Cost-based Memory Partitioning and Management in Memcached

Damiano Carra
Computer Science Department
University of Verona

Pietro Michiardi
Network and Security Department
Eurecom - Sophia Antipolis
Memcached

- Key-value store
  - Simple APIs: `set`, `get`, `delete`, ...

- Common component of Web architectures
  - Caching layer

- All data is kept in **RAM**
  - Fast response time
Memory management: Classes and Slabs

- Memcached divides the objects into *classes*
  - Based on their sizes

- Example: 3 classes of objects
  - Small
  - Medium
  - Large
Memory management: Classes and Slabs

- Memcached divides the objects into *classes*
  - Based on their sizes

- Example: 3 classes of objects
  - Small
  - Medium
  - Large

- Memory is divided into blocks called *slabs*
  - Default size of a slab: 1 MB

- A slab contains a variable number of objects
  - Many small ones
  - Some medium
  - Few large
Slabs are assigned to classes upon request
**Slabs** are assigned to *classes* upon request
**Slabs** are assigned to *classes* upon request

available memory (divided into slabs)

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

set()  

```
set()
```

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

set()  

```
set()
```
Slabs are assigned to classes upon request.

available memory (divided into slabs)

```
<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>
```

```
set() |
```

```
set() |
```

```
set() |
```

```
set() |
```
Slabs are assigned to classes upon request.
**Slabs** are assigned to *classes* upon request

- After many requests...

  available memory (divided into slabs)

- What happens when the cache receives a new $\text{set}(\Box)$?
**Slabs** are assigned to *classes* upon request

- After many requests...

  available memory (divided into slabs)

  ![Available Memory Diagram]

- What happens when the cache receives a new \( \text{set}() \)?
  - **LRU** eviction *within* the class

  ![LRU Diagram]
Key challenges

- **How** the memory can be divided among the classes?
  - Static vs dynamic assignment

- **What** are the criteria used to assign the memory?
  - Hit-ratio vs cost-hit-ratio
Outline

Background

Design

Evaluation

Conclusions
Miss-ratio curves (MRC)

- Given a trace and an eviction policy, we can compute the MRC
  - What would be the miss ratio if the class had that specific memory assigned?
- MRC can be found with a single pass of the traces
  - Mattson stack algorithm
Optimal offline slab assignment

- **Goal**: Minimize the miss ratio

- **Assumptions**
  - MRC are concave
  - Memory is divided into finite number of blocks

---

**Algorithm 1 Optimal Off-line Slab Assignment**

1. **Input**: Miss ratio curves for all classes
2. **Input**: Number of available slabs
3. 
4. **repeat**
5. Sort classes by their miss ratio difference
6. Assign a slab to the class with the highest difference
7. **until** All slabs are assigned
Online slab assignment

- The offline optimal slab assignment can be used as a reference
  - It can be computed once we have the whole trace

- We provide a heuristic for the online assignment

- Before showing the heuristic, we consider the object costs
A new interface

- The interface \( \text{set}(k,v) \) implies that all the objects have the same cost or weight

- But some objects can be more difficult to obtain
  - E.g., simply because they are larger
  - or because it takes more time to retrieve them from the DB
    - complex query

- We have added a new interface: \( \text{set}(k,v,c) \)
  - The application can associate a cost (or weight) to the object
Slab Allocation Scheme (SAS)

- We define an observation epoch
  - E.g., when the system has experienced $M$ misses
- During each epoch, we collect, for each classes:
  - The number of (weighted) misses
  - The number of (weighted) requests

- High number of misses $\Rightarrow$ The class is suffering
- Low number of miss ratio per assigned slab $\Rightarrow$ The class can lose a slab
Settings

- We use a representative Web architecture
  - Application server, cache, database

- Trace driven experiment
  - Traces collected by a major CDN operator
  - The objects in the trace are stored in the database
  - The cost of retrieving the objects is stored in the application server
  - The sequence of the requests is issued by a client
About the traces

- The costs of the objects are not correlated with the sizes
  - Correlation coefficient $\rightarrow 0.013$
Results

- We compute the optimal cost hit ratio with the offline algorithm.

- Our scheme progressively adapts the slab assignment:
  - It converges to the optimal.
  - While Memcached (static assignment) is far from optimal.
Results: slab assignment

- Snapshot taken when half of the requests has been processed by the client

- Comparison with the
  - Optimal assignment (offline)
  - Static assignment (Memcached)
Results: Different Costs

- All objects have the same cost
- The cost of the object is given by its size
Outline

- Background
- Testbed
- Results

Conclusions
Discussion

- Synthetic traces
  - We tested different traces derived from the literature

- Calcification
  - What happens if the statistical properties of the requested objects change?
  - The dynamic nature of the scheme solves a problem known in the literature as calcification
Conclusions

- **Memcached**: in-memory key value store largely adopted...
  - Facebook, Twitter
- **.. with a specific memory allocation scheme**
  - Slabs, Classes, LRU within the same class
- **We provide a scheme for dynamically assigning the slabs to the different classes**
  - Trying to minimize the number of (weighted) misses
  - Many solutions proposed in the literature not applicable to Memcached due to its memory allocation scheme
Thanks!