MICA: A Holistic Approach to Fast In-Memory Key-Value Storage

Hyeontaek Lim

Dongsu Han, David G. Andersen, Michael Kaminsky

1Carnegie Mellon University
2KAIST, 3Intel Labs
Goal: Fast In-Memory Key-Value Store

- Improve per-node performance (op/sec/node)
  - Less expensive
  - Easier hotspot mitigation
  - Lower latency for multi-key queries

- Target: small key-value items (fit in single packet)

- Non-goals: cluster architecture, durability
Q: How Good (or Bad) are Current Systems?

- Workload: YCSB [SoCC 2010]
  - Single-key operations

- In-memory storage
  - Logging turned off in our experiments

- End-to-end performance over the network

- Single server node
End-to-End Performance Comparison

Throughput (M operations/sec)

- Published results; Logging on RAMCloud/Masstree
- Using Intel DPDK (kernel bypass I/O); No logging
- (Write-intensive workload)
End-to-End Performance Comparison

Throughput (M operations/sec)

- **95% GET ORIG** - Published results; Logging on RAMCloud/Mastree
- **95% GET OPT** - Using Intel DPDK (kernel bypass I/O); No logging
- **50% GET OPT** - (Write-intensive workload)

Performance collapses under heavy writes
End-to-End Performance Comparison

Throughput (M operations/sec)

- 95% GET ORIG
- 95% GET OPT
- 50% GET OPT

Maximum packets/sec attainable using UDP

<table>
<thead>
<tr>
<th>System</th>
<th>Memcached</th>
<th>RAMCloud</th>
<th>MemC3</th>
<th>Masstree</th>
<th>MICA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Throughput (M operations/sec)</td>
<td>1.41</td>
<td>0.1</td>
<td>4.45</td>
<td>8.9</td>
<td>65.6</td>
</tr>
<tr>
<td>Throughput (M operations/sec)</td>
<td>30.7</td>
<td>1</td>
<td>5.7</td>
<td>16.5</td>
<td>70.4</td>
</tr>
</tbody>
</table>
MICA Approach

- **MICA**: Redesigning in-memory key-value storage
  - Applies new SW architecture and data structures to general-purpose HW in a holistic way

Diagram:

1. Parallel data access
2. Request direction
3. Key-value data structures (cache & store)
Parallel Data Access

Modern CPUs have many cores (8, 15, ...)  
How to exploit CPU parallelism efficiently?

- Modern CPUs have many cores (8, 15, ...)
- How to exploit CPU parallelism efficiently?
Parallel Data Access Schemes

Concurrent Read
Concurrent Write

+ Good load distribution
- Limited CPU scalability (e.g., synchronization)
- Cross-NUMA latency

Exclusive Read
Exclusive Write

+ Good CPU scalability
- Potentially low performance under skewed workloads
In MICA, Exclusive Outperforms Concurrent

Throughput (Mops)

End-to-end performance with kernel bypass I/O
Request Direction

- Sending requests to appropriate CPU cores for better data access locality
- Exclusive access benefits from correct delivery
  - Each request must be sent to corresp. partition’s core

2. Request direction

1. Parallel data access

3. Key-value data structures
Request Direction Schemes

Flow-based Affinity

Server node

Client → CPU
NIC
Client

Classification using 5-tuple

+ Good locality for flows (e.g., HTTP over TCP)

- Suboptimal for small key-value processing

Object-based Affinity

Server node

Client → CPU
NIC
Client

Classification depends on request content

+ Good locality for key access

- Client assist or special HW support needed for efficiency
Crucial to Use NIC HW for Request Direction

Throughput (Mops)

- **Uniform**
- **Skewed**

Request direction done solely by software:
- Uniform: 33.9
- Skewed: 28.1

Client-assisted hardware-based request direction:
- Uniform: 76.9
- Skewed: 70.4

Using exclusive access for parallel data access
• Significant impact on key-value processing speed
• New design required for very high op/sec for both read and write
• “Cache” and “store” modes
MICA’s “Cache” Data Structures

• Each partition has:
  • **Circular log** (for memory allocation)
  • **Lossy concurrent hash index** (for fast item access)

• Exploit Memcached-like cache semantics
  • Lost data is easily recoverable (not free, though)
  • Favor fast processing

• Provide good memory efficiency & item eviction
Circular Log

- Allocates space for key-value items of any length
- Conventional logs + Circular queues
  - Simple garbage collection/free space defragmentation

New item is appended at tail

Head — Tail

Insufficient space for new item?

Evict oldest item at head (FIFO)

Tail — Head

(fixed log size)

Support LRU by reinserting recently accessed items
Lossy Concurrent Hash Index

- Indexes key-value items stored in the circular log
- Set-associative table

- Full bucket? Evict oldest entry from it
  - Fast indexing of new key-value items
MICA’s “Store” Data Structures

• Required to preserve stored items

• Achieve similar performance by trading memory
  • Circular log -> Segregated fits
  • Lossy index -> Lossless index (with bulk chaining)

• See our paper for details
Evaluation

• Going back to end-to-end evaluation...

• Throughput & latency characteristics
Throughput Comparison

Throughput (Mops)

- **Uniform, 50% GET**
- **Uniform, 95% GET**
- **Skewed, 50% GET**
- **Skewed, 95% GET**

End-to-end performance with kernel bypass I/O

**Similar performance regardless of skew/write**

**Large performance gap**

**Bad at high write ratios**
Throughput-Latency on Ethernet

Average latency (μs)

Throughput (Mops)

Original Memcached using standard socket I/O; both use UDP

200x+ throughput
MICA

- Redesigning in-memory key-value storage
- 65.6+ Mops/node even for heavy skew/write
- Source code: github.com/efficient/mica

![Diagram of MICA architecture](image)

1. Parallel data access
2. Request direction
3. Key-value data structures (cache & store)