ElCached: Elastic Multi-Level Key-Value Cache

Rahman Lavaee,
Stephen Choi, Yang-Suk Kee,
Chen Ding

November 1st, Savannah, GA
Web Caching
Resource Provisioning for Memcached
Resource Provisioning for Memcached

- The service provider must wisely allocate the resource to guarantee each tenant’s SLA, while minimizing TCO.
Elastic Resource Provisioning

• Optimal resource provisioning requires **elasticity**
  • capability to adapt to workload changes by dynamic resource provisioning.
Multitenant Resource Provisioning

• The problem becomes more complex as
  • more tenants are added to the system.
  • more web caching layers are used.
MlCached [HotCloud’16]

- A multi-level **key-value** caching system.
  - L1: DRAM-based Memcached
  - L2: exclusive NAND-flash-based key-value cache (SSD).

- MlCached implements direct key-to-PBA mappings on SSD.
  - Independent resource provisioning
- In this work, we extend MlCached by adding the elasticity feature.
Independent Resource Provisioning in MiCached

- MiCached implements direct key-to-PBA mappings on SSD.
- This removes the need for storing redundant key-to-LBA mapping tables in memory
Performance Model
Latency Model

\[ \text{Lat} = l_m + l_s \cdot M_m + l_{db} \cdot M_s \]

<table>
<thead>
<tr>
<th>$l_m$</th>
<th>Latency of Memcached server</th>
</tr>
</thead>
<tbody>
<tr>
<td>$l_s$</td>
<td>Latency of SSD</td>
</tr>
<tr>
<td>$l_{db}$</td>
<td>Latency of backend DB server</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$M_m$</th>
<th>Miss rate of Memcached</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M_s$</td>
<td>Miss rate of SSD</td>
</tr>
</tbody>
</table>
Cost Model

\[ \text{Cost} = c_m \cdot p_m + c_s \cdot p_s \]

<table>
<thead>
<tr>
<th>$c_m$</th>
<th>Size of DRAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c_s$</td>
<td>Size of SSD</td>
</tr>
<tr>
<td>$p_m$</td>
<td>Price per unit of DRAM</td>
</tr>
<tr>
<td>$p_s$</td>
<td>Price per unit of SSD</td>
</tr>
</tbody>
</table>
Latency Based on Miss Ratio Curve

- The key to optimal resource provisioning is to find the miss ratio curve (MRC).

\[ Lat = l_m + l_s \cdot mr(c_m) + l_{db} \cdot mr(c_s) \]
How to compute the MRC?
Reuse Distance

- We use the reuse distance theory to compute the miss ratio curve.
- Reuse distance is the number of distinct memory locations accessed between two consecutive uses of the same memory location.

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>b</th>
<th>a</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>
Reuse Distance Histogram

- The reuse distance information is best represented by the reuse distance histogram, which shows the frequency for every reuse distance.

- The MRC can be computed from this histogram.

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>b</th>
<th>a</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

![Reuse Distance Histogram](image)

![MRC Computation](image)
Reuse Distance Computation

• Olken tree [Olken 1981]
• Approximate reuse distance [Ding+ 2001]
• Footprint estimation [Xiang+ 2011]
• Stack counters [Wires+ 2014]
Reuse Distance Computation

• Olken tree [Olken 1981] ✓
• Approximate reuse distance [Ding+ 2001]
• Footprint estimation [Xiang+ 2011]
• Stack counters [Wires+ 2014]
Reuse Distance for Memcached

• Memcached distributes items among different slab classes, according to their sizes.
• Slab allocation is done during the cold start.
LRU Replacement in Memcached

• Once the Memcached system reaches its memory limit, LRU replacement is done separately for every slab class.
Slab-Aware Reuse Distance Profiling

• Rather than analyzing the whole Memcached system in a single reuse distance model, we model each slab class separately.

• We compose the MRCs from different slab classes.
How To Solve Resource Provisioning?
Resource Provisioning as a Linear Program

• The resource provisioning problem can be described in one of the two ways.
  • Minimize $Cost$ such that $Lat \leq SLA$.
  • Minimize $Lat$ such that $Cost \leq TCO$. 
Resource Provisioning as a Linear Program

• The resource provisioning problem can be described in one of the two ways.
  • Minimize $Cost$ such that $Lat \leq SLA$.
  • Minimize $Lat$ such that $Cost \leq TCO$. 
Resource Provisioning as a Linear Program

• The resource provisioning problem can be described in one of the two ways.
  • Minimize Cost such that $Lat \leq SLA$.
  • Minimize $Lat$ such that $Cost \leq TCO$.

