Segcache: a memory-efficient and scalable in-memory key-value cache for small objects

Juncheng Yang  Yao Yue  Rashmi Vinayak
Carnegie Mellon University, Twitter
In-memory key-value caches

Reduce latency
Increase throughput and scalability
Reduce backend load

services
- Notification
- Tweet
- Ad
- Rate limiter

cache
- Memcached

backend
- RocksDB
- MySQL
- Tensorflow
Today’s in-memory caching systems

Have significant room for improvement

- Memory efficiency
  - TTL and expiration
  - Huge per-object metadata
  - Memory fragmentation

- Throughput and scalability
  - Tradeoff between efficiency and throughput or scalability
**TTL and expiration**

**Time-to-live (TTL)**
- TTL is set during object write
- Expired objects cannot be served
- Short TTLs are widely used in production

**TTL usages**
- Reduce stale data (cache writes are best-effort)
- Periodic refresh (e.g. ML predictions)
- Implicit deletions (e.g. limiters, GDPR)

**Impact of TTL**
- Reduce effective working set size
- Removing expired objects is critical
TTL and expiration: takeaway

Timely removal of expired objects is critical for memory efficiency

• expiration: remove objects that cannot be used in the future
• eviction: remove objects that could potentially be used in the future
## Existing solutions for TTL expiration

**Efficient:** low overhead  
**Sufficient:** can remove all or most expired objects

<table>
<thead>
<tr>
<th>Category</th>
<th>Technique</th>
<th>Efficient</th>
<th>Sufficient</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lazy expiration</td>
<td>Delete upon re-access</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td></td>
<td>Check LRU tail</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Proactive expiration</td>
<td>Scanning</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Sampling</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td></td>
<td>Transient object pool</td>
<td>✓</td>
<td>✗</td>
</tr>
</tbody>
</table>

### Color: expiration time
- Blue: expire at $T$  
- Yellow: $T+20$  
- Green: $T+9600$  
- Red: $T+86400$

- How can I find expired objects?

- Either not efficient or not sufficient
Today’s in-memory caching systems:

• Memory efficiency
  - Cannot efficiently and timely remove expired objects
  - Have huge per-object metadata (56 bytes in Memcached), but objects are small (10s-100s bytes)
  - Suffer from memory fragmentation

• Throughput and scalability
  - Tradeoff between efficiency and throughput or scalability

<table>
<thead>
<tr>
<th></th>
<th>MICA</th>
<th>MemC3</th>
<th>Memshare</th>
<th>LHD</th>
<th>Hyperbolic</th>
<th>pRedis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory efficiency</td>
<td>❌</td>
<td>❌</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Throughput/scalability</td>
<td>✔️</td>
<td>✔️</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
</tr>
</tbody>
</table>
Segcache: a memory-efficient and scalable in-memory key-value cache for small objects
Segcache overview

Segcache: segment-structured cache

High memory efficiency
• Efficient and sufficient TTL expiration
• Tiny object metadata (5-byte)
• Almost no memory fragmentation
• Merge-based eviction for low miss ratio

High performance
• High throughput
• Close-to-linear scalability

Expect to enter Twitter production this year
Segcache design

TTL buckets

segment chain

write

segment: a small fixed-size log storing objects of similar TTLs

object store

segment 1 (S1)

segment 2 (S2)

... 

segment M

row: hash bucket

bucket info (shared)

object info

hash table

read
Design principles
Design principle 1: Maximize metadata approximation and sharing

Group objects into segments to approximate and share metadata
Segment: a small fixed-size log storing objects of similar TTLs

Memcached object store

Segcache object store

hash table
Design principle 2: Be proactive, don’t be lazy

*Efficiently and proactively remove expired objects*

- Objects in a segment share creation time and TTL => expire at the same time
- Segments in a chain have same TTL with sorted creation time => examine the first segment only
- Background thread scans TTL buckets (small array of metadata) => efficient and proactive expiration
Design principle 3: Perform macro management

- Manage segments (groups of objects), not objects
- Perform less bookkeeping in batched sequential fashion with high throughput
- Achieve a close-to-linear scalability

Expiration and eviction happen on the segment level

Only segment chain changes needs locking
In the paper (not covered in the talk)

• Segment homogeneity
• Merge-based eviction
  - Approximate and smoothed frequency counter
    ◦ Low overhead
    ◦ Burst-resistant
    ◦ Scan-resistant
    ◦ Eviction-friendly
Evaluation

Implemented on Pelikan
  • Twitter’s open-source caching framework

Setup
  • Five systems (research + production)
    - Production
    - Memcached and Memcached + scanning
    - LHD + sampling
    - Hyperbolic + sampling
    - Segcache
  • Five production traces
  • Twitter production fleet
Evaluation: memory efficiency

Reduce memory footprint by

- 40-90% compared to production
  - 60% on Twitter’s largest cache cluster
- 22-60% compared to state-of-the-art

Metric: relative cache size to achieve production miss ratio
Evaluation: throughput and scalability

**Higher is better**

**Single-thread**
- similar to production
- up to 40% higher than Memcached
- significantly higher than the rest

**Multi-thread**
- 8x improvement with 24 threads
Summary

Segcache: segment-structured cache, groups objects into segments for

- high memory efficiency and high performance
  - Efficient proactive TTL expiration
  - Object metadata reduction using metadata approximation and sharing
  - Almost no memory fragmentation
  - Small miss ratio/memory footprint with merge-based eviction
  - High throughput and high scalability using macro management

Traces: https://www.github.com/twitter/cache-trace
Code: https://www.github.com/thesys-lab/segcache
Production code: https://www.github.com/twitter/pelikan
Thank you!

Acknowledgement: Jack Kosaian from CMU, Rebecca Isaacs, Xi Wang, Dan Luu, Brian Martin from Twitter, IOP, cache, HWEng team from Twitter, Parallel Data Lab at CMU, Siobiosys lab at Emory University.