Cliffhanger: Scaling Performance Cliffs in Memory Caches [NSDI 2016]
Cache OS: Data Center Dynamic Cache Management

Asaf Cidon, Assaf Eisenman,
Mohammad Alizadeh and Sachin Katti
Key-Value Caches are Essential to Web-scale Application Performance

• Web-scale applications heavily reliant on caches
Caches Drive Performance and Availability

• Memcached most widely used cache in large data centers

• Single Memcached failure can cause 5 minute loss of service [Box, 10/14]

• +1% cache hit-rate → 35% speedup
  – Old latency: 374 µs
  – New latency: 278 µs

  – Facebook study: Atikoglu et al [Sigmetrics ’12]
Cache Clusters are Static and Difficult to Manage

• Caches are static
  – Cache allocation is static, not application aware
  – Applications are statically split into separate cache server pools
  – Cache server pool sizes determined arbitrarily

• Cache clusters are hard to evaluate and optimize
  – Heterogeneous cache resources (memory, SSD, NV RAM) managed and allocated independently
  – No way to define QoS/prioritization for applications
Cache Clusters are Static and Difficult to Manage

• Caches are static
  – Cache allocation is static, not application aware
  – Applications are statically split into separate cache server pools
  – Cache server pool sizes determined arbitrarily

• Cache clusters are hard to evaluate and optimize
  – Heterogeneous cache resources (memory, SSD, NV RAM) managed and allocated independently
  – No way to define QoS/prioritization for applications
Static Cache Allocation Far From Optimal

- Cache arbitrarily assigns memory to different request sizes
- Does not optimize for hit rate across different request sizes and applications
- Memory assignment remains static
Memcached’s Static Cache Allocation

Memcached Server

Item 1
35 bytes
Memcached’s Static Cache Allocation

Memcached Server

Item 1
35 bytes
Memcached’s Static Cache Allocation

Memcached Server

Item 2
60 bytes

…

Item 8
60 bytes

8 7 6 5 4 3 2 1

64 byte slab
Memcached’s Static Cache Allocation

Memcached Server

Item 9
100 bytes

Item 13
378 bytes

64 byte slab
128 byte slab
512 byte slab
Memcached’s Static Cache Allocation

- Item 10: 100 bytes
- Item 12: 100 bytes
- Item 14: 433 bytes
- Item 20: 100 bytes

- slab 8: 128 bytes
- slab 9: 64 bytes
- slab 13: 512 bytes

Memcached Server

- 64 byte slab
- 128 byte slab
- 512 byte slab
Memcached’s Static Cache Allocation

Memcached Server

- 64 byte slab
- 128 byte slab
- 512 byte slab

Item 10: 100 bytes
Item 12: 100 bytes
Item 14: 433 bytes
Item 20: 100 bytes
Memcached’s Static Cache Allocation

Memcached Server

- 64 byte slab
- 128 byte slab
- 512 byte slab

Item 9
120 bytes
Memcached’s Static Cache Allocation

Memcached Server

64 byte slab

128 byte slab

512 byte slab

Item 9
120 bytes
Memcached’s Static Cache Allocation

Memcached Server

Item 21
43 bytes

8 7 6 5 4 3 2 1
9 12 11 10
20 19 18 17
16 15 14 13

64 byte slab
128 byte slab
512 byte slab
Memcached’s Static Cache Allocation

- 64 byte slab
- 128 byte slab
- 512 byte slab

Item 21
43 bytes
Memcached’s Static Cache Allocation
Memcached’s Static Cache Allocation

Memcached Server

- 32 byte slab
- 64 byte slab
- 128 byte slab
- 512 byte slab

Item 1: 35 bytes
Problems with Memcached Static Cache Allocation

1. Greedy slab class allocation favors large slab classes
2. “Slab calcification” when request sizes change over time

• Can we do better?
Understanding Memcached Workloads With MemCachier Traces

• Weeklong trace taken from MemCachier
  – 490 applications on 30 Memcached servers
  – Each application has its own pages
Profiling Hit Rate Curves

