An In-Memory Object Caching Framework with Adaptive Load Balancing

Yue Cheng (Virginia Tech)
Aayush Gupta (IBM Research – Almaden)
Ali R. Butt (Virginia Tech)
In-memory caching in datacenters

Local deployment

Cloud deployment
In-memory caching in datacenters

Local deployment

Cloud deployment

Web app servers

Client library

Network

DB Query

Persistent storage tier
e.g., MySQL
In-memory caching in datacenters

Local deployment

Cloud deployment

In-memory caching tier
- e.g., Memcached

Persistent storage tier
- e.g., MySQL

Web app servers

Network

Client library

get(key)

set(key)

Cache miss

set(key)
In-memory caching is desirable

- Offers high performance
- Enables quick deployment
- Provides ease of use
- Supports elastic scale-out
In-memory caching is desirable

• Offers high performance
• Enables quick deployment
• Provides ease of use
• Supports elastic scale-out

• **Problem:** Load imbalance impacts performance
Access load imbalance

Per-client throughput (QPS in thousands)

Workload skewness (Zipfian constant)

- Ideal balance
- high imbalance

95% GET, 5% SET, Zipfian, 20 cache servers
Access load imbalance

Per-client throughput (QPS in thousands)

Workload skewness (Zipfian constant)

- unif
- 0.4
- 0.8
- 0.9
- 0.99
- 1.01
- 1.1

95% GET, 5% SET, Zipfian, 20 cache servers

Ideal balance → high imbalance

Key popularity distribution: different Zipfian constant

0.5
0.9
1.1

* http://www.percona.com/blog/2012/05/09/new-distribution-of-random-generator-for-sysbench-zipf/*
Access load imbalance

Per-client throughput (QPS in thousands)

Workload skewness (Zipfian constant)

- **unif**
- **0.4**
- **0.8**
- **0.9**
- **0.99**
- **1.01**
- **1.1**

- **Ideal balance**
- **high imbalance**

95% GET, 5% SET, Zipfian, 20 cache servers
Access load imbalance

![Bar chart showing per-client throughput (QPS in thousands) for different workload skewness (Zipfian constant). The x-axis represents various skewness values from 0.4 to 1.1, and the y-axis shows throughput from 0 to 16. The chart indicates ideal balance and high imbalance.

- Ideal balance: 95% GET, 5% SET, Zipfian, 20 cache servers
- High imbalance: > 60% decrease in throughput]
Access load imbalance

![Graph showing the relationship between workload skewness and per-client throughput and latency. The graph illustrates that as the workload skewness increases, the per-client throughput decreases, and the 99th percentile latency increases. There is an ideal balance between these two metrics.]

95% GET, 5% SET, Zipfian, 20 cache servers
Access load imbalance

Great opportunity for performance improvement

Workload skewness (Zipfian constant)

Ideal balance  high imbalance

95% GET, 5% SET, Zipfian, 20 cache servers
Our contribution: **MBal**

Revisiting in-memory cache design

A holistic in-memory caching framework with adaptive **Multi-phase load Balancing**

- Synthesizes different load balancing techniques
  - Key replication
  - Server-local cachelet migration
  - Coordinated cachelet migration

- Improves scale-up gains
- Mitigates load imbalance
Outline

MBal cache design
MBal load balancer design
Evaluation
Related work
Outline

MBal Cache Design
MBal load balancer design
Evaluation
Related work
A typical in-memory cache design

Worker 1  Worker 2  ...  Worker N

Shared in-memory data structure

In-memory data

CPU

DRAM
MBal: Fine-grained resource partitioning
MBal cachelet: Resource encapsulation

- **Cachelet**
  - Encapsulates resources
  - Avoids lock contention
Key-to-cachelet mapping

MBal client

Query

Client side
Server side

MBal cache
Key-to-cachelet mapping

1. Compute VN # with hash

Key ring

Client side
Server side

MBal client

MBal cache
Key-to-cachelet mapping

1. Compute VN # with hash

2. Map VN # to Cachelet ID

Client side
Server side

MBal cache
Key-to-cachelet mapping

1. Compute VN # with hash
2. Map VN # to Cachelet ID
3. Map Cachelet ID to the worker thread

Key ring
hash(key)

