Data consistency check of *very large* execution traces

SHOARAAEE Suzanne <suzanne.shoaraee@arm.com>
Verification engineer, ARM France

Dagstuhl, Germany
Concurrency with Weak Memory Models seminar
November 2016

©ARM 2016
ARM in a few words

- Global leader in the development of semiconductor IP:
  - 12 billion chips shipped in 2014
  - 390 ARM chips are shipped each second!

- Delivers full stack solutions:
  - Software: optimized ecosystem from IoT to servers
  - CPUs and GPUs: designed for specific tasks
  - System: delivering integrated multi-core systems
  - Physical technology: optimized processes for foundries

- Founded in 1990 in Cambridge (UK), now employing 4000+ people in 40 offices all over the world
I. Which problems are we trying to solve?

II. The beginning of a data consistency checker

III. Where are we today?

IV. What are the next challenges?
Our goal

- Ensure that **CPU implementations** are **compliant** with the **ARM memory model** described in the ARM Architecture

- Detect data corruption and help debugging

![Diagram showing the process flow from Processor design simulation, through Execution trace, to Data Consistency Checker]

© ARM 2016
Our goal

Constraints:

Be able to **handle very large execution traces**
Up to hundreds of millions of memory accesses!

The checker must be:

- **Efficient** with reasonable runtime and memory consumption
- **Generic** and **reusable** (black box approach)
Starting point

- The ARM Architecture describes memory ordering rules in **English**:
  “For a single-copy atomic store, if the store overlaps another single-copy atomic store, then all of the writes from one of the stores are inserted into the Coherence order of each overlapping byte before any of the writes of the other store are inserted into the Coherence orders of the overlapping bytes.”

- Need to **formalize** the English sentences into an implementable model

- After few trials and as many failures, we decided to look for **existing** formalizations
Existing ARM memory model formalizations

- Several approaches:
  - **Operational**
    - Abstractions of actual machines, composed of idealised hardware components such as buffers and queues
  - **Axiomatic**
    - A graph-based representation of relations on memory accesses, verifying axioms to distinguish allowed allowed behaviours from forbidden behaviours.

  We decided to start from the axiomatic model from Jade Alglave et al.
Axiomatic formalization

- However, the ARM memory model has multiple facets
  Its formalization is **non trivial**

1. **SC per location**: acyclic(po U com)
   with \( \text{com} = \text{co U rf U fr} \)
   \( \text{fr} = \{(r,w,l) | \exists w0.(w0, r) \in \text{rf} \land (w0,w,l) \in \text{co}\} \)

3. **No thin air**: acyclic(hb)
   with \( \text{hb} = \text{ppo U bar U rfe} \)

4. **Observation**: irreflexive(fre; prop; hb³)
   with \( \text{A-cumul} = \text{rfe; bar} \)
   and \( \text{prop-base} = (\text{bar U A-cumul}); \text{hb}^* \)
   and \( \text{prop} = (\text{prop-base} \cap \text{WW}) \cup (\text{com}^*; \text{prop-base}^*; \text{bar}; \text{hb}^*) \)

5. **Propagation**: acyclic(co U prop)
First challenge: Incorporate additional requirements
Axiomatic formalization

- We wanted to adapt the axioms to our additional requirements:
  - Consider barrier \textit{operation type} information
  - Consider barrier \textit{shareability domain} information
  - Take into account \textit{timestamps} information
  - Handle \textit{load-acquire} and \textit{store-release} instructions
  - ...

Barrier operation type information

- DMB and DSB may apply to all type of accesses or only to loads or stores

\[ \text{op type} = \text{all} \quad \text{op type} = \text{store} \quad \text{op type} = \text{load} \]

\[
\begin{align*}
\text{op type} = \text{all} & : & \text{R}[x] \quad \text{W}[y] \quad \text{BAR} \quad \text{R}[z] \\
& & \downarrow \quad \downarrow \quad \downarrow \\
& & \text{W}[x] \\
\text{op type} = \text{store} & : & \text{R}[x] \quad \text{W}[y] \quad \text{BAR} \\
& & \downarrow \quad \downarrow \\
& & \text{W}[x] \\
\text{op type} = \text{load} & : & \text{R}[x] \quad \text{W}[y] \\
& & \downarrow \\
& & \text{W}[x]
\end{align*}
\]
Barrier shareability domain information