Stack distances:
5
1
∞
Hit Rate Curve Profiling

Application 4, Slab 0

Hit Rate

Number of Chunks in Queue

x 10^4
Optimizing Hit Rate Curves

Application 4, Slab 0

Application 4, Slab 1
Optimize Memory Allocation Using Hit-rate curves

\[
\text{maximize} \quad \sum_{i=1}^{s} f_i h_i(m_i) \\
\text{subject to} \quad \sum_{i=1}^{s} m_i \leq M
\]

f – frequency of requests
h – hit-rate of requests
m – memory allocated to slab class
M – memory allocated to application
Optimizing Hit Rate Curves

Allocate 1178 Items

Allocate 41381 Items

Application 4, Slab C

Application 4, Slab 1
Potential for Improvement

The bar chart illustrates the hit rate for Memcached applications, comparing default hit rates (1/3 memory) with optimized hit rates (1/3 memory). The applications are numbered from 1 to 20, with some marked with an asterisk (*) indicating special conditions or optimizations.
Potential for Improvement

+19% Hit Rate
+66% Hit Rate
+45% Hit Rate
Optimizing Hit Rate Curves is Expensive and Not Dynamic

• Hit rate optimization is expensive
  – Requires estimating stack distances for each curve
  – Requires centralized optimizer

• Hit rate optimization is static
  – How frequently should we optimize?

• Instead of optimizing entire hit rate curve, we can optimize incrementally
  – Estimate local gradient for each curve
  – Increase memory for curve with highest gradient
Using Shadow Queues to Estimate Local Gradient

<table>
<thead>
<tr>
<th>Queue</th>
<th>Credits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Queue 1</td>
<td>0</td>
</tr>
<tr>
<td>Queue 2</td>
<td>0</td>
</tr>
</tbody>
</table>
Using Shadow Queues to Estimate Local Gradient

<table>
<thead>
<tr>
<th>Queue</th>
<th>8</th>
<th>53</th>
<th>1</th>
<th>22</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Queue 1</td>
<td>Physical Queue</td>
<td>Shadow Queue</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Queue</th>
<th>9</th>
<th>87</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Queue 2</td>
<td>Physical Queue</td>
<td>Shadow Queue</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Credits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Queue 1</td>
<td>0</td>
</tr>
<tr>
<td>Queue 2</td>
<td>0</td>
</tr>
</tbody>
</table>
Using Shadow Queues to Estimate Local Gradient

<table>
<thead>
<tr>
<th>Queue 1</th>
<th>Credits</th>
</tr>
</thead>
<tbody>
<tr>
<td>23 8 53 1 22</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Queue 2</th>
<th>Credits</th>
</tr>
</thead>
<tbody>
<tr>
<td>9 87</td>
<td>0</td>
</tr>
</tbody>
</table>
Using Shadow Queues to Estimate Local Gradient

<table>
<thead>
<tr>
<th></th>
<th>Credits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Queue 1</td>
<td>0</td>
</tr>
<tr>
<td>Queue 2</td>
<td>0</td>
</tr>
</tbody>
</table>
Using Shadow Queues to Estimate Local Gradient

<table>
<thead>
<tr>
<th>Queue</th>
<th>Credits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Queue 1</td>
<td>1</td>
</tr>
<tr>
<td>Queue 2</td>
<td>-1</td>
</tr>
</tbody>
</table>
Using Shadow Queues to Estimate Local Gradient

<table>
<thead>
<tr>
<th></th>
<th>Credits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Queue 1</td>
<td>1</td>
</tr>
<tr>
<td>Queue 2</td>
<td>-1</td>
</tr>
</tbody>
</table>
Using Shadow Queues to Estimate Local Gradient

<table>
<thead>
<tr>
<th></th>
<th>Credits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Queue 1</td>
<td>1</td>
</tr>
<tr>
<td>Queue 2</td>
<td>-1</td>
</tr>
</tbody>
</table>
Using Shadow Queues to Estimate Local Gradient

<table>
<thead>
<tr>
<th>Credits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Queue 1</td>
</tr>
<tr>
<td>Queue 2</td>
</tr>
</tbody>
</table>
Using Shadow Queues to Estimate Local Gradient