VN1
VN2
VN_N
VN_N-1

key, (value*)
cachelets

C1 C2 C3

Client side
Server side

MBal client

MBal cache
Outline

MBal cache design

MBal Multi-Phase Load Balancer

Evaluation

Related work
Phase 1: Key replication

- **TRIGGER?**
  - EWMA access > threshold

- **ACTION?**
  - Randomly pick a shadow server
  - Replicate hot keys
  - Proportional sampling

- **FEATURES?**
  - Fine-grained
  - Temporary

* SPORE [SoCC’13]
Phase 2: Server-local cachelet migration

- **TRIGGER?**
  - # hot keys > REPL\textsubscript{HIGH}
  - Enough local headroom

- **ACTION?**
  - Migrate/swap cachelet(s) within a server
  - ILP

- **FEATURES?**
  - Coarse-grained
  - Temporary
Phase 2: Server-local cachelet migration

- **TRIGGER?**
  - \# hot keys > REPL_{HIGH}
  - Enough local headroom

- **ACTION?**
  - Migrate/swap cachelet(s) within a server
  - ILP

- **FEATURES?**
  - Coarse-grained
  - Temporary

Server-local migration
Phase 3: Coordinated cachelet migration

- **TRIGGER**?
  - # hot keys > $\text{REPL}_{\text{HIGH}}$
  - Not enough local headroom

- **ACTION**?
  - Migrate/swap cachelet(s) across servers
  - ILP

- **FEATURES**?
  - Coarse-grained
  - Permanent
Phase 3: Coordinated cachelet migration

- **TRIGGER?**
  - # hot keys > REPL
  - Not enough local headroom

- **ACTION?**
  - Migrate/swap cachelet(s) across servers
  - ILP

- **FEATURES?**
  - Coarse-grained
  - Permanent
Phase 3: Coordinated cachelet migration

- **TRIGGER?**
  - # hot keys > REPL_{HIGH}
  - Not enough local headroom

- **ACTION?**
  - Migrate/swap cachelet(s) across servers
  - ILP

- **FEATURES?**
  - Coarse-grained
  - Permanent
Client-side mapping change

- Phase 2: Server-local cachelet migration
  - Clients are informed of cachelet migration when cache home worker receives requests about that migrated cachelet

- Phase 3: Coordinated cachelet migration
  - Once migration is done, source worker informs coordinator about the mapping change
  - Clients ping coordinator periodically
**MBal: Cost/benefit trade-offs**

- **Benefit:** fast fix for hot keys
- **Cost:** metadata; space; n/w transfer

**P1: Key replication**
MBal: Cost/benefit trade-offs

Benefit: fast fix for hot keys
Cost: metadata; space; n/w transfer

P1: Key replication
Cost: metadata
Benefit: fast fix for hot partitions

P2: Server-local cachelet migration
MBal: Cost/benefit trade-offs

P1: Key replication
Cost: metadata
Benefit: fast fix for hot partitions

P2: Server-local cachelet migration
Cost: metadata; space; n/w transfer
Benefit: fast fix for hot keys

P3: Coordinated cachelet migration
Cost: metadata; bulk transfer n/w
Benefit: global load balancing
Outline

MBal cache design
MBal load balancer design
Evaluation
Related work
Methodology

• Scale-up cache performance tests
  – Local testbed (8-core server)
  – Single instance

• End-to-end load balancer evaluation
  – 20-VM cluster (Amazon EC2, c3.large)
MBal evaluation – micro-benchmark

- 8-core 2.5GHz, 2×10MB L3 LLC, 64GB DRAM
- Uniform workload, 100% GET, 10B key 20B value
- Without network
MBal evaluation – micro-benchmark

- 8-core 2.5GHz, 2×10MB L3 LLC, 64GB DRAM
- Uniform workload, 100% GET, 10B key 20B value
- Without network

