PipeCheck Hands-On
Overview

- Will take you through modelling simple uarches in µSpec
- Begin by modelling an SC uarch
  - Partially completed uarch in VM, you will fill in remainder
- Post-coffee break, will look at expanding this SC uarch to TSO
  - Store buffers
  - Reading own write early (time permitting)
  - Fences (time permitting)
Specifying a Simple (SC) Microarch. in µSpec
Specifying a Simple (SC) Microarch. in µSpec

Core 0
- Fetch
- Execute
- Writeback

Core 1
- Fetch
- Execute
- Writeback

Memory Hierarchy

3-stage in-order pipelines
Specifying a Simple (SC) Microarch. in µSpec

Memory Hierarchy

Core 0

Fetch

Execute

Writeback

Load access
Mem in Execute stage

Core 1

Fetch

Execute

Writeback
Specifying a Simple (SC) Microarch. in µSpec

Memory Hierarchy

Core 0
- Fetch
- Execute
- Writeback

Core 1
- Fetch
- Execute
- Writeback

Stores sent to Memory after Writeback

Stores sent to Memory after Writeback
Specifying a Simple (SC) Microarch. in µSpec

1. Start VirtualBox VM

2. Open a Terminal

3. Partially completed SC uarch in 
/home/check/pipecheck_tutorial/uarches/SC_fillable.uarch
µSpec: A DSL for Specifying Microarchitectures

- Language has capabilities similar to first-order logic
  - forall, exists, AND (\&\&), OR (\|), NOT (~), implication (\Rightarrow)

- Microarchitecture spec has three components:
  - Stage identifier definitions
  - Macro definitions (optional)
  - Axiom definitions

- Macros allow:
  - decomposition of axioms into smaller parts
  - reuse of uspec fragments
µSpec: A DSL for Specifying Microarchitectures

- Axioms are each a **partial** ordering on the events in an execution
- Job of PipeCheck is to ensure that these axioms correctly work **together** to uphold the requirements of the ISA-level MCM
- Can reference nodes and edges in an axiom
  - EdgeExists ((i1, Fetch), (i2, Fetch)) =>
    AddEdge ((i1, Execute), (i2, Execute), "PPO")
- µSpec also has predicates for various architecture-level properties
  - SameCore <instr1> <instr2>
  - SamePhysicalAddress, SameData, IsAnyRead, ProgramOrder,...
Finding Axioms

Core 0

Fetch

Execute

Writeback

Core 1

Fetch

Execute

Writeback

Memory Hierarchy
Finding Axioms

Stores go through the pipeline stages and reach memory in order.

Memory Hierarchy
The Writes_Path Axiom

Axiom "Writes_Path":
forall microops "i",
IsAnyWrite i =>
AddEdges
[((i, Fetch), (i, Execute), "path")];
[((i, Execute), (i, Writeback), "path")];
[((i, Writeback), (i,(0,MemoryHierarchy)), "path")].

Memory Hierarchy
The Writes_Path Axiom

Axiom "Writes_Path":
for all microops "i",
IsAnyWrite i =>
AddEdges [((i, Fetch), (i, Execute), "path");
((i, Execute), (i, Writeback), "path");
((i, Writeback), (i,(0,MemoryHierarchy)), "path")].

Memory Hierarchy
The Writes_Path Axiom

Axiom "Writes_Path":
forall microops "i",
IsAnyWrite i =>
AddEdges [((i, Fetch), (i, Execute), "path");
((i, Execute), (i, Writeback), "path");
((i, Writeback), (i,(0,MemoryHierarchy)), "path")].

Microop: A single load/store op. May correspond to an ISA instr, or part of an ISA instr.
The Writes_Path Axiom

For all writes (note use of IsAnyWrite predicate)

Axiom "Writes_Path":
forall microops "i",
IsAnyWrite i =>
AddEdges [((i, Fetch), (i, Execute), "path");
((i, Execute), (i, Writeback), "path");
((i, Writeback), (i, (0,MemoryHierarchy)), "path").
The Writes_Path Axiom

Axiom "Writes_Path":
forall microops "i",
IsAnyWrite i =>
AddEdges [((i, Fetch), (i, Execute), "path");
((i, Execute), (i, Writeback), "path");
((i, Writeback), (i,(0,MemoryHierarchy)), "path")].
Axiom "Writes_Path": 
forall microops "i", 
IsAnyWrite i => 
AddEdges [((i, Fetch), (i, Execute), "path"); 
((i, Execute), (i, Writeback), "path"); 
((i, Writeback), (i,(0,MemoryHierarchy)), "path")].