<table>
<thead>
<tr>
<th>Queue 1</th>
<th>Queue 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Physical Queue</td>
<td>Physical Queue</td>
</tr>
<tr>
<td>Shadow Queue</td>
<td>Shadow Queue</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Credits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Queue 1</td>
</tr>
<tr>
<td>Queue 2</td>
</tr>
</tbody>
</table>

Resize Queues
Using Shadow Queues to Estimate Local Gradient

<table>
<thead>
<tr>
<th></th>
<th>Credits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Queue 1</td>
<td>0</td>
</tr>
<tr>
<td>Queue 2</td>
<td>-2</td>
</tr>
</tbody>
</table>
Algorithm 1: Hill-climbing

Algorithm 1 Hill Climbing Algorithm

1: if request $\in$ shadowQueue(i) then
2:    queue(i).size = queue(i).size + credit
3:    chosenQueue = pickRandom({queues} - {queue(i)})
4:    chosenQueue.size = chosenQueue.size - credit
5: end if
Algorithm 1: Hill-climbing

**Algorithm 1 Hill Climbing Algorithm**

1: if request ∈ shadowQueue(i) then
2: queue(i).size = queue(i).size + credit
3: chosenQueue = pickRandom({queues} - {queue(i)})
4: chosenQueue.size = chosenQueue.size - credit
5: end if

Approximates optimization
Performance Cliffs Hurt Local Optimization

![Graph](image)

**Application 19, Slab Class 0**

**Application 11, Slab Class 6**
Why Do Performance Cliffs Occur?

• Applications issues requests 1, 2, 3, 4, 5
• Queue size = 4
  – 0% hitrate
Intuition Behind Talus

• 2 queues with size = 2
  – First queue gets 1, 2, 3
  – Second queue gets 4, 5
  – First queue hitrate: 0%
  – Second queue hitrate: 100%
  – Overall hitrate: 40%
Talus: Simulating Two Virtual Queues

![Graph showing hitrate versus number of items in LRU Queue]

- **Concave Hull**
- **Application 19, Slab 0**

Number of Items in LRU Queue vs. Hitrate graph.
Talus: Simulating Two Virtual Queues

- Left virtual queue: 2000 items
- Right virtual queue: 13,000 items
Algorithm 2: Cliff Scaling

• Talus requires knowledge of hitrate curve
  – Where the performance cliff starts and ends

• Algorithm 2 locally estimates where the performance cliff starts and ends
  – Estimate the second derivative with shadow queues
Estimating Second Derivative with Shadow Queues

Number of Items in LRU Queue

Hitrate

Queue Size

Shadow Queues

Left Queue

Right Queue

Concave Hull

Application 19, Slab 0
Cliffhanger Runs Both Algorithms in Parallel

• Algorithm 1: incrementally optimize memory across queues
  – Across slab classes
  – Across applications

• Algorithm 2: scales performance cliffs
Cliffhanger Reduces Misses and Can Save Memory

• Average misses reduced: 36.7%
• Average potential memory savings: 55%
Cliffhanger Outperforms Default and Optimized Schemes

- Average Cliffhanger hitrate increase: 1.2%
Low Overheads

• Latency overhead:

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Operation</th>
<th>Cache Hit</th>
<th>Cache Miss</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hill Climbing</td>
<td>GET</td>
<td>0%</td>
<td>1.4%</td>
</tr>
<tr>
<td>Hill Climbing</td>
<td>SET</td>
<td>0%</td>
<td>4.7%</td>
</tr>
<tr>
<td>Cliffhanger</td>
<td>GET</td>
<td>0.8%</td>
<td>1.4%</td>
</tr>
<tr>
<td>Cliffhanger</td>
<td>SET</td>
<td>0.8%</td>
<td>4.8%</td>
</tr>
</tbody>
</table>

• Throughput overhead:

<table>
<thead>
<tr>
<th>% GETs</th>
<th>% SETs</th>
<th>Throughput Slowdown</th>
</tr>
</thead>
<tbody>
<tr>
<td>96.7%</td>
<td>3.3%</td>
<td>1.5%</td>
</tr>
<tr>
<td>50%</td>
<td>50%</td>
<td>3%</td>
</tr>
<tr>
<td>10%</td>
<td>90%</td>
<td>3.7%</td>
</tr>
</tbody>
</table>