- The barrier may only have effects on a given set of agents and impact the validity of the execution

<table>
<thead>
<tr>
<th>Agent 1</th>
<th>Agent 2</th>
<th>Agent 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$W[x] = 0$</td>
<td>$W[x] = 1$</td>
<td>$W[y] = 2$</td>
</tr>
<tr>
<td>$W[y] = 2$</td>
<td>$R[Y] = 2$</td>
<td>$R[y] = 2$</td>
</tr>
<tr>
<td>$R[x] = 0$</td>
<td>$R[x] = 0$</td>
<td>$R[x] = 0$</td>
</tr>
</tbody>
</table>

**Shareability domain** | **Valid execution**
--- | ---
All agents | No
Agent 1 + 2 | Yes
Timestamp information

- It is possible to define **integer boundaries** for a memory access:
  - A lower boundary, called *begin timestamp* and noted *ts_b*
  - An upper boundary, called *end timestamp* and noted *ts_e*

- Such as:
  - At *ts_b*, the memory access "doesn't have any effect"
  - At *ts_e*:
    - a write is guaranteed to be observable by all agents
    - a read *R* won't be affected by subsequent writes to the location
    - a barrier has its effects visible, i.e. reads and writes before the barrier in program order and reads and stores from other agents observed by the agent who executes the barrier (group A) are observable
Timestamp information

- In other words:
  - For a given memory access $A$:
    \[ ts_b(A) \leq ts_e(A) \]
  - For any pair of read write accesses $(R, W)$:
    \[ ts_b(R) > ts_e(W) \Rightarrow \text{observe}(R, W) \]
  - For any pair of read write accesses $(R, W)$:
    \[ ts_b(W) > ts_e(R) \Rightarrow \sim RF(W, R) \]
    a valid read-from candidate $W$ verifies $ts_b(W) \leq ts_e(R)$
  - For any write accesses before a barrier in program order $(W, B)$:
    \[ \exists \ ts_e(W) \text{ such as } ts_e(W) \leq ts_e(B) \]

It exists several valid boundaries respecting the previous conditions
Timestamp information

Example of timestamp information **influencing** the **execution compliance**

<table>
<thead>
<tr>
<th>ts_b</th>
<th>ts_e</th>
<th>Agent 0</th>
<th>Agent 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>R[x] = 1</td>
<td>W[x] = 0</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>W[x] = 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\begin{array}{|c|c|c|c|}
\hline
\text{Timestamp information} & \text{Valid execution} \\
\hline
\text{No} & \text{Yes} \\
\text{Yes} & \text{No} \\
\hline
\end{array}
\]

\[ ts_b(W[x] = 1) > ts_e(R[x] = 1) \Rightarrow \neg RF(W[x] = 1, R[x] = 1) \]

- Reduce the observation space
- Invalidate the scenario
Adapting existing formalizations is not easy but it’s not the only issue...
I. Which problems are we trying to solve?

II. The beginning of a data consistency checker

III. Where are we today?

IV. What are the next challenges?
**Problem:** The formalization is generally used with small size scenarios. We have to handle several millions of memory accesses.
Idea about the explosion

- Several prototypes considering all memory accesses together

<table>
<thead>
<tr>
<th>Implementation</th>
<th>4 axioms</th>
<th>2 axioms</th>
</tr>
</thead>
<tbody>
<tr>
<td>First C++ prototype</td>
<td>Few dozen accesses</td>
<td>x</td>
</tr>
<tr>
<td>Second C++ prototype</td>
<td>x</td>
<td>A couple of thousand simple accesses (Dhrystone)</td>
</tr>
<tr>
<td>Gecode solver prototype</td>
<td>~ 20 accesses</td>
<td>~ 40 accesses</td>
</tr>
</tbody>
</table>

- Exponential complexity in the C++ prototypes
How to solve the scalability issue?