Specify that there is only one memory hierarchy rather than one per core.
Specifying µSpec Nodes

- A node represents a particular event in a particular instruction’s execution.

- Format for nodes is: `(instr, ([core ID], stage/event_name))`

- Most of the time, core of a stage/event is core of the instruction
  - So the default `(instr, stage_name)` evaluates to...
  - `...(instr, (CoreOf instr, stage_name))`

- Thus, `(i, Fetch)` represents the fetch stage of instruction `i` on the core of instruction `i`...

- ...(and `(i, (0, MemoryHierarchy))` represents `i` reaching memory which is nominally on core 0
  - If `everyone` uses core 0 for memory hierarchy, this enforces a single memory hierarchy in the system (which is what we want).
Finding Axioms

Core 0
- Fetch
- Execute
- Writeback

Core 1
- Fetch
- Execute
- Writeback

Memory Hierarchy
Finding Axioms

Core 0
- Fetch
- Execute
- Writeback

Core 1
- Execute
- Writeback

Loads go through the pipeline stages in order

Memory Hierarchy
Finding Axioms

Core 0
- Fetch
- Execute
- Writeback

Loads go through the pipeline stages in order.

Reads Path Axiom is very similar to Writes Path axiom (no WB→MemHier edge).

Note: Reads don’t have a Memory Hierarchy node because they access memory during their Execute stage.
μhb Graphs for co-mp Using Axioms

Each column represents an instruction flowing through the pipeline

<table>
<thead>
<tr>
<th>Column</th>
<th>Instruction 1</th>
<th>Instruction 2</th>
<th>Instruction 3</th>
<th>Instruction 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td>Store [x] ←1</td>
<td>Store [x] ←2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Execute</td>
<td></td>
<td></td>
<td>r1 = Load [x]</td>
<td>r2 = Load [x]</td>
</tr>
<tr>
<td>Writeback</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MemHier</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Initially, Mem[x] = 0

SC Forbids: r1=2, r2=1, Mem[x] = 2
μhb Graphs for co-mp Using Axioms

Each column represents an instruction flowing through the pipeline

(i1) Fetch
(i2) Execute
(i3) Writeback
(i4) MemHier

Edges added according to Reads_Path axiom

Initially, Mem[x] = 0

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>i1: Store [x] ← 1</td>
<td>i3: r1 = Load [x]</td>
</tr>
<tr>
<td>i2: Store [x] ← 2</td>
<td>i4: r2 = Load [x]</td>
</tr>
</tbody>
</table>

SC Forbids: r1=2, r2=1, Mem[x] = 2
μhb Graphs for co-mp Using Axioms

Each column represents an instruction flowing through the pipeline.

- **Fetch**
  - (i1)

- **Execute**
  - (i2)
  - Edges added according to Writes_Path axiom

- **Writeback**
  - (i3)
  - Initially, Mem[x] = 0

- **MemHier**
  - (i4)

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>i1: Store [x] ← 1</td>
<td>i3: r1 = Load [x]</td>
</tr>
<tr>
<td>i2: Store [x] ← 2</td>
<td>i4: r2 = Load [x]</td>
</tr>
</tbody>
</table>

SC Forbids: r1=2, r2=1, Mem[x] = 2
Finding Axioms

Core 0

Fetch

Execute

Writeback

Core 1

Fetch

Execute

Writeback

Memory Hierarchy
All instructions on the same core go through Fetch in program order.
The PO_Fetch Axiom

Axiom "PO_Fetch":
forall microops "i1",
forall microops "i2",
SameCore i1 i2 /
\ ProgramOrder i1 i2 =>
AddEdge ((i1, Fetch), (i2, Fetch), "PO", "blue").
The PO_Fetch Axiom

Axiom "PO_Fetch":
forall microops "i1",
forall microops "i2",
SameCore i1 i2 \ ProgramOrder i1 i2 =>
AddEdge ((i1, Fetch), (i2, Fetch), "PO", "blue").

Use of predicates to check that instrs are on the same core and in program order
The PO_Fetch Axiom

Axiom "PO_Fetch":
for all microops "i1",
for all microops "i2",
SameCore i1 i2 \ ProgramOrder i1 i2 =>
AddEdge ((i1, Fetch), (i2, Fetch), "PO", "blue").

