2)

n - 1 survivor segments (n = 2)

App 1  App 2
Cleaner Pass

n candidate segments (n = 2)

n - 1 survivor segments (n = 2)

1 free segment
Cleaner Pass (n = 4): Twice the Work

4 candidate segments (n = 4)

3 survivor segments (n = 4)

1 free segment
Application Need: How Far is Memory Allocation from Target Memory?

\[
\text{need}(\text{app}) = \frac{\text{targetMemory}(\text{app})}{\text{actualMemory}(\text{app})}
\]
Within Each Application, Evict by Rank

- To implement LRU: rank = last access time
Cleaning: Max Need and then Max Rank

Max Need?
Max Rank?

<table>
<thead>
<tr>
<th>Need</th>
<th>0.8</th>
<th>1.4</th>
<th>0.9</th>
</tr>
</thead>
<tbody>
<tr>
<td>App 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>App 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>App 3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Cleaning: Max Need and then Max Rank

Max Need? → App 2
Max Rank?

<table>
<thead>
<tr>
<th>Need</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>App 1</td>
<td>0.8</td>
</tr>
<tr>
<td>App 2</td>
<td>1.4</td>
</tr>
<tr>
<td>App 3</td>
<td>0.9</td>
</tr>
</tbody>
</table>
Cleaning: Max Need and then Max Rank

n segments

Rank 2  Rank 1  Rank 0  Rank 3

Max Need? → App 2
Max Rank? → Rank 2

n-1 segments

Need

<table>
<thead>
<tr>
<th>App 1</th>
<th>0.8</th>
</tr>
</thead>
<tbody>
<tr>
<td>App 2</td>
<td>1.4</td>
</tr>
<tr>
<td>App 3</td>
<td>0.9</td>
</tr>
</tbody>
</table>
Cleaning: Max Need and then Max Rank

<table>
<thead>
<tr>
<th>Need</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>App 1</td>
<td>0.9</td>
</tr>
<tr>
<td>App 2</td>
<td>0.8</td>
</tr>
<tr>
<td>App 3</td>
<td>1.2</td>
</tr>
</tbody>
</table>

Max Need?
Max Rank?
Cleaning: Max Need and then Max Rank

Max Need? → App 3
Max Rank?

<table>
<thead>
<tr>
<th>Need</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>App 1</td>
<td>0.9</td>
</tr>
<tr>
<td>App 2</td>
<td>0.8</td>
</tr>
<tr>
<td>App 3</td>
<td>1.2</td>
</tr>
</tbody>
</table>
Cleaning: Max Need and then Max Rank

<table>
<thead>
<tr>
<th>Need</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>App 1</td>
<td>0.9</td>
</tr>
<tr>
<td>App 2</td>
<td>0.8</td>
</tr>
<tr>
<td>App 3</td>
<td>1.2</td>
</tr>
</tbody>
</table>

Max Need? → App 3
Max Rank? → Rank 1
Trading Off Eviction Accuracy and Cleaning Cost

- Eviction accuracy is determined by \( n \)
  - For example: rank = time of last access
  - When \( n \rightarrow \# \) segments: ideal LRU
  - Intuition: \( n \) is similar to cache associativity

- CPU consumption is determined by \( n \)
Trading Off Eviction Accuracy and Cleaning Cost

• Eviction accuracy is determined by
  
  For example:
  
  When
  
  Intuition: is similar to cache associativity
  
  CPU consumption is determined by

“In practice Memcached is never CPU-bound in our data centers. Increasing CPU to improve the hit rate would be a good trade off.”

  - Nathan Bronson, Facebook
Implementation

- Implemented in C++ on top of Memcached
- Reuse Memcached’s hash table, transport, request processing
- Implemented log-structured memory allocator
## Partitioned vs. Memshare

<table>
<thead>
<tr>
<th>Application</th>
<th>Hit Rate Partitioned</th>
<th>Hit Rate Memshare (50% Reserved)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Combined</td>
<td>87.8%</td>
<td>89.2%</td>
</tr>
<tr>
<td>A</td>
<td>97.6%</td>
<td>99.4%</td>
</tr>
<tr>
<td>B</td>
<td>98.8%</td>
<td>98.8%</td>
</tr>
<tr>
<td>C</td>
<td>30.1%</td>
<td>34.5%</td>
</tr>
</tbody>
</table>
Reserved vs. Pooled Behavior

- Combined Hit Rates:
  - App A: 90.2%
  - App B: 89.2%
  - App C: 88.8%
State-of-the-art Hit rate

- Misses reduced by 40%
- Combined hit rate increase: 6% (85% → 91%)
State-of-the-art Hit Rate Even for Single Tenant Applications

<table>
<thead>
<tr>
<th>Policy</th>
<th>Memcached</th>
<th>Memshare (100% Reserved)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average Single Tenant Hit Rate</td>
<td>88.3%</td>
<td>95.5%</td>
</tr>
</tbody>
</table>
Cleaning Overhead is Minimal

- Hit rate
- Memory Bandwidth

n (number of cleaning candidate segments)

88.00% 88.50% 89.00% 89.50% 90.00% 90.50% 91.00%

1 10 20 40 60 80 100

0 2 4 6 8 10 MB/s
Cleaning Overhead is Minimal

Modern servers have 10GB/s or more!
Related Work

• Optimizing memory allocation using shadow queues
  • Cliffhanger [Cidon 2016]

• Log-structured single-tenant key-value stores
  • RAMCloud [Rumble 2014] and MICA [Lim 2014]

• Taxing idle memory
  • ESX Server [Waldspurger 2002]
Summary

• First multi-tenant key-value cache that:
  • Optimizes share for highest hit rate
  • Provides minimal guarantees

• Novel log-structured design
  • Use cleaner as enforcer
Appendix
Idle Tax for Selfish Applications

- Some sharing models do not support pooled memory, each application is selfish
  - For example: Memcached’s Cache-as-a-Service
- Idle tax: reserved memory can be reassigned if idle
- Tax rate: determines portion of idle memory that can be reassigned
- If all memory is active: target memory = reserved memory
## Partitioned vs. Idle Tax

<table>
<thead>
<tr>
<th>Application</th>
<th>Hit Rate Partitioned</th>
<th>Hit Rate Memshare Idle Tax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Combined</td>
<td>87.8%</td>
<td>88.8%</td>
</tr>
<tr>
<td>A</td>
<td>97.6%</td>
<td>99.4%</td>
</tr>
<tr>
<td>B</td>
<td>98.8%</td>
<td>98.6%</td>
</tr>
<tr>
<td>C</td>
<td>30.1%</td>
<td>31.3%</td>
</tr>
</tbody>
</table>
State-of-the-art Hit rate

Memcached
Cliffhanger
Memshare (75% Reserved)

- Combined Hit Rate
- Miss Reduction vs. Memcached
Nearly Identical Latency