- Improving the implementation?
  - Not enough, exponential complexity

- Reducing graph size?

- Cutting the input trace?
  - Cleanly (barrier…)?
  - Complicated, not enough and scenario dependent
  - Arbitrarily (window checking)?
  - With a byte granularity
Reducing graph size
Reducing graph size

Reduce vertices number:

Vertices without any “external” relation can be discarded. By “external” relation, we intend linked with a vertex representing a memory access performed by another agent.

From runtime validation of memory ordering using constraint graph checking

Kaiyu Chen, Sharad Malik and Priyadarsan Patra

Need to be refined, no successful implementation expected in short-term.
Reducing graph size

Dynamic graph slicing:
If a sub-graph in the constraint graph can be identified to not have an incoming edge from subsequent instructions, this sub-graph can be pruned since it can never participate in a cycle that involves instructions executed in the future.

From runtime validation of memory ordering using constraint graph checking

Kaiyu Chen, Sharad Malik and Priyadarsan Patra
Cutting the input trace
Arbitrarily cutting the input trace

Test execution

Agent 1   Agent 2

Execution start

Window checking

Execution end

Only (but major) issue:
We may not be able to explain some value read

Example:

<table>
<thead>
<tr>
<th>Agent 1</th>
<th>Agent 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>W[x] = 1</td>
<td></td>
</tr>
<tr>
<td>W[y] = 2</td>
<td></td>
</tr>
<tr>
<td>R[y] = 2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>? Cutting border</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>R[x] = 1</td>
</tr>
</tbody>
</table>

Cutting border
Cutting the input trace with a byte granularity

- **Issue**: we are not able anymore to detect failures involving several addresses (atomicity, barriers, …)

Example: MP + DMBs:

- But… drastically reduces the number of memory accesses to be considered together

- And it’s much simpler!
Check with a byte granularity: partial but scalable
I. Which problems are we trying to solve?

II. The beginning of a data consistency checker

III. Where are we today?

IV. What are the next challenges?
The current checker

- Relies on clear and rigorously defined principles
- Easy to maintain code, no false fails, possibility to relax constraints on timestamps
- Reduced memory consumption: very few memory exhaustion scenarios found to date
- Limited execution time
- Easy replay on specific addresses
Glimpse of the checker concepts

INPUT

Agent 1
(1) @x W2
(2) @x R2
(3) @x R3
...

Agent 2
(4) @x W3
(5) @x R4
...

Agent n
(6) @x W4
...

Execution traces (byte granularity)

OUTPUT

The checker tries to find a sequential consistent scheduling for memory accesses from the different agents

(1) @x W2
(2) @x R2
(4) @x W3
(3) @x R3
(6) @x W4
(5) @x R4
...

A sequence has been found, we are able to explain the values read

All possible sequences have been explored and none explain the value read: bug found!
Glimpse of the checker concepts

- Example:

**INPUT**
Let’s consider an address byte $x$

- (1) R4
- (2) R5
- (1) W4
- (2) W5

**INTERNAL**

<table>
<thead>
<tr>
<th>Agent 1</th>
<th>Agent 2</th>
<th>Sequence found</th>
<th>Value currently observable</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) R4</td>
<td>(3) W4</td>
<td>Empty</td>
<td>Initial memory value (£ 4)</td>
</tr>
<tr>
<td>(2) R5</td>
<td>(4) W5</td>
<td>(3) W4</td>
<td>4</td>
</tr>
<tr>
<td>(2) R5</td>
<td>(4) W5</td>
<td>(3) W4 (1) R4</td>
<td>4</td>
</tr>
<tr>
<td>(2) R5</td>
<td>(4) W5</td>
<td>(3) W4 (1) R4</td>
<td>5</td>
</tr>
<tr>
<td>(2) R5</td>
<td>(4) W5</td>
<td>(3) W4 (1) R4</td>
<td>5</td>
</tr>
</tbody>
</table>
Few facts

- Usable from the **early stage** of the CPU development projects

- Able to **check** up to **hundreds millions of memory accesses** thank to an **optimized** implementation
  