Add edge from Fetch stage of earlier instruction to Fetch stage of later instruction
μhb Graphs for co-comp Using Axioms

Each column represents an instruction flowing through the pipeline

(i1) (i2) (i3) (i4)

Fetch

Execute

Writeback

MemHier

Thread 0 | Thread 1
---|---
i1: Store [x] ← 1 | i3: r1 = Load [x]
i2: Store [x] ← 2 | i4: r2 = Load [x]

Initially, Mem[x] = 0

SC Forbids: r1=2, r2=1, Mem[x] = 2
μhb Graphs for co-mp Using Axioms

Each column represents an instruction flowing through the pipeline

(i1) Fetch 
(i2) Edges Added Using PO_Fetch axiom 
(i3) Execute 
(i4) Writeback 

Initially, Mem[x] = 0

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>i1: Store [x] ← 1</td>
<td>i3: r1 = Load [x]</td>
</tr>
<tr>
<td>i2: Store [x] ← 2</td>
<td>i4: r2 = Load [x]</td>
</tr>
</tbody>
</table>

SC Forbids: r1=2, r2=1, Mem[x] = 2
Finding Axioms

Core 0

- Fetch\textsubscript{i1}
- Execute\textsubscript{i1}
- Writeback

Core 1

- Fetch
- Execute
- Writeback

Memory Hierarchy
Finding Axioms

Core 0
- Fetch_{i_1}
  - Execute_{i_1}
    - Writeback
- Fetch_{i_2}
  - Execute_{i_2}

Core 1
- Fetch
  - Execute
  - Writeback

Memory Hierarchy
Finding Axioms

- If two instructions on the same core go through Fetch in order, they will go through Execute in the same order.
The Execute_Stage_Is_In_order Axiom

Axiom "Execute_stage_is_in_order":
forall microops "i1",
forall microops "i2",
SameCore i1 i2 /\
EdgeExists ((i1, Fetch), (i2, Fetch), "") =>
AddEdge ((i1, Execute), (i2, Execute), "PPO").
The Execute_STAGE_Is_In_order Axiom

Axiom "Execute_stage_is_in_order":
forall microops "i1",
forall microops "i2",
SameCore i1 i2 \/
EdgeExists ((i1, Fetch), (i2, Fetch), "") =>
AddEdge ((i1, Execute), (i2, Execute), "PPO").

If instructions on same core go through Fetch in order...
The Execute_Phase_Is_In_order Axiom

Axiom "Execute_stage_is_in_order":
forall microops "i1",
forall microops "i2",
SameCore i1 i2 /
EdgeExists ((i1, Fetch), (i2, Fetch), "") =>
AddEdge ((i1, Execute), (i2, Execute), "PPO").

...then they go through Execute in the same order.
Finding Axioms

Memory Hierarchy

Core 0

Fetch_{i_1}

Execute

Writeback_{i_1}

Core 1

Fetch

Execute

Writeback

Memory Hierarchy
Finding Axioms

Core 0

Fetch_{i1} → Execute → Writeback_{i1}

Core 1

Fetch → Fetch_{i2} → Execute → Writeback

Writeback_{i2}

Memory Hierarchy
If two instructions on the same core go through Fetch in order, they will go through Writeback in the same order.
The Writeback_Stage_Is_In_Order Axiom

If two instructions on the same core go through Fetch in order, they will go through Writeback in the same order.

Axiom "Writeback_stage_is_in_order":
for all microops "i1",
for all microops "i2",
______ i1 i2 \/
EdgeExists ((i1, ____), (i2, ____), "") =>
AddEdge ((i1, __________), (i2, __________), "PPO").
The Writeback Stage Is In Order Axiom

If two instructions on the same core go through Fetch in order, they will go through Writeback in the same order.

Axiom "Writeback_stage_is_in_order":
forall microops "i1",
forall microops "i2",
SameCore i1 i2 /
EdgeExists ((i1, Fetch), (i2, Fetch), "") =>
AddEdge ((i1, Writeback), (i2, Writeback), "PPO").
μhb Graphs for co-mp Using Axioms

Each column represents an instruction flowing through the pipeline

(i1) (i2) (i3) (i4)

Fetch
execute
writeback
memhier

(i1) Store [x] ← 1
(i2) Store [x] ← 2
(i3) r1 = Load [x]
(i4) r2 = Load [x]

Initially, Mem[x] = 0

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>i1: Store [x] ← 1</td>
<td>i3: r1 = Load [x]</td>
</tr>
<tr>
<td>i2: Store [x] ← 2</td>
<td>i4: r2 = Load [x]</td>
</tr>
</tbody>
</table>

