TSO-Atomicity: TSO Enforcement for Aggressive Program Optimization

Cheng Wang, Youfeng Wu, Jaewoong Chung
Programming Systems Lab / MPR
Intel Labs
Motivation

• Two different usage models in architecture support for atomic region execution
  – HW primitive to enforce exclusive memory access
    – Support transactional memory, lock elision, etc
  – HW primitive to enforce Sequential Consistency (SC)
    – Support SC-preserve aggressive program optimization

• Popular architectures like X86 / SPARC implement Total Store Ordering (TSO) instead of SC
  – Weaker consistency in TSO leads to more efficient HW implementation

• TSO-Atomicity: new HW primitive to enforce TSO
  – Weaker consistency than atomicity leads to more efficient HW implementation than atomicity
  – Support TSO-preserve aggressive program optimization
Relationship between Different Consistency Model

Enforce Memory Consistency for Optimization

Atomicity

TSO-Atomicity

Weaker Consistency with More Efficient Implementation
**SC Vs. TSO**

**SC Execution Trace**
- ld1
- st2_start
- st2_end
- st3_start
- st3 cache miss
- st3_end
- ld4
- st5_start
- st5_end
- ld6
- ld7

**TSO Execution Trace**
- ld1
- st2_start
- st2_end
- st3_start
- st3 cache miss
- st3_start
- ld4
- st5_start
- ld6
- st3_end
- st5_end
- ld7

- **st_start** write store data into store buffer
- **st_end** write store data from store buffer to cache/memory
- Later load may be reordered with earlier store
- Data forwarded from store buffer to later load
Relationship between Different Consistency Model

Enforce Memory Consistency for Optimization

Atomicity

SC

TSO

TSO-Atomicity

Weaker Consistency with More Efficient Implementation
Execution with Atomicity

- first inst execution
- first load execution
- first store execution
- commit all load/store atomically
- snoop load
- snoop store
Inefficiency of Atomicity

region1

ld1
st2_start
st2_end
st3_start
st3 cache miss
st3_end

region2

ld4
st5_start
st5_end
ld6
ld7

Across region boundary, atomicity suffer similar inefficiency as SC
Atomicity Vs. TSO-Atomicity

Execution with Atomicity

Execution with TSO-Atomicity
Efficient Implementation of TSO-Atomicity

**TSO-Atomicity Execution Trace**

- ld1
- st2_start
- st2_end
- st3_start
- ld4
- st5_start
- ld6
- st3_end
- st5_end
- ld7

- Load snoop in region2 use the speculative bits release by load commit in region1
- No snooping for store data in store-buffer until written to cache
- Across region boundary, TSO-atomicity achieve same efficiency as TSO
Prevent Unnecessary Abort

Init: A=0, B=0

thread1

\[ r_1 \leftarrow \text{ld} \ B \]
\[ \text{st} \ A \leftarrow 1 \]

thread2

\[ r_2 \leftarrow \text{ld} \ A \]
\[ \text{st} \ B \leftarrow 1 \]

Results: \( r_1 = 0, r_2 = 0 \)

- By committing load earlier, reduce unnecessary abort due to memory conflicts
- To enforce TSO instead of exclusive memory access, TSO-atomicity is not constrained to produce only serializable scheduling
Semantics of Memory Consistency Models

**Init:** $A=0$, $B=0$

- **Thread 1:**
  - `st A ← 1`
  - `r1 ← ld B`

- **Thread 2:**
  - `st B ← 1`
  - `r2 ← ld A`

<table>
<thead>
<tr>
<th>Consistency Model</th>
<th>Results ($r1$, $r2$)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>TSO</code></td>
<td>(0, 0), (0, 1), (1, 0), (1, 1)</td>
</tr>
<tr>
<td><code>SC</code></td>
<td>(0, 1), (1, 0), (1, 1)</td>
</tr>
<tr>
<td><code>Atomicity</code></td>
<td>(0, 1), (1, 0)</td>
</tr>
<tr>
<td><code>TSO-Atomicity</code></td>
<td>(0, 0), (0, 1), (1, 0)</td>
</tr>
</tbody>
</table>
Relationship between Different Consistency Model

Enforce Memory Consistency for Optimization

Atomicity

SC

TSO

TSO-Atomicity

Weaker Consistency with More Efficient Implementation
Atomicity Enforce SC for Optimization

- Logically, all computation execute atomically at region commit time
- The semantic of region with atomicity is independent of optimizations within the region
  - Enforce SC for aggressive program optimization
Relationship between Different Consistency Model

Enforce Memory Consistency for Optimization

Atomicity  SC  TSO

TSO-Atomicity

Weaker Consistency with More Efficient Implementation
**TSO-Atomicity Enforce TSO for Optimization**

- Logically, all upward-exposed loads execute atomically at load commit
  - Non-upward-exposed loads get forwarded data
- Logically, all downward-exposed stores execute atomically at store commit
  - Non-downward-exposed stores are overwritten
- **The semantic of region with TSO-atomicity is also independent of optimizations within the region**
  - Enforce TSO for aggressive program optimization

---

**Programming Systems Lab**

15
Conclusions and Future Works

• TSO-atomicity is a new HW primitive to enforce TSO for aggressive program optimization
  – Weaker than atomicity for efficient HW implementation
  – Prevent unnecessary abort due to conflicts

• Experimental results shows that TSO-atomicity is efficient to support dynamic binary optimizations
  – 20% performance improvement through dynamic binary optimization with TSO-atomicity support

• It is interesting future works to study new HW primitives to enforce other relaxed memory consistency models for program optimizations
  – weaker atomicity than TSO-atomicity to enforce weaker memory consistency than TSO