  Ex: ~6h runtime to check a Linux boot + small app
  
  - Easy check, easy scenario
  
  - Long time spent retrieving the data (accesses for each address)

- Generally **~seconds of check** for several hours simulation, **few MB memory consumption**

- Allow to **find many consistency bugs** during development phases on:
  - Hardware
  - TestBenches
  - Stimuli
Checker limitations

- The checker doesn’t support hot start

- The checker is able to work without timestamps but will likely explode on large input traces

- Even with timestamps, the check cost may become important depending on the test

- **Barriers** (DMB/DSB) could be taken into account only if they come with an end timestamp
  - Having end timestamp for DMB seems challenging!

- **Barriers** and “Half-barriers” (DMB STR, LDAR/STLR, …) aren’t supported yet
Still a lot to explore and implement...
I. Which problems are we trying to solve?

II. The beginning of a data consistency checker

III. Where are we today?

IV. What are the next challenges?
Widen the checks with further constraints

- Take into account **constraints across distinct bytes**
- Example of single-copy atomic stores:
  
  “For a single-copy atomic store, if the store overlaps another single-copy atomic store, then all of the writes from one of the stores are inserted into the Coherence order of each overlapping byte before any of the writes of the other store are inserted into the Coherence orders of the overlapping bytes.”

*Extract from ARM Architecture*
Take into account the atomicity

- Currently, **each byte** is **individually** considered
- Following scheduling are perfectly valid:

```
<table>
<thead>
<tr>
<th>WRITE 1</th>
<th>WRITE 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>B5</td>
</tr>
<tr>
<td>B2</td>
<td>B6</td>
</tr>
<tr>
<td>B3</td>
<td>B7</td>
</tr>
<tr>
<td>B4</td>
<td>B8</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Addr x+1</th>
<th>Addr x+2</th>
<th>Addr x+3</th>
</tr>
</thead>
<tbody>
<tr>
<td>B2</td>
<td>B3</td>
<td>B4</td>
</tr>
<tr>
<td>B5</td>
<td>B6</td>
<td>B7</td>
</tr>
<tr>
<td>B5</td>
<td>B6</td>
<td>B7</td>
</tr>
<tr>
<td>B2</td>
<td>B3</td>
<td>B4</td>
</tr>
</tbody>
</table>
```

*Without atomicity constraints*
Take into account the atomicity

- Up to 1,000,000,000 accesses
- Up to 256 agents
- Up to 8 overlapping bytes under single-copy atomicity constraint!

- To explore for several hundreds of thousands/millions single-atomic accesses…

- Same for loads’ single-copy atomicity and multi-copy atomicity
Atomic instructions
Atomic instructions

- For now, atomic instructions are considered as several totally independent instructions:

  “LDADD – atomic add of a location with value in a register, with original data loaded into a register. Each instruction can have one of four possible orderings - acquire, release, acquire&release, no order, as it performs both a load and a store.”

- Considered as:
  \[ R[x] = Y \]
  \[ W[X] = ? \]

- Atomic instructions can be executed in different places of the memory system:
  - **Near atomic**: executed locally (ex: for a cacheable atomic that hits in Unique in the L1 data cache)
  - **Far atomic**: instruction forwarded externally
    - The values written by far atomics may be unknown!
Atomic instructions

- In the current checker:

  1. The value written by the atomic instruction may be unknown

      ➡ Better than nothing: still allows to check that only one value is “assigned” to the unknown
      ➡ But unknown writes bring uncertainty and may hide potential issues (“joker”)
      ➡ No solution for some kind of atomic instructions

  2. The checker may schedule other accesses between the two accesses supposed to be atomic

      ➡ Could be improved
Barriers
What about barriers?

- Barriers are completely excluded from the current consistency checker
- The problem is even trickier as many barrier flavours exist:
  - Half-barriers: Load-acquire / Store-release
  - Data Memory Barrier: DMB
  - Data Synchronization Barrier: DSB
  - With different shareability domains
- How to check barriers?
  - Compute valid schedulings for each byte address under barrier’s effect
  - Intersect the set of valid scheduling for all byte addresses
Questions??