SC Forbids: r1=2, r2=1, Mem[x] = 2
µhb Graphs for co-mp Using Axioms

Each column represents an instruction flowing through the pipeline

(i1) (i2) (i3) (i4)

Fetch

Execute

Writeback

MemHier

Thread 0

Thread 1

Initially, Mem[x] = 0

<table>
<thead>
<tr>
<th>Action</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>i1: Store [x]</td>
<td>1</td>
</tr>
<tr>
<td>i2: Store [x]</td>
<td>2</td>
</tr>
<tr>
<td>i3: r1 = Load [x]</td>
<td></td>
</tr>
<tr>
<td>i4: r2 = Load [x]</td>
<td></td>
</tr>
</tbody>
</table>

SC Forbids: r1=2, r2=1, Mem[x] = 2
μhb Graphs for co-mp Using Axioms

Each column represents an instruction flowing through the pipeline

(i1) Fetch
(i2) Execute
(i3) Writeback
(i4) MemHier

Edges from
Execute_stage_is_in_order & Writeback_stage_is_in_order

Initially, Mem[x] = 0

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>i1: Store [x] ← 1</td>
<td>i3: r1 = Load [x]</td>
</tr>
<tr>
<td>i2: Store [x] ← 2</td>
<td>i4: r2 = Load [x]</td>
</tr>
</tbody>
</table>

SC Forbids: r1=2, r2=1, Mem[x] = 2

Table:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>i1</td>
<td>Store [x] ← 1</td>
<td></td>
</tr>
<tr>
<td>i2</td>
<td>Store [x] ← 2</td>
<td></td>
</tr>
<tr>
<td>i3</td>
<td></td>
<td>r1 = Load [x]</td>
</tr>
<tr>
<td>i4</td>
<td></td>
<td>r2 = Load [x]</td>
</tr>
</tbody>
</table>
Finding Axioms

Core 0
Fetch
Execute
Writeback_{i_1}

Core 1
Fetch
Execute
Writeback_{i_2}

Memory Hierarchy
All writes to the same address must be totally ordered at the Memory Hierarchy. (coherence order)
Finding Axioms

All writes to the same address must be totally ordered at the Memory Hierarchy.
(coherence order)

Writeback_{i1}

Writeback_{i2}

i1: Store y=1

i2: Store y=2

Memory Hierarchy
Finding Axioms

All writes to the same address must be totally ordered at the Memory Hierarchy.
(coherence order)

Writeback$_{i_1}$

Coherence order:

Writeback$_{i_2}$

i1: Store y=1

OR

i2: Store y=2

Memory Hierarchy
Axiom "WriteSerialization":
\[
\forall \text{ microops } i_1, \forall \text{ microops } i_2, \\
( \sim (\text{SameMicroop } i_1 i_2) \land \text{IsAnyWrite } i_1 \\
\land \text{IsAnyWrite } i_2 \land \text{SamePhysicalAddress } i_1 i_2 ) \Rightarrow \\
(\text{EdgeExists } ((i_1, (0, \text{MemHier})), (i_2, (0, \text{MemHier})))) \lor \\
\text{EdgeExists } ((i_2, (0, \text{MemHier})), (i_1, (0, \text{MemHier})))).
\]
The WriteSerialization Axiom

Axiom "WriteSerialization":
forall microops "i1",
forall microops "i2",
( ~(SameMicroop i1 i2) \ IsAnyWrite i1 \ IsAnyWrite i2 \ SamePhysicalAddress i1 i2) =>
(EdgeExists ((i1, (0,MemHier)), (i2, (0,MemHier))) \ EdgeExists ((i2, (0,MemHier)), (i1, (0,MemHier)))).

Two different writes to the same address in the same order
The WriteSerialization Axiom

Axiom "WriteSerialization":
forall microops "i1",
forall microops "i2",
( ~(SameMicroop i1 i2) \ / IsAnyWrite i1
\ / IsAnyWrite i2 \ / SamePhysicalAddress i1 i2) =>
(EdgeExists ((i1, (0,MemHier)), (i2, (0,MemHier))) \ /
EdgeExists ((i2, (0,MemHier)), (i1, (0,MemHier)))).

