Store Atomicity

What does atomicity really require?

Jan-Willem Maessen (Sun Microsystems)
Based on joint work with Arvind from ISCA’06

From Dataflow to Synthesis
May 18, 2007
What is atomic memory?

- **Operational view**: instruction at a time
- **Declarative view**: *serializability*

---

Memory, cache, buffers

Monolithic memory

Out of order processors
The Atomicity Puzzle
Puzzle 1: Serializability

Many serializations exist for a given execution:

- $S \times 1 \rightarrow L \ x=1 \rightarrow S \ y \ 2 \rightarrow L \ y=2$
- $S \ y \ 2 \rightarrow S \ x \ 1 \rightarrow L \ x=1 \rightarrow L \ y=2$
- $S \ y \ 2 \rightarrow S \ x \ 1 \rightarrow L \ y=2 \rightarrow L \ x=1$
- $S \ y \ 2 \rightarrow L \ y=2 \rightarrow S \ x \ 1 \rightarrow L \ x=1$
- $S \ x \ 1 \rightarrow S \ y \ 2 \rightarrow L \ y=2 \rightarrow L \ x=1$
Puzzle 1: Serializability

Only two serializations are possible.
Potential violations of Serializability: Example 1

Thread 1
S x,1
Fence
S y,2
L y = 3

Thread 2
S y,3
Fence
S x,4
L x = 1?

Predecessor Stores of a Load are ordered before its source
Potential violations of Serializability: Example 2

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>S x,1</td>
<td>S y,3</td>
</tr>
<tr>
<td>S x,2</td>
<td>S y,5</td>
</tr>
<tr>
<td>Fence</td>
<td>Fence</td>
</tr>
<tr>
<td>L y = 3</td>
<td>L x = 1?</td>
</tr>
</tbody>
</table>

Successor Stores of a Store are ordered after its observer
For Serializability we must have ...

Predecessor Stores of a Load are ordered before its source

Successor Stores of a Store are ordered after its observer

Surprisingly not enough to ensure serializability!

Recognized by Hangal, Vahia, Manovit, et al.
[TSOtool, ISCA ’04]
Must pay attention to pairs of unrelated observations ...

Mutual ancestors of unordered Loads are ordered before mutual successors of the Stores they observe.

Overconstraining rules out legal executions.
Potential violations of Serializability: Example 3

**Thread 1**
- S x,1
- Fence
- Ly = 2
- Ly = 4

**Thread 2**
- S y,2
- Fence
- S z,6

**Thread 3**
- S y,4
- Fence
- Lz = 6
- S x,8
- Lx = 1?

---

**Diagram**

- S x 1
- L y
- S y 2
- L y
- S z 6
- L z
- S x 8
- L x
Store Atomicity

- Predecessor Stores of a Load are ordered before its source
- Successor Stores of a Store are ordered after its observer
- Mutual ancestors of unordered Loads are ordered before mutual successors of the Stores they observe

Claim: Store Atomicity guarantees Serializability
### Instruction Reordering

<table>
<thead>
<tr>
<th>2\textsuperscript{nd} →</th>
</tr>
</thead>
<tbody>
<tr>
<td>1\textsuperscript{st} ↓</td>
</tr>
<tr>
<td>+,...</td>
</tr>
<tr>
<td>Br</td>
</tr>
<tr>
<td>L y</td>
</tr>
<tr>
<td>S y, w</td>
</tr>
<tr>
<td>Fence</td>
</tr>
<tr>
<td>+,...</td>
</tr>
<tr>
<td>indep</td>
</tr>
<tr>
<td>indep</td>
</tr>
<tr>
<td>indep</td>
</tr>
<tr>
<td>indep</td>
</tr>
<tr>
<td>✓</td>
</tr>
<tr>
<td>×</td>
</tr>
<tr>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
</tr>
<tr>
<td>Br</td>
</tr>
<tr>
<td>✓</td>
</tr>
<tr>
<td>×</td>
</tr>
<tr>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
</tr>
<tr>
<td>L x</td>
</tr>
<tr>
<td>indep</td>
</tr>
<tr>
<td>indep</td>
</tr>
<tr>
<td>indep</td>
</tr>
<tr>
<td>x≠y</td>
</tr>
<tr>
<td>×</td>
</tr>
<tr>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
</tr>
<tr>
<td>S y, w</td>
</tr>
<tr>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
</tr>
<tr>
<td>x≠y</td>
</tr>
<tr>
<td>x≠y</td>
</tr>
<tr>
<td>×</td>
</tr>
<tr>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
</tr>
<tr>
<td>Fence</td>
</tr>
<tr>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
</tr>
<tr>
<td>×</td>
</tr>
<tr>
<td>✓</td>
</tr>
</tbody>
</table>
Programming Language viewpoint

Pointers and array indices give rise to dependent loads; these operations must be ordered.

\[ r_1 = L \times \]
\[ r_2 = L[r_1] \]
\[ r_3 = r_2 + 1 \]
\[ S[r_1], r_3 \]

Flow of register state reflected in edges of graph; implicit register renaming
Address Speculation

Speculation = any decision which may break the rules down the line.

