Constructing a Weak Memory Model

Sizhuo Zhang¹, Muralidaran Vijayaraghavan¹, Andrew Wright¹, Mehdi Alipour², Arvind¹
¹MIT CSAIL ²Uppsala University
ISCA 2018, Los Angeles, CA
06/04/2018
The Specter of Weak Memory Models

Two major questions on weak memory models

- Do weak memory models improve PPA (performance/power/area) over strong models? **Highly controversial**

- Is there a common semantic base for weak memory models? **This paper answers this question**

Architects find SC constraining, and have unleashed the specter of weak memory models to the world.
Importance of a Common Base for Weak Memory Models

- Even experts cannot agree on the precise definitions of different weak models, or the differences between them
  - Example: Researchers have been formalizing the definitions of commercial weak memory models empirically (i.e., come up with models that match the observed behaviors on commercial machines)
  - With more experiments being performed on the commercial machines, either unexpected behaviors show up, or expected behaviors never arise: Keep revising the model
    - POWER: [PLDI 2011] -> [TOPLAS 2014]
    - ARM: [POPL 2016] -> [POPL 2018]

A common base model helps us understand the nature of weak models without being drowned in the details of commercial machines or ISAs

[PLDI 2011] Understanding POWER Multiprocessors
[TOPLAS 2014] Herding cats: Modelling, Simulation, Testing, and Data-mining for Weak Memory
[POPL 2016] Modelling the ARMv8 Architecture, Operationally: Concurrency and ISA
[POPL 2018] Simplifying ARM Concurrency: Multicopy-Atomic Axiomatic and Operational Models for ARMv8
Executive Summary

Our approach to construct the base memory model, **GAM** (General Atomic Memory Model)

- Step 1: Minimum ordering constraints of a uniprocessor
- Step 2: Additional constraints when processors are connected by an *atomic* shared memory system*

*An atomic memory system can be abstracted to monolithic memory which executes loads and stores instantaneously and forms a natural total order*

- Step 3: *Optional* ordering constraints to match programmers’ expectations

... while ensuring equivalent axiomatic and operational definitions

Performance evaluation

- GAM has similar performance to other weak models
- Some optional ordering constraints have little impact on performance
Meaning of Reorderings in Uniprocessor

- **Commit order**: order of instructions being committed from ROB
- **Execution order**: order of times when instructions finish execution
  - A non-memory instruction finishes execution by doing its computation
  - Load finishes execution by reading its value from L1 or SB
  - Store finishes execution by writing data into L1
    - If the store has forwarded the data to a younger load, then the load appears before the store in the execution order

**Execution order differs from commit order**

- Commit order: order of instructions being committed from ROB
- Execution order: order of times when instructions finish execution

- **Fetch**
  - Load Buffer (LB)
  - Store Buffer (SB)
  - L1 cache
  - Memory

- Reorder Buffer (ROB)

- Load req/resp
- Store req/resp
Uniprocessor Ordering Constraints

Ordering constraints are derived from two aspects:

- To maintain single-thread correctness, some memory instructions for the same address must be ordered.
- The execution of some instructions cannot start until older instructions have finished execution:
  - A speculative stores cannot be sent to the memory system.
  - No instruction can start execution until its source operands are ready (rules out value speculation).
Uniprocessor Constraints for Same-Address Memory Instructions

**GAM Constraint:** A younger store cannot be reordered with an older memory instruction to the same address.

- I1: \( r1 = \text{Ld}[a] \)
  - I2: \( \text{St}[a] 1 \)

- I1: \( r1 = \text{St}[a] 1 \)
  - I2: \( \text{St}[a] 2 \)

- I1: \( r1 = \text{Ld}[a] \)
  - I2: \( r2 = \text{Ld}[a] \)
  - Crossed out

- I1: \( r1 = r2 + 1 \)
  - I2: \( \text{St}[a] r1 \)
  - I3: \( r1 = \text{Ld}[a] \)

**No ordering constraints for two loads for the same address.**

**GAM Constraint:** A younger load cannot be reordered with the instruction that produces the address or data of the immediately preceding store for the same address.

- I1: \( r1 = \text{Ld}[a] \)
  - I2: \( \text{St}[a] \textbf{r1} \)
  - Crossed out
  - Forward

- I1: \( r1 = \text{Ld}[a] \)
  - I2: \( r2 = \text{Ld}[a] \)
  - Crossed out
Uniprocessor Constraints for Starting Execution

- **GAM Constraint**: Cannot reorder instructions with register read-after-write dependencies

- **GAM Constraint**: A younger store cannot be reordered with any older branch

- **GAM Constraint**: A younger store cannot be reordered with an instruction that produces the address of any older memory instruction

I1: \[ r1 = \text{Ld}[a] \]
I2: \[ r2 = \text{Ld}[r1] \]
I1: if(r1==0) goto somewhere
I2: \[ \text{St}[a]~1 \]
I1: \[ r1 = 1+99 \]
I2: \[ r2 = \text{Ld}[a+r1] \]
I3: \[ \text{St}[a]~1 \]
Additional Constraints on Load Values in Multiprocessor