Either i1 is before i2 in coherence order, OR vice-versa.
μhb Graphs for co-mp Using Axioms

WriteSerialization axiom

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>i1: Store [x] ← 1</td>
<td>i3: r1 = Load [x]</td>
</tr>
<tr>
<td>i2: Store [x] ← 2</td>
<td>i4: r2 = Load [x]</td>
</tr>
</tbody>
</table>

SC Forbids: r1=2, r2=1, Mem[x] = 2
μhb Graphs for co-mp Using Axioms

**WriteSerialization** axiom

Two solutions;
Each enumerated *separately*

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>i1: Store [x] ← 1</td>
<td>i3: r1 = Load [x]</td>
</tr>
<tr>
<td>i2: Store [x] ← 2</td>
<td>i4: r2 = Load [x]</td>
</tr>
</tbody>
</table>

SC **Forbids**: r1=2, r2=1, Mem[x] = 2
μhb Graphs for co-mp Using Axioms

WriteSerialization axiom

Two solutions;
Each enumerated separately

<table>
<thead>
<tr>
<th></th>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>i1</td>
<td>Store ([x] \leftarrow 1)</td>
<td>i3: (r_1 = \text{Load} [x])</td>
</tr>
<tr>
<td>i2</td>
<td>Store ([x] \leftarrow 2)</td>
<td>i4: (r_2 = \text{Load} [x])</td>
</tr>
<tr>
<td>SC</td>
<td>Forbids: (r_1=2, r_2=1, \text{Mem}[x] = 2)</td>
<td></td>
</tr>
</tbody>
</table>
μhb Graphs for co-mp Using Axioms

**WriteSerialization axiom**

Two solutions; Each enumerated separately

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>i1: Store (x) (\leftarrow) 1</td>
<td>i3: (r_1 = \text{Load } [x])</td>
</tr>
<tr>
<td>i2: Store (x) (\leftarrow) 2</td>
<td>i4: (r_2 = \text{Load } [x])</td>
</tr>
</tbody>
</table>

SC Forbids: \(r_1=2, r_2=1, \text{Mem}[x] = 2\)
μhb Graphs for co-mp Using Axioms

WriteSerialization axiom

Two solutions; Each enumerated separately

We will focus on left graph going forward

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>i1: Store [x] ← 1</td>
<td>i3: r1 = Load [x]</td>
</tr>
<tr>
<td>i2: Store [x] ← 2</td>
<td>i4: r2 = Load [x]</td>
</tr>
</tbody>
</table>

SC Forbids: r1=2, r2=1, Mem[x] = 2
Finding Axioms

i1: Store y=1

i2: Store y=2

Coherence order:

OR
Finding Axioms

Core 0
- Fetch
- Execute
- Writeback_{i1}

Core 1
- Fetch
- Execute
- Writeback_{i2}

Test requires $y_{\text{final}} = 2$

Coherence order:
OR

i1: Store $y=1$

i2: Store $y=2$

Memory Hierarchy
Finding Axioms

If a litmus test requires that an address has the value of a certain write at the end of the test, that write must be the last to reach the Memory Hierarchy.

Test requires $y_{\text{final}} = 2$

Coherence order:

i1: Store $y=1$

i2: Store $y=2$
If a litmus test requires that an address has the value of a certain write at the end of the test, that write must be the last to reach the Memory Hierarchy.

Coherence order:

\[ \text{i1: Store } y=1 \rightarrow \text{Writeback}_{i1} \rightarrow \text{Coherence order:} \rightarrow \text{Writeback}_{i2} \rightarrow \text{i2: Store } y=2 \]

Test requires \( y_{\text{final}} = 2 \)
Finding Axioms

If a litmus test requires that an address has the value of a certain write at the end of the test, that write must be the last to reach the MemoryHierarchy.

EnforceFinalWrite axiom in the \(\mu\)Spec

\[\text{Writeback}_{i_1}\]
\[\text{Writeback}_{i_2}\]

Test requires \(y_{\text{final}} = 2\)

Coherence order:

\(i_1: \text{Store } y=1\)

Enforced by test

\(i_2: \text{Store } y=2\)

Memory Hierarchy
Finding Axioms
A write must reach the Memory Hierarchy before memory instructions on the same core that are after the write in program order. (otherwise the write could be reordered with later writes or later reads)
A write must reach the Memory Hierarchy before memory instructions on the same core that are after the write in program order.

(Otherwise the write could be reordered with later writes or later reads)
A write must reach the Memory Hierarchy before memory instructions on the same core that are after the write in program order. (otherwise the write could be reordered with later writes or later reads)
The Enforce_Write_Ordering Axiom

A write must reach the Memory Hierarchy before execution of memory instructions that are after the write in program order.