Here we relax the reordering axioms.

Behavior consistent with Store Atomicity observed by [Martin, Sorin, Cain, Hill, Lipasti 01]

S r 7 and L y are ordered if r = y

Non-speculative execution must wait until r has been computed.

Speculation assumes r ≠ y; if this fails, discard the execution.
Optimizations Are Tricky

Thread 1
S x 0
r₁ = L x = 2
r₂ = L x
if (r₁ = r₂)
S y 2

Thread 2
S y 0
r₃ = L y = 2
S x, r₃

- Ban invention of values “out of thin air”
- Permit any other imaginable optimization

[Manson, Pugh, Adve 05]
TSO is Non-Atomic

- Satisfy some Loads with local Stores
- Memory order ignores them
- Makes model non-atomic
Transactional Serializability

Serialize instructions in transaction together.
- Clearly atomic
- Too strong; can’t interleave independent operations

Disallowed executions actually are ok for this example!
Ordering and transactions

Predecessor operations precede the start of a transaction

Successor operations follow the end of a transaction
Enumeration of legal behaviors

- Find *all* legal behaviors
  - Must get the edges right

- Find *one* legal behavior
  - Can impose unnecessary ordering
  - Example: invalidation-based cache
Choosing a candidate Store

Candidate stores for a Load must be:
- To same address as that Load
- Resolved
- Not overwritten

Guarantees Store Atomicity is maintained
Store Atomicity
Summary

- High-level unifying property for memory consistency protocols
- Separation between processor local, memory behavior
- Captures ordering dependencies which *must* be enforced by memory system
- A memory model with no memory
Thanks!

JanWillem.Maessen@sun.com
Implications / Applications

- Address Speculation, new behaviors but no violation of Store Atomicity (SA)
- Non-atomic models, e.g., TSO
- Properly synchronized programs
- Java Memory Model
- Transactional memory
Permit Aliasing Speculation

- New behaviors do not violate Store Atomicity
- Exploited by current architectures
- Banning complicates reordering
  - Dependency from source of Store address to any subsequent Load/Store
Overview

- Serializability, graphs
- Instruction Reordering
- Store Atomicity
- Enumerating behaviors operationally

Putting Store Atomicity to use
- Address aliasing speculation
- TSO
Drawbacks of TSO

- Complicates memory model
  - Two kinds of source edges—local, non-local
  - Must track interaction of these orderings
  - Definition of candidates(L) is subtle

- Problem on multi-core architectures
  - Separate Load/Store buffer per thread
  - Each must be large to tolerate latency

Avoid any model which treats some threads differently from others
Multithreaded Languages

- Discipline programmer must follow
  - Locks in well-synchronized programs
  - Use of synchronized and volatile in the Java™ Programming Language
- Obey discipline $\rightarrow$ Atomicity (SC)
- Every model has an atomic aspect:
  - Lock ordering
  - Volatile variables
Looking ahead

- Exploit flexible ordering constraints
  - Cache protocols
  - Cross-thread speculation
- Transactional memory
  - Serialization which reflects practice
- Programmer-level memory models
  - Well-synchronized programs
  - Implement language-level models in Store Atomic setting
Programmer’s view
High-level vs. low-level models

- Store Atomicity is a very low-level property
  - Specifies what happens
  - No intuition about “how to program”

- Programmer-level models are important
  - Give a discipline for programming
  - Strong model (SC) within discipline
  - Hope: can check compliance
  - Example: Properly synchronized programs
Well synchronized programs
[Adve, Hill 90] [Keleher, Cox, Zwaenepoel 92]

- Divide the variables in two classes: synchronization variables and the rest
- In a well synchronized program a non-synchronizing Load L has only one element in candidates(L)!

Atomicity edges can be grouped and drawn lazily
# Instruction Reordering

<table>
<thead>
<tr>
<th>2&lt;sup&gt;nd&lt;/sup&gt; → 1&lt;sup&gt;st&lt;/sup&gt; ↓</th>
<th>+,...</th>
<th>L y</th>
<th>S y, w</th>
<th>Fence</th>
<th>Trans</th>
<th>Commit</th>
</tr>
</thead>
<tbody>
<tr>
<td>+,...</td>
<td>indep</td>
<td>indep</td>
<td>indep</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>L x</td>
<td>indep</td>
<td>indep</td>
<td>x≠y</td>
<td>×</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>S y, w</td>
<td>✓</td>
<td>x≠y</td>
<td>x≠y</td>
<td>×</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>Fence</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Trans</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>N/A</td>
<td>×</td>
</tr>
<tr>
<td>Commit</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Partial order (dag) ≤<sub>local</sub> on local instructions.
Resolving Transactional Loads in Parallel

We resolve a load in both transactions

Observed Stores overwritten

Results in a cycle between transactions

Roll back some Load which breaks cycle

Bad speculation introduces cycle
Roll back Load which break cycle
Along with its direct dependencies