Throughput (QPS in millions)

Throughput (QPS in millions)

Number of threads

Throughput (QPS in millions)

MBal, MBal no NUMA, Mercury, Memcached
MBal evaluation – micro-benchmark

- 8-core 2.5GHz, 2×10MB L3 LLC, 64GB DRAM
- Uniform workload, 100% GET, 10B key 20B value
- Without network

Graph showing throughput (QPS in millions) vs. number of threads for MBal, MBal no NUMA, Mercury, and Memcached. The graph indicates performance improvements and comparisons among the different systems.
MBal evaluation – micro-benchmark

✓ MBal uses fine-grained cachelet design
✓ MBal eliminates bucket-level lock contention
MBal evaluation – micro-benchmark

- 8-core 2.5GHz, 2×10MB L3 LLC, 64GB DRAM
- Uniform workload, 100% SET, 10B key 20B value
- Without network
MBal evaluation – micro-benchmark

- 8-core 2.5GHz, 2×10MB L3 LLC, 64GB DRAM
- Uniform workload, 100% SET, 10B key 20B value
- Without network
MBal evaluation – micro-benchmark

- 8-core 2.5GHz, 2×10MB L3 LLC, 64GB DRAM
- Uniform workload, 100% SET, 10B key 20B value
- Without network
MBal evaluation – micro-benchmark

- 8-core 2.5GHz, 2×10MB L3 LLC, 64GB DRAM
- Uniform workload, 100% SET, 10B key 20B value
- Without network

![Throughput Graph](image)
MBal evaluation – micro-benchmark

✓ MBal eliminates global cache lock contention!

Throughput (QPS in millions)

Number of threads

0.00 2.00 4.00 6.00 8.00 10.00 12.00 14.00 16.00

MBal  MBal no NUMA  Mercury  Memcached

8-core 2.5GHz, 2 × 10MB L3 LLC, 64GB DRAM

Uniform workload, 100% SET, 10B key 20B value

Throughput (QPS in millions)

Number of threads

0.00 2.00 4.00 6.00 8.00 10.00 12.00 14.00 16.00

MBal  MBal no NUMA  Mercury  Memcached

Without network

44
# End-to-end load balancer evaluation

<table>
<thead>
<tr>
<th>Workload</th>
<th>Characteristics</th>
<th>Application scenario</th>
</tr>
</thead>
<tbody>
<tr>
<td>Workload A</td>
<td>100% read, Zipfian</td>
<td>User account status info</td>
</tr>
<tr>
<td>Workload B</td>
<td>95% read, 5% update, hotspot (95% ops on 5% data)</td>
<td>Photo tagging</td>
</tr>
<tr>
<td>Workload C</td>
<td>50% read, 50% update, Zipfian</td>
<td>Session store recording actions</td>
</tr>
</tbody>
</table>

Amazon EC2, us-west-2b, Clients on 36 instances (c3.2xlarge), MBal caches on 20-VM cluster (c3.large)
Load balancer evaluation

Memcached is unable to sustain write-intensive workload

<table>
<thead>
<tr>
<th>Workload</th>
<th>Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Workload A</td>
<td>100% read, Zipfian</td>
</tr>
<tr>
<td>Workload B</td>
<td>95% read, 5% update, hotspot</td>
</tr>
<tr>
<td>Workload C</td>
<td>50% read, 50% update, Zipfian</td>
</tr>
</tbody>
</table>
Load balancer evaluation

<table>
<thead>
<tr>
<th>Workload</th>
<th>Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Workload A</td>
<td>100% read, Zipfian</td>
</tr>
<tr>
<td>Workload B</td>
<td>95% read, 5% update, hotspot</td>
</tr>
<tr>
<td>Workload C</td>
<td>50% read, 50% update, Zipfian</td>
</tr>
</tbody>
</table>

