HICAMP:
Architectural Support for Efficient Concurrency-Safe Shared Structured Data Access

Cheriton et al., ASPLOS 2012

Yoongu Kim
11/18/2013
INTRODUCTION
Intro: Shared Data

Thread 1

Thread 2

DRAM

4GB

0GB

Private

Shared Data

Private
Intro: Shared Data

Thread 1

Thread 2

DRAM

Private

a[999]

a[1]

a[0]

4GB

0GB

shared array
Problem: Concurrent Accesses

for (i = 0; i < 1000; i++)
    sum = sum + a[i];

Read access to shared array

CONFLICT!

Write access to shared array

a[900] = -1;
Traditional Solutions are Expensive

Solution #1: Lock

– Only *one* thread can access shared data ...
– ... the thread that holds the lock

• But what if shared data is *very large*?
– Example: Bank database
– When an auditing thread accesses the bank database, all other threads would starve
  • No deposits/withdrawals for any customer
Traditional Solutions are Expensive

Solution #2: Transaction

- *Speculatively* allow multiple threads to access shared data in a concurrent manner
- If lucky $\Rightarrow$ no conflict
- If unlucky $\Rightarrow$ undo changes to shared data & retry

• But what if a transaction is *very long*?
  - 100% chance of being unlucky
  - Undoing/retrying a transaction is wasteful
**Throughput vs. Number of Cores**

- Ideal
- Actual

**Gap**

*Sharing is the root of all evil*
Figure 3: MOSBENCH results summary. Each bar shows the ratio of per-core throughput with 48 cores to throughput on one core, with 1.0 indicating perfect scalability. Each pair of bars corresponds to one application before and after our kernel and application modifications.
Alternative Solution: “Snapshotting”

Thread 1

Thread 2

Shared Data

READ
Alternative Solution: “Snapshottting”
Alternative Solution: “Snapshotting”
Alternative Solution: “Snapshotting”
Key Question

*How to make memory “snapshots” cheap?*

- Naïve approaches are very expensive
  1. Performance waste: copying data
  2. Capacity waste: duplicate data

- A better approach: **HICAMP**
  - Provides hardware-support for “snapshots” while incurring only small overheads
HICAMP: THE BASICS
What is HICAMP?

1. Hierarchical
2. Immutable
3. Content-Addressable Memory
4. Processor
1. ‘H’ of HICAMP: “Hierarchical”
1. ‘H’ of HICAMP: “Hierarchical”
1. ‘H’ of HICAMP: “Hierarchical”
1. ‘H’ of HICAMP: “Hierarchical”

Non-Hierarchical

<table>
<thead>
<tr>
<th>Data₁</th>
<th>Data₂</th>
<th>Data₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addr₁</td>
<td>Addr₂</td>
<td>Addr₃</td>
</tr>
</tbody>
</table>

| 0GB   | 4GB   | 0GB   |

Hierarchical

<table>
<thead>
<tr>
<th>Data₁</th>
<th>Data₂</th>
<th>Data₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₁</td>
<td>A₂</td>
<td></td>
</tr>
</tbody>
</table>

| 0GB   | 4GB   | 0GB   |

Addr₄
1. ‘H’ of HICAMP: “Hierarchical”
1. ‘H’ of HICAMP: “Hierarchical”
1. ‘H’ of HICAMP: “Hierarchical”

Non-Hierarchical

Data₃
Data₂
Data₁

Addr₃
Addr₂
Addr₁

0GB

4GB

Root Addr:

Addr₅

A₄
A₃

A₁
A₂

Data₃

Data₁

Data₂
What is HICAMP?

1. Hierarchical
2. Immutable
3. Content-Addressable Memory
4. Processor
2. ‘I’ of HICAMP: “Immutable”

Overwriting of data is not allowed
You must create a new hierarchy
2. ‘I’ of HICAMP: “Immutable”

*Old and new hierarchies coexist*

Old and new hierarchies coexist.

DEDUPLICATION
What is HICAMP?

1. Hierarchical
2. Immutable
3. Content-Addressable Memory
4. Processor
3. “CAM” of HICAMP

Traditional

4GB

0x123
0x123
0x123
0x123

How to eliminate duplicate values?

0GB
3. “CAM” of HICAMP

• **Q:** Why do duplicates exist?
• **A:** Because you can store the same value anywhere you want.

For a particular value, let’s restrict the addresses it can have.
3. “CAM” of HICAMP

Set of 64-byte values

\[ 2^{(64 \times 8)} \text{ elements} \]

\[ f(x) \]

Hash function

Set of addresses in 4GB DRAM

\[ \approx 2^{(32-6)} \]
3. “CAM” of HICAMP

```
<table>
<thead>
<tr>
<th>Row</th>
<th>Col1</th>
<th>Col2</th>
<th>ColM</th>
</tr>
</thead>
<tbody>
<tr>
<td>RowN</td>
<td>64B</td>
<td>64B</td>
<td>64B</td>
</tr>
<tr>
<td>Row3</td>
<td>64B</td>
<td>64B</td>
<td>64B</td>
</tr>
<tr>
<td>Row2</td>
<td>64B</td>
<td>64B</td>
<td>64B</td>
</tr>
<tr>
<td>Row1</td>
<td>64B</td>
<td>64B</td>
<td>64B</td>
</tr>
</tbody>
</table>
```

4GB 0GB
3. “CAM” of HICAMP

64-byte data value

Row Address

Row 77

Column Address: 1–M
3. “CAM” of HICAMP

\[ f(x) \]

Data Value

Data Address

*fixed*  *flexible*: to reduce hash conflicts
PROGRAMMING MODEL
Terminology

“Root PLID”

“Segment”

“Physical Line”

“Physical Line ID” (PLID)
Virtual-to-Physical Translation

"Virtual Segment ID"

Segment Map

Virtual Segment ID

Segment Map

Data

Hardware

Software
Example Program

1: it = obj.begin(); /* it = iterator */
2: it++;
3: it++;
4: *it = newVal;
5: it->tryCommit();
Example Program

1: it = obj.begin(); /* it = iterator */
2: it++;
3: it++;
4: *it = newVal;
5: it->tryCommit();
Example Program

1: it = obj.begin(); /* it = iterator */
2: it++;
3: it++;
4: *it = newVal;
5: it->tryCommit();
Example Program

1: it = obj.begin();
2: it++;
3: it++;
4: *it = newVal;
5: it->tryCommit();

/* it = iterator */

obj
(VSID)

begin()

it
Example Program

1: it = obj.begin();
2: it++;
3: it++;
4: *it = newVal;
5: it->tryCommit();

/* it = iterator */

obj (VSID)

it

begin()
Example Program

1: it = obj.begin(); /* it = iterator */
2: it++;
3: it++;
4: it = newVal;
5: it->tryCommit();

Diagram:

- obj (VSID)
- A4 A3 copy A4 A10
- A1 A2
- Data1
- Data3 copy newVal
- Data2

Code:

```
Example Program
1: it = obj.begin(); /* it = iterator */
2: it++;
3: it++;
4: it = newVal;
5: it->tryCommit();
```