• Cost is already linear in terms of DRAM/SSD capacities.
Resource Provisioning as a Linear Program

• The resource provisioning problem can be described in one of the two ways.
  • Minimize Cost such that \( \text{Lat} \leq \text{SLA} \).
  • Minimize \( \text{Lat} \) such that \( \text{Cost} \leq \text{TCO} \).

• Cost is already linear in terms of DRAM/SSD capacities.

•Latency is linear only in terms of the miss ratio function.
Resource Provisioning as a Linear Program

• We observe that the miss ratio curves in our workloads are always convex.
• We formulate the miss ratio curve using linear constraints.
Evaluation

• We compare ElCached against a proportional approach that fixes the ratio between DRAM and SSD capacities to 1:4 (Pareto principle).
Evaluation

• We compare ElCached against a proportional approach that fixes the ratio between DRAM and SSD capacities to 1:4 (Pareto principle).

<table>
<thead>
<tr>
<th>Cost ($/GB)</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>DRAM</td>
<td>10</td>
</tr>
<tr>
<td>SSD</td>
<td>0.68</td>
</tr>
<tr>
<td>Memcached</td>
<td>100μs</td>
</tr>
<tr>
<td>KVD</td>
<td>200μs</td>
</tr>
<tr>
<td>Back-end DB</td>
<td>10ms</td>
</tr>
</tbody>
</table>
Evaluation

• We compare ElCached against a proportional approach that fixes the ratio between DRAM and SSD capacities to 1:4 (Pareto principle).

<table>
<thead>
<tr>
<th>cost ($/GB)</th>
<th>latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>DRAM</td>
<td>Memcached</td>
</tr>
<tr>
<td>SSD</td>
<td>KVD</td>
</tr>
<tr>
<td></td>
<td>Back-end DB</td>
</tr>
</tbody>
</table>

• Workloads:
  • Zipfian key distribution with $\alpha = 1.15$
  • Exponential key distribution with $\lambda = 10^{-6}$
  • Both workloads issue 800 million requests to a range of 4 billion keys.
Miss Ratio Prediction Accuracy

• Mean relative error on Zipfian workload: 4%
Experiment 1: Elastic Resource Provisioning

• For each workload
  • For each latency limit, we find the minimum cost resource provisioning.
Experiment 1: Elastic Resource Provisioning

- For each workload
  - For each latency limit, we find the minimum cost resource provisioning.

- Elastic saves cost by around 60% for both workloads.
Experiment 2: Multitenant Resource Provisioning

• First, we use the proportional scheme to partition a fixed 3GB memory between the two tenants.

![Diagram showing latency vs DRAM partition for two tenants](image_url)

<table>
<thead>
<tr>
<th>Proportional</th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>DRAM</td>
<td>1.63</td>
<td>1.37</td>
</tr>
<tr>
<td>Lat.</td>
<td>0.71</td>
<td>0.20</td>
</tr>
<tr>
<td>Cost</td>
<td>7.20</td>
<td>5.41</td>
</tr>
</tbody>
</table>
Experiment 2: Multitenant Resource Provisioning

• Then, we use ElCached to find the latency-optimal DRAM/SSD partitioning.

<table>
<thead>
<tr>
<th>ELASTIC</th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>DRAM</td>
<td>0.46</td>
<td>1.17</td>
</tr>
<tr>
<td>Lat.</td>
<td>0.67</td>
<td>0.20</td>
</tr>
<tr>
<td>Cost</td>
<td>5.30</td>
<td>4.01</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PROPORTIONAL</th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>DRAM</td>
<td>1.63</td>
<td>1.37</td>
</tr>
<tr>
<td>Lat.</td>
<td>0.71</td>
<td>0.20</td>
</tr>
<tr>
<td>Cost</td>
<td>7.20</td>
<td>5.41</td>
</tr>
</tbody>
</table>
Experiment 2: Multitenant Resource Provisioning

• Elasticity improves
  • tenant 1’s latency by 5%,
  • both tenants’ cost by 26%, and
  • total memory consumption by 46%.
Summary

• ElCached is a multi-level key-value caching system.
• It uses a reuse distance profiler to estimate the miss rate curve across all capacity limits.
• It reduces the total cost by up to around 60% compared to a proportional scheme.
• Multi-tenant experiment indicates that we can improve latency, cost, and total DRAM usage, compared to the proportional scheme.
Thank You

Any Questions?