Memcached is unable to sustain write-intensive workload

Memcached

MBal, w/o load balancer

Ideal balance
Load balancer evaluation

<table>
<thead>
<tr>
<th>Workload</th>
<th>Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Workload A</td>
<td>100% read, Zipfian</td>
</tr>
<tr>
<td>Workload B</td>
<td>95% read, 5% update, hotspot</td>
</tr>
<tr>
<td>Workload C</td>
<td>50% read, 50% update, Zipfian</td>
</tr>
</tbody>
</table>

Memcached is unable to sustain write-intensive workload

Memcached

MBal, w/o load balancer

MBal, all phases

Ideal balance

90th %ile latency (ms)

0 100 200 300 400 500
Runtime (seconds)

Workload A  Workload B  Workload C
Load balancer evaluation

<table>
<thead>
<tr>
<th>Workload</th>
<th>Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Workload A</td>
<td>100% read, Zipfian</td>
</tr>
<tr>
<td>Workload B</td>
<td>95% read, 5% update, hotspot</td>
</tr>
<tr>
<td>Workload C</td>
<td>50% read, 50% update, Zipfian</td>
</tr>
</tbody>
</table>
Load balancer evaluation

<table>
<thead>
<tr>
<th>Workload</th>
<th>Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Workload A</td>
<td>100% read, Zipfian</td>
</tr>
<tr>
<td>Workload B</td>
<td>95% read, 5% update, hotspot</td>
</tr>
<tr>
<td>Workload C</td>
<td>50% read, 50% update, Zipfian</td>
</tr>
</tbody>
</table>

- **Workload A**: 100% read, Zipfian
- **Workload B**: 95% read, 5% update, hotspot
- **Workload C**: 50% read, 50% update, Zipfian

![Graph showing latency and runtime for different load balancer configurations.](image)

- **Memcached**: All 3 phases are triggered.
- **MBal, w/o load balancer**: 35% reduction in latency.
- **MBal, all phases**: Ideal balance.
Load balancer evaluation

<table>
<thead>
<tr>
<th>Workload</th>
<th>Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Workload</td>
<td>100% read, Zipfian</td>
</tr>
<tr>
<td>Workload B</td>
<td>95% read, 5% update, hotspot</td>
</tr>
<tr>
<td>Workload C</td>
<td>50% read, 50% update, Zipfian</td>
</tr>
</tbody>
</table>

only Phase 2 is needed

90th %ile latency (ms)

Workload B

Runtime (seconds)

Memcached

MBal, w/o load balancer

MBal, all phases

Ideal balance

only Phase 2 is needed
Load balancer evaluation

<table>
<thead>
<tr>
<th>Workload</th>
<th>Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Workload A</td>
<td>100% read, Zipfian</td>
</tr>
<tr>
<td>Workload B</td>
<td>95% read, 5% update, hotspot</td>
</tr>
<tr>
<td>Workload C</td>
<td>50% read, 50% update, Zipfian</td>
</tr>
</tbody>
</table>

A combination of Phase 2 & 3 is triggered when MBal, w/o load balancer is applied.

- **Workload C**
  - MBal, w/o load balancer: 23%
  - MBal, all phases: Ideal balance
Summary of results

• MBal fine-grained partitioning design
  – $2 \times$ more QPS for GETs
  – $62 \times$ more QPS for SETs

• MBal multi-phase load balancer
  – 35% lower tail latency
  – 20% higher throughput
Summary of results

- MBal fine-grained partitioning design
  - $2 \times$ more QPS for GETs
  - $62 \times$ more QPS for SETs

- MBal multi-phase load balancer
  - 35% lower tail latency
  - 20% higher throughput

Improves “BANG for the buck”
Outline

MBal cache design
MBal load balancer design
Evaluation
Related work
Related work

• High performance in-memory KV store
  – Masstree [EuroSys’12], MemC3 [NSDI’12], MICA [NSDI’14]

• Storage load balancing
  – DHT (Pastry [Middleware’01], CFS [SOSP’01], Chord [SIGCOMM’01]), Proteus [ICDCS’13]

• Access load balancing
  – SmallCache [SoCC’11], Chronos [SoCC’12], SPORE [SoCC’13], Streaming Analytics [Feedback’14]
Related work