• Memory overhead: 200KB for each queue
Cache OS
Cache Clusters are Static and Difficult to Manage

• Caches are static
  – Cache allocation is static, not application aware
  – Applications are statically split into separate cache server pools
  – Cache server pool sizes determined arbitrarily

• Cache clusters are hard to evaluate and optimize
  – Heterogeneous cache resources (memory, SSD, NV RAM) managed and allocated independently
  – No way to define QoS/prioritization for applications
Vision: Cache OS

- Memcached Server
- Memcached Server
- SSD Cache Server
- SSD Cache Server

Client

Cache OS
Research Questions

• Maximizing performance given QoS constraints for multiple applications with heterogeneous hardware
• Minimize TCO given QoS
  – Utilize Flash instead of memory
• Automatically classifying application characteristics for optimized multi-tenancy
Thank You!
Appendix
Example: Application 19

Graph showing Hitrate over Minutes for Application 19, Slab Class 1.
Comparison with “Facebook LRU”

<table>
<thead>
<tr>
<th>Application</th>
<th>Original Hitrate</th>
<th>Facebook Hitrate</th>
<th>Cliffhanger + LRU Hitrate</th>
<th>Cliffhanger + Facebook Hitrate</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>94.2%</td>
<td>94.4%</td>
<td>98%</td>
<td>98%</td>
</tr>
<tr>
<td>5</td>
<td>95.8%</td>
<td>96.5%</td>
<td>97.8%</td>
<td>98.1%</td>
</tr>
</tbody>
</table>
Related Work

• CPU cache partitioning for performance cliffs
  – Talus: Beckmann et al [HPCA ‘15]

• Optimizing memory allocation across applications based on hitrate curves
  – Mimir: Saemundsson et al [SOCC ‘14]

• Memcached client
  – McRouter: Likhtarov et al [Facebook blog ‘14]

• Rebalancing slabs to reduce slab calcification
  – Twitter: Rajashekar et al [Twitter blog ‘12]
  – Facebook: Nishtala et al [NSDI ’13]
Cache OS Functionality

- Policy API
- Policy Enforcement
- Instrumentation
Policy API

• Allows operators to set QoS policies
  – Min/max hit rate/latency
  – Fair queueing
  – Prioritization

• Exposes operators to cost/benefit of resources
  – Cost per hit
  – Cost per hit/bit

• Operators can provide hints to prioritize total system performance
  – For example: latency of missed object
Policy Enforcement

• Enforce application prioritization and QoS
  – Control frequency of requests from different applications to different servers to simulate different queue sizes

• Leverage unique properties of hardware
  – E.g.: route large, infrequent requests to Flash
Instrumentation

- Add / remove cache resources automatically based on application requirements
- Load balance requests across servers
Log Structured Memory is Still Greedy

<table>
<thead>
<tr>
<th>Original Hitrate</th>
<th>Log-structured Hitrate</th>
<th>Optimized Hitrate</th>
</tr>
</thead>
<tbody>
<tr>
<td>98.4%</td>
<td>98.6%</td>
<td>99.4%</td>
</tr>
</tbody>
</table>

Table 2: Hit rates of Application 5 under log-structured memory and optimized slab classes.
## Algorithms are Complementary
(Memcachier’s Application 19)

<table>
<thead>
<tr>
<th>Slab Class</th>
<th>Original Hitrate</th>
<th>Cliff Scaling Hitrate</th>
<th>Hill Climbing Hitrate</th>
<th>Combined Algorithm Hitrate</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>38.1%</td>
<td>43.3%</td>
<td>95.3%</td>
<td>98.3%</td>
</tr>
<tr>
<td>1</td>
<td>37.3%</td>
<td>41.1%</td>
<td>67.4%</td>
<td>69.1%</td>
</tr>
<tr>
<td>Total Hitrate</td>
<td>37.3%</td>
<td>41.3%</td>
<td>70.3%</td>
<td>72.1%</td>
</tr>
</tbody>
</table>