A load can get its value either from memory or by bypassing from SB

- Reading from Memory: value is determined by atomic memory order
  - Atomic memory order must be consistent with the execution order of each processor
- Bypassing from SB:
  - The forwarding store is the immediately preceding store for the same address in commit order
  - The load always appears before the forwarding store in the execution order

Execution order must also incorporate constraints imposed by fences

Atomic memory system can be abstracted to a monolithic memory which gives a total (Atomic) Memory Order on all instructions that access memory
Define *Global memory order* as the order of execution finish times of all memory instructions

- This is the join of all execution orders and the atomic memory order of all memory instructions
- One load-value constraint to cover both cases (similar to RMO and Alpha)
Programmers’ Intuition: Same Address Loads

Three choices for the reordering of two consecutive loads for the same address – major difference among RMO, ARM, and GAM

- No ordering constraint, e.g., RMO: does not match programmers’ intuition
- Only allow reordering when two loads read from the same store, e.g., ARM:
  - Matches programmers’ intuition in the shown example
  - However, can lead to confusing behaviors in other cases (see paper for the example)
- Cannot be reordered, e.g., GAM: avoid all confusing behaviors

<table>
<thead>
<tr>
<th>Proc. 1</th>
<th>Proc. 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>I1: St [a] 1</td>
<td>I2: r1 = Ld [a] (=1)</td>
</tr>
<tr>
<td>I2: r2 = Ld [a] (=0)</td>
<td></td>
</tr>
</tbody>
</table>

Programmers do not expect the second load to return a stale value
Performance Overheads in Ordering Same-Address Loads

**GAM**
- Stall load issue (to start execution) if the load cannot forward from a store younger than an unissued older load.
- When a load is issued, kill younger loads which have been issued to memory or have got data forwarded from a store older than the issuing load.

**ARM**
- When a load finishes loading memory, kill younger loads whose values have been overwritten by other processors.
- Optionally stall load issue as in GAM.

**Single-thread application is enough if the goal is to show that the performance overheads of GAM is negligible**
- Kills and stalls in GAM are not caused by stores from other processors.
- Best case for ARM: no overwrite by other processor, so no kill.
Evaluation Results of Same-Address Load-Load Ordering

- Modelling RMO, ARM and GAM using GEM5
  - Haswell-like OOO core
  - All SPECCPU ref inputs: take 10 checkpoints for each input, simulate 100M instructions from each checkpoint

<table>
<thead>
<tr>
<th></th>
<th>Average</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM</td>
<td>0.2%</td>
<td>2.3%</td>
</tr>
<tr>
<td>RMO</td>
<td>0.2%</td>
<td>2.3%</td>
</tr>
</tbody>
</table>

Stalls/Kills by same-address load-load ordering (number of events per 1000 micro-ops)

<table>
<thead>
<tr>
<th></th>
<th>Average</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kills in GAM</td>
<td>0.2</td>
<td>3.24</td>
</tr>
<tr>
<td>Issue stalls in GAM</td>
<td>0.19</td>
<td>2.15</td>
</tr>
<tr>
<td>Issue stalls in ARM</td>
<td>0.19</td>
<td>2.15</td>
</tr>
</tbody>
</table>

- Performance impact caused by same-address load-load ordering is negligible
Programmers’ Intuition: Data-Dependent Loads

<table>
<thead>
<tr>
<th>Proc. 1</th>
<th>Proc. 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>I1: St [a] 1</td>
<td>I3: ( r1 = Ld [b] (=a) )</td>
</tr>
<tr>
<td>I2: FenceSS</td>
<td>I4: ( r2 = Ld [r1] (=0) )</td>
</tr>
<tr>
<td>I3: St [b] a</td>
<td></td>
</tr>
</tbody>
</table>

All memory models except Alpha forbids this behavior

Programmers expect data-dependent loads to be ordered
- GAM has already enforced this ordering

Enforcing this ordering may bring restrictions on implementations
- Value prediction
- Load-to-load forwarding
- Coherence protocols that delay cache invalidations (e.g. VIPS, Tardis)

Comprehensive performance evaluation is hard
- Implementation restrictions vs. extra fences
Equivalent Axiomatic and Operational Models of GAM

- **Axiomatic model**
  - Constraints $\rightarrow$ axioms

- **Operational model**
  - Models speculative execution
  - Memory accesses are instantaneous

**Diagram**:
- Proc 1. ROB
- ... (omitted)
- Proc n. ROB
- Monolithic Memory
Conclusion

- Constructed the common base memory model, GAM
  - RMO, ARM and RISC-V differ in same-address load-load ordering
  - Alpha differs in data-dependency ordering
- Constraint on same-address load-load ordering is a major difference between these memory models, but it has little impact on performance
- Data-dependency ordering is a feature expected by programmers, but it may constrain implementations; its performance implication needs further evaluation
Questions?