• High performance in-memory KV store
  – Masstree [EuroSys’12], MemC3 [NSDI’12], MICA [NSDI’14]

• Storage load balancing
  – DHT (Pastry [Middleware’01], CFS [SOSP’01], Chord [SIGCOMM’01]), Proteus [ICDCS’13]

• Access load balancing
  – SmallCache [SoCC’11], Chronos [SoCC’12], SPORE [SoCC’13], Streaming Analytics [Feedback’14]
Related work

• High performance in-memory KV store
  – Masstree [EuroSys’12], MemC3 [NSDI’12], MICA [NSDI’14]

• Storage load balancing
  – DHT (Pastry [Middleware’01], CFS [SOSP’01], Chord [SIGCOMM’01]), Proteus [ICDCS’13]

• Access load balancing
  – SmallCache [SoCC’11], Chronos [SoCC’12], SPORE [SoCC’13], Streaming Analytics [Feedback’14]
Related work

• High performance in-memory KV store
  – Masstree [EuroSys’12], MemC3 [NSDI’12], MICA [NSDI’14]

• Storage load balancing
  – DHT (Pastry [Middleware’01], CFS [SOSP’01], Chord [SIGCOMM’01]), Proteus [ICDCS’13]

• Access load balancing
  – SmallCache [SoCC’11], Chronos [SoCC’12], SPORE [SoCC’13], Streaming Analytics [Feedback’14]
Conclusions

• Fine-grained, horizontal partitioning of in-memory data structure
  – Eliminates sync overhead
  – Enables load balancing

• MBal synthesizes three replication and migration techniques into a holistic system
  – Reduces load imbalance
  – Improves tail latency
Thank you!

http://research.cs.vt.edu/dssl/

Yue Cheng  Aayush Gupta  Ali R. Butt
Backup Slides
Memcached is desirable

• Quick deployment
• Ease of use
Memcached deployment in the Cloud

- Quick deployment
- Ease of use
- Elastic scale-up
- Elastic scale-out
Memcached deployment in the Cloud

- Quick deployment
- Ease of use
- Elastic scale-up
- Elastic scale-out

<table>
<thead>
<tr>
<th>Instance type</th>
<th>vCPU</th>
<th>ECU</th>
<th>N/w (Gbps)</th>
<th>Price/hr</th>
</tr>
</thead>
<tbody>
<tr>
<td>m1.small</td>
<td>1</td>
<td>1</td>
<td>0.1</td>
<td>$0.044</td>
</tr>
<tr>
<td>m3.medium</td>
<td>1</td>
<td>3</td>
<td>0.5</td>
<td>$0.070</td>
</tr>
<tr>
<td>c3.large</td>
<td>2</td>
<td>7</td>
<td>0.6</td>
<td>$0.105</td>
</tr>
<tr>
<td>m3.xlarge</td>
<td>4</td>
<td>13</td>
<td>0.7</td>
<td>$0.280</td>
</tr>
<tr>
<td>c3.2xlarge</td>
<td>8</td>
<td>28</td>
<td>1</td>
<td>$0.420</td>
</tr>
<tr>
<td>c3.8xlarge</td>
<td>32</td>
<td>108</td>
<td>10</td>
<td>$1.680</td>
</tr>
</tbody>
</table>
Memcached deployment in the Cloud

