Fast Memory Snapshot for Concurrent Programming without Synchronization

Jaewoong Chung  Woongki Baek, Christos Kozyrakis
Advanced Development and Research Lab (RADL)  Computer System Lab (CSL)
AMD  Stanford
Fast Memory Snapshot

• Era of multi-core processors
  ▪ Not easy to develop concurrent software

• Memory Snapshot
  ▪ Not a new idea: atomic multi-word read
  ▪ But, not used for performance-critical applications

• Hardware-assisted Memory Snapshot
  ▪ Leverage hardware transactional memory resources
Example: Memory Profiling (1)

- Global Lock
  - Performance degradation

Application Thread

Profiling Thread
Example: Memory Profiling (2)

- Fine-grain Lock
  - Potential deadlock
Example: Memory Profiling (3)

- Transaction
  - Frequent rollback
Example: Memory Profiling (4)

- Tri-coloring
  - White, grey, and black
  - Specific to graph traversal
  - Not general enough for atomic multiword read

- Virtual memory protection
  - Page fault, false sharing
Example: Memory Profiling (5)

- Snapshot

Application Thread

Profiling Thread
More Formally ...

• Snapshot
  ▪ Given M memory elements and P processors,
  ▪ Scan: build a snapshot of M elements
  ▪ Update: update each element

• Software Implementations
  ▪ Leverages single-word atomic instructions
  ▪ Typically, $O(M)$ scan time and $O(MP)$ update time
  ▪ Slow for performance-critical applications
Our Proposal

• **MShot**
  - Hardware-assisted snapshot
  - \( O(P) \) scan time, \( O(1) \) update time

• **Use Cases**
  - Fast Checkpoint: Fault tolerance, Guest OS migration
  - Concurrent Garbage Collection and Profiling
  - In-memory Database
  - More use cases in the paper
MShot Images

• Master (up-to-date) Image

• Snapshot Image
  - Multiple disjoint snapshot regions + Registers
  - Read-only
MShot API

- **Take_snapshot (snapshot regions)**
  - Creates a snapshot on the regions and returns snapshot ID (SID)

- **Join_snapshot (SID)**
  - Become *Snapshot User*
  - Load instructions start to read from snapshot images

- **Leave_snapshot (SID)**
  - Load instructions start to read from master images

- **Destroy_snapshot (SID)**
MShot components

• Software
  ▪ Snapshot Information Table (SIT)
    – Entry per snapshot
    – SID, snapshot regions, join_list

• Hardware
  ▪ Snapshot Information Look-aside Buffer (SLB)
    – To speed up accesses to SIT
  ▪ MShot metadata bits per cache line
    – To distinguish snapshot data from master data
MShot Hardware Components

- Computing Core
  - TLB
  - Snapshot Information Look-aside Buffer (SLB)
    - Start_addr
    - End_addr
    - SID
    - J
  - Physical Address
  - SID, J
- Data Cache
  - Tags
  - Data
  - MS
  - RS
  - SID
MShot Metadata Bits

• **MS (Modified after Snapshot)**
  - Set for master data

• **RS (Read after Snapshot)**
  - Set for snapshot data

• **Neither bits are set if master data == snapshot data**
Create Snapshot = Scan

• Three-way handshaking
  - Through Inter-Processor Interrupt
    - $O(P)$ scan time

```
start
  take_snapshot()  Snapshot-handler
  Wait for All handlers
  SIG_MShot Ready Resume
  Save registers
  Load SLB entries
  Handler returns
end
```
Write to Snapshot = update

• Hit if addr_tag matches and RS is not set.
  - Globally visible by invalidating other copies
    - \(O(1)\) update time

• Snapshot user can’t write.
Read from Snapshot

- Snapshot user
  - Hit if addr_tag matches and MS is not set
  - RS is set if MS is piggy-backed in refill message

Data Cache 1

Data | MS | RS | SID
---|----|----|----
Snapshot |     |    |    
Same |     |    |    
Snapshot |     |    |    

Refill Data From Memory

Data Cache 2

Data | MS | RS | SID
---|----|----|----
Master |     |    |    

MS snooped
Destroy snapshot

• Invalidate cache lines with RS

• Gang-clear MS and RS

• Remove entry from SLB and SIT
Integration with TM

• Similarity between MShot and TM
  ▪ Snapshot = read-only transaction that never aborts.
  ▪ Snapshot image = last committed data
  ▪ Master image = transactional data
  ▪ Snapshot virtualization = transaction virtualization

• Our approach
  ▪ Given TM resources, reuse them to low MShot implementation cost.
Baseline TM = Cache-based HTM

• Additional bits per cache line
  - TW : Transactional Write
  - TR : Transactional Read
  - TID : Transaction ID

• Page-based TM (PTM) for virtualization (ASPLOS ‘06)
  - Shadow page table
    - Secondary TLB to manage pages with overflowed transactional data
  - Home (original) page for transactional data
  - Shadow (new) page for last committed data
## TM Resource Sharing

<table>
<thead>
<tr>
<th>HTM resource</th>
<th>MShot resource</th>
</tr>
</thead>
<tbody>
<tr>
<td>TM Metadata bits (TW,TR)</td>
<td>MShot Metadata bits (MS, RS)</td>
</tr>
<tr>
<td>TID</td>
<td>SID</td>
</tr>
<tr>
<td>TW/TR Gang-clear logic</td>
<td>MS/RS Gang-clear logic</td>
</tr>
<tr>
<td>Home/Shadow pages</td>
<td>Master/snapshot images</td>
</tr>
<tr>
<td>Shadow page table</td>
<td>Providing access to overflowed master/snapshot images</td>
</tr>
</tbody>
</table>

- More details in the paper
Hardware (Storage) Cost Calculation

- Assuming a baseline system
  - 16 cores, 64K L1, 1M private L2, 32B cache line, 16GB memory

<table>
<thead>
<tr>
<th>Baseline System</th>
<th>Storage Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>No TM Resources</td>
<td>2.98 %</td>
</tr>
<tr>
<td>TM with virtualization support</td>
<td>0.07 %</td>
</tr>
<tr>
<td>TM without virtualization support</td>
<td>0.43 %</td>
</tr>
</tbody>
</table>
Simulation Environment

- Cycle-accurate multi-core simulator

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>2GHz, single-issue, in-order x86 core</td>
</tr>
<tr>
<td>SLB</td>
<td>64 entries</td>
</tr>
<tr>
<td>Cache</td>
<td>64K L1, 1M private L2, 32B line, MESI</td>
</tr>
<tr>
<td>Shadow Page Table</td>
<td>2048 entries</td>
</tr>
<tr>
<td>Memory</td>
<td>4GH, 100 cycle latency</td>
</tr>
<tr>
<td>Interconnect</td>
<td>Tiled, 32B link, 3 cycles per hope</td>
</tr>
</tbody>
</table>
Snapshot GC

- Take a snapshot of a heap generation before GC
  - Collectors on snapshot image
  - Mutators (application threads) on master image
Snapshot Call-path Profiling

- Take a snapshot of thread stack before profiling
  - Profiler on snapshot image
  - Application threads on master image
Takeaway points

• Fast Memory Snapshot
  - Useful concurrent programming primitive
  - Atomic multi-word read operation
    - Snapshot GC, Snapshot Call-path profilers

• Cost-effective implementation with TM resources
  - Down to 0.07 % storage overhead

• AMD Advanced Synchronization Facility (ASF)
  - AMD 64 extension for Transactional Memory
Questions?
Jaewooong.Chung@amd.com