- Quick deployment
- Ease of use
- Elastic scale-up
- Elastic scale-out

<table>
<thead>
<tr>
<th>Instance type</th>
<th>vCPU</th>
<th>ECU</th>
<th>N/w (Gbps)</th>
<th>Price/hr</th>
</tr>
</thead>
<tbody>
<tr>
<td>m1.small</td>
<td>1</td>
<td>1</td>
<td>0.1</td>
<td>$0.044</td>
</tr>
<tr>
<td>m3.medium</td>
<td>1</td>
<td>3</td>
<td>0.5</td>
<td>$0.070</td>
</tr>
<tr>
<td>c3.large</td>
<td>2</td>
<td>7</td>
<td>0.6</td>
<td>$0.105</td>
</tr>
<tr>
<td>m3.xlarge</td>
<td>4</td>
<td>13</td>
<td>0.7</td>
<td>$0.280</td>
</tr>
<tr>
<td>c3.2xlarge</td>
<td>8</td>
<td>28</td>
<td>1</td>
<td>$0.420</td>
</tr>
<tr>
<td>c3.8xlarge</td>
<td>32</td>
<td>108</td>
<td>10</td>
<td>$1.680</td>
</tr>
</tbody>
</table>
“Decision paralysis ...”

<table>
<thead>
<tr>
<th>Instance type</th>
<th>vCPU</th>
<th>ECU</th>
<th>N/w (Gbps)</th>
<th>Price/hr</th>
</tr>
</thead>
<tbody>
<tr>
<td>m1.small</td>
<td>1</td>
<td>1</td>
<td>0.1</td>
<td>$0.044</td>
</tr>
<tr>
<td>m3.medium</td>
<td>1</td>
<td>3</td>
<td>0.5</td>
<td>$0.070</td>
</tr>
<tr>
<td>c3.large</td>
<td>2</td>
<td>7</td>
<td>0.6</td>
<td>$0.105</td>
</tr>
<tr>
<td>m3.xlarge</td>
<td>4</td>
<td>13</td>
<td>0.7</td>
<td>$0.280</td>
</tr>
<tr>
<td>c3.2xlarge</td>
<td>8</td>
<td>28</td>
<td>1</td>
<td>$0.420</td>
</tr>
<tr>
<td>c3.8xlarge</td>
<td>32</td>
<td>108</td>
<td>10</td>
<td>$1.680</td>
</tr>
</tbody>
</table>

Getting the most “BANG for the buck”
“Decision paralysis ...”

<table>
<thead>
<tr>
<th>Instance type</th>
<th>vCPU</th>
<th>ECU</th>
<th>N/w (Gbps)</th>
<th>Price/hr</th>
</tr>
</thead>
<tbody>
<tr>
<td>m1.small</td>
<td>1</td>
<td>1</td>
<td>0.1</td>
<td>$0.044</td>
</tr>
<tr>
<td>m3.medium</td>
<td>1</td>
<td>3</td>
<td>0.5</td>
<td>$0.070</td>
</tr>
<tr>
<td>c3.large</td>
<td>2</td>
<td>7</td>
<td>0.6</td>
<td>$0.105</td>
</tr>
<tr>
<td>m3.xlarge</td>
<td>4</td>
<td>13</td>
<td>0.7</td>
<td>$0.280</td>
</tr>
<tr>
<td>c3.2xlarge</td>
<td>8</td>
<td>28</td>
<td>1</td>
<td>$0.420</td>
</tr>
<tr>
<td>c3.8xlarge</td>
<td>32</td>
<td>108</td>
<td>10</td>
<td>$1.680</td>
</tr>
</tbody>
</table>

Getting the most “BANG for the buck”

- Desire 1: performance
- Desire 2: $ efficiency
### Desire 1: Performance

<table>
<thead>
<tr>
<th>Instance type</th>
<th>vCPU</th>
<th>ECU</th>
<th>N/w (Gbps)</th>
<th>Price/hr</th>
</tr>
</thead>
<tbody>
<tr>
<td>m1.small</td>
<td>1</td>
<td>1</td>
<td>0.1</td>
<td>$0.044</td>
</tr>
<tr>
<td>m3.medium</td>
<td>1</td>
<td>3</td>
<td>0.5</td>
<td>$0.070</td>
</tr>
<tr>
<td>c3.large</td>
<td>2</td>
<td>7</td>
<td>0.6</td>
<td>$0.105</td>
</tr>
<tr>
<td>m3.xlarge</td>
<td>4</td>
<td>13</td>
<td>0.7</td>
<td>$0.280</td>
</tr>
<tr>
<td>c3.2xlarge</td>
<td>8</td>
<td>28</td>
<td>1</td>
<td>$0.420</td>
</tr>
<tr>
<td>c3.8xlarge</td>
<td>32</td>
<td>108</td>
<td>10</td>
<td>$1.680</td>
</tr>
</tbody>
</table>

#### Throughput (QPS in millions)

![Graph showing throughput vs Memcached cluster size]

- **Desired cluster size**: 10
- **Throughput**: 2.4M QPS
- **Data type**: 95% GET, 5% SET, Uniform

**Note**: The graph shows the throughput in millions of QPS for different Memcached cluster sizes. The x-axis represents the Memcached cluster size, and the y-axis represents the throughput in millions. The data points correspond to different instance types, with the c3.8xlarge instance achieving the highest throughput of 2.4M QPS at a cluster size of 10.
## Desire 1: Performance

<table>
<thead>
<tr>
<th>Instance type</th>
<th>vCPU</th>
<th>ECU</th>
<th>N/w (Gbps)</th>
<th>Price/hr</th>
</tr>
</thead>
<tbody>
<tr>
<td>m1.small</td>
<td>1</td>
<td>1</td>
<td>0.1</td>
<td>$0.044</td>
</tr>
<tr>
<td>m3.medium</td>
<td>1</td>
<td>3</td>
<td>0.5</td>
<td>$0.070</td>
</tr>
<tr>
<td>c3.large</td>
<td>2</td>
<td>7</td>
<td>0.6</td>
<td>$0.105</td>
</tr>
<tr>
<td>m3.xlarge</td>
<td>4</td>
<td>13</td>
<td>0.7</td>
<td>$0.280</td>
</tr>
<tr>
<td>c3.2xlarge</td>
<td>8</td>
<td>28</td>
<td>1</td>
<td>$0.420</td>
</tr>
<tr>
<td>c3.8xlarge</td>
<td>32</td>
<td>108</td>
<td>10</td>
<td>$1.680</td>
</tr>
</tbody>
</table>

### Network is the bottleneck!

- Throughput (QPS in millions) vs Memcached cluster size
- 95% GET, 5% SET, Uniform

![Graph showing throughput vs cluster size]
# Desire 1: Performance

## Instance type

<table>
<thead>
<tr>
<th>Instance type</th>
<th>vCPU</th>
<th>ECU</th>
<th>N/w (Gbps)</th>
<th>Price/hr</th>
</tr>
</thead>
<tbody>
<tr>
<td>m1.small</td>
<td>1</td>
<td>1</td>
<td>0.1</td>
<td>$0.044</td>
</tr>
<tr>
<td>m3.medium</td>
<td>1</td>
<td>3</td>
<td>0.5</td>
<td>$0.070</td>
</tr>
<tr>
<td>c3.large</td>
<td>2</td>
<td>7</td>
<td>0.6</td>
<td>$0.105</td>
</tr>
<tr>
<td>m3.xlarge</td>
<td>4</td>
<td>13</td>
<td>0.7</td>
<td>$0.280</td>
</tr>
<tr>
<td>c3.2xlarge</td>
<td>8</td>
<td>28</td>
<td>1</td>
<td>$0.420</td>
</tr>
<tr>
<td>c3.8xlarge</td>
<td>32</td>
<td>108</td>
<td>10</td>
<td>$1.680</td>
</tr>
</tbody>
</table>

**CPU is the bottleneck!**

---

95% GET, 5% SET, Uniform
Desire 2: $ efficiency

$ efficiency = QPS/$

<table>
<thead>
<tr>
<th>Instance type</th>
<th>vCPU</th>
<th>ECU</th>
<th>N/w (Gbps)</th>
<th>Price/hr</th>
</tr>
</thead>
<tbody>
<tr>
<td>m1.small</td>
<td>1</td>
<td>1</td>
<td>0.1</td>
<td>$0.044</td>
</tr>
<tr>
<td>m3.medium</td>
<td>1</td>
<td>3</td>
<td>0.5</td>
<td>$0.070</td>
</tr>
<tr>
<td><strong>c3.large</strong></td>
<td>2</td>
<td>7</td>
<td>0.6</td>
<td><strong>$0.105</strong></td>
</tr>
<tr>
<td>m3.xlarge</td>
<td>4</td>
<td>13</td>
<td>0.7</td>
<td>$0.280</td>
</tr>
<tr>
<td>c3.2xlarge</td>
<td>8</td>
<td>28</td>
<td>1</td>
<td>$0.420</td>
</tr>
<tr>
<td>c3.8xlarge</td>
<td>32</td>
<td>108</td>
<td>10</td>
<td>$1.680</td>
</tr>
</tbody>
</table>

95% GET, 5% SET, Uniform

\[ \text{Million QPS/} \]
Desire 2: $ efficiency

- Adding more resources is NOT a good solution
- Extra CPU capacity is wasted in the cloud
- Instance with modest CPU offers best $ efficiency

$ efficiency = QPS/$
**MBal evaluation – complete system**

- 8-core 2.5GHz, 2×10MB L3 LLC, 64GB DRAM
- Zipfian workload, 75% GET, 10B key 20B value
- 10Gb Ethernet, MultiGET

<table>
<thead>
<tr>
<th>Number of threads/instances</th>
<th>MBal</th>
<th>Mercury</th>
<th>Memcached</th>
<th>Multi-inst Mc</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.2</td>
<td>0.1</td>
<td>0.2</td>
<td>0.2</td>
</tr>
<tr>
<td>2</td>
<td>0.5</td>
<td>0.4</td>
<td>0.2</td>
<td>0.4</td>
</tr>
<tr>
<td>4</td>
<td>0.9</td>
<td>0.6</td>
<td>0.3</td>
<td>0.8</td>
</tr>
<tr>
<td>6</td>
<td>1.3</td>
<td>0.7</td>
<td>0.2</td>
<td>1.1</td>
</tr>
<tr>
<td>8</td>
<td>1.8</td>
<td>0.8</td>
<td>0.3</td>
<td>1.5</td>
</tr>
</tbody>
</table>

Throughput (QPS in millions)
MBal evaluation – complete system

- 8-core 2.5GHz, 2×10MB L3 LLC, 64GB DRAM
- Zipfian workload, 75% GET, 10B key 20B value
- 10Gb Ethernet, MultiGET
MBal evaluation – complete system

✔ MBal uses lightweight CPU cache-aligned bucket locks!

Throughput (QPS in millions)

<table>
<thead>
<tr>
<th>Number of threads/instances</th>
<th>MBal</th>
<th>Mercury</th>
<th>Memcached</th>
<th>Multi-inst Mc</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.2</td>
<td>0.2</td>
<td>0.1</td>
<td>0.2</td>
</tr>
<tr>
<td>2</td>
<td>0.5</td>
<td>0.4</td>
<td>0.2</td>
<td>0.4</td>
</tr>
<tr>
<td>4</td>
<td>0.9</td>
<td>0.6</td>
<td>0.3</td>
<td>0.8</td>
</tr>
<tr>
<td>6</td>
<td>1.3</td>
<td>0.7</td>
<td>0.2</td>
<td>1.1</td>
</tr>
<tr>
<td>8</td>
<td>1.8</td>
<td>0.8</td>
<td>0.3</td>
<td>1.5</td>
</tr>
</tbody>
</table>

20%
Event breakdown in E2E test

Phase 3 is sparingly used
Multi-core scalability

- 32-core 2GHz, 64GB DRAM
- memaslap with MultiGET, 16B key 32B value
- 10GbE network

![Graph showing per-core throughput vs. number of threads for MBal, Mercury, and Memcached with 90% and 50% GET requests. Ideal scalability is indicated by a dashed line.](image-url)
99\textsuperscript{th} percentile latency vs. throughput

Throughput improvement

Latency improvement

Memcached
Mercury
MBal (w/o load balancer)
MBal (P1)
MBal (P2)
MBal (P3)
MBal (Unif)