Axiom "EnforceWriteOrdering":
for all microop "w",
for all microop "i",
(________ w /\ __________ w i) =>
AddEdge ((w, (0, __________)), (i, _____)).
The Enforce_Write_Ordering Axiom

A write must reach the Memory Hierarchy before execution of memory instructions that are after the write in program order.

Axiom "EnforceWriteOrdering":
forall microop "w",
forall microop "i",
(IsAnyWrite w \ ProgramOrder w i) =>
AddEdge ((w, (0, MemoryHierarchy)), (i, Execute)).
μhb Graphs for co-mp Using Axioms

Each column represents an instruction flowing through the pipeline.

(i1) Fetch
(i2) Fetch
(i3) Fetch
(i4) Fetch

(i1) Execute
(i2) Execute
(i3) Execute
(i4) Execute

(i1) Writeback
(i2) Writeback
(i3) Writeback
(i4) Writeback

(i1) MemHier
(i2) MemHier
(i3) MemHier
(i4) MemHier

Initially, Mem[x] = 0

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>i1: Store [x] ← 1</td>
<td>i3: r1 = Load [x]</td>
</tr>
<tr>
<td>i2: Store [x] ← 2</td>
<td>i4: r2 = Load [x]</td>
</tr>
</tbody>
</table>

SC Forbids: r1=2, r2=1, Mem[x] = 2
μhb Graphs for co-mp Using Axioms

Each column represents an instruction flowing through the pipeline.

(i1) (i2)

Fetch

(i3) (i4)

Execute

Thread 0

(i1): Store [x] ← 1
(i2): Store [x] ← 2
(i3): r1 = Load [x]
(i4): r2 = Load [x]

Edge added by Enforce_Write_Ordering axiom

Writeback

MemHier

Initially, Mem[x] = 0

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>i1: Store [x] ← 1</td>
<td>i3: r1 = Load [x]</td>
</tr>
<tr>
<td>i2: Store [x] ← 2</td>
<td>i4: r2 = Load [x]</td>
</tr>
</tbody>
</table>

SC Forbids: r1=2, r2=1, Mem[x] = 2
Finding Axioms

Core 0
- Fetch
- Execute
- Writeback_{i_1}

Core 1
- Fetch
- Execute
- Writeback

Y = 0
Finding Axioms

If a load reads the initial value of a memory location, it must execute before any write to that location reaches Mem.

\[ Y = 0 \]
Finding Axioms

If a load reads the initial value of a memory location, it must execute before any write to that location reaches Mem.

\[
i: \text{Load } y=0
\]

\[
Y = 0
\]
If a load reads the initial value of a memory location, it must execute before any write to that location reaches Mem.

w: Store y=1
i: Load y=0

Y = 1
The BeforeAllWrites Macro

DefineMacro "BeforeAllWrites":
   DataFromInitialStateAtPA i \\n   forall microop "w", ( (IsAnyWrite w \\ SamePhysicalAddress w i \\ ~SameMicroop i w) => AddEdge ((i, _______), (w, (0, _______________))))).
The BeforeAllWrites Macro

DefineMacro "BeforeAllWrites":
DataFromInitialStateAtPA i \/
forall microop "w", ( (IsAnyWrite w \/
SamePhysicalAddress w i \/
~SameMicroop i w) =>
AddEdge ((i, _______), (w, (0, _______________)))).

Macro: This is a µSpec fragment that can be instantiated as part of a larger axiom.
The BeforeAllWrites Macro

DefineMacro "BeforeAllWrites":

DataFromInitialStateAtPA i \/
forall microop "w", (IsAnyWrite w \/
SamePhysicalAddress w i \/
~SameMicroop i w) ->
AddEdge ((i, _______), (w, (0, _______________))).

Check that the load reads the data from the initial state of the litmus test
The BeforeAllWrites Macro

DefineMacro "BeforeAllWrites":
DataFromInitialStateAtPA i \ /
forall microop "w", ( (IsAnyWrite w \ /\ SamePhysicalAddress w i \
 /\ ~SameMicroop i w) =>
AddEdge ((i, _______), (w, (0, _______________))))).

If a load reads the initial value of a memory location, it
must execute before any write to that addr reaches Mem.
The BeforeAllWrites Macro

DefineMacro "BeforeAllWrites":
  DataFromInitialStateAtPA i \ /
  forall microop "w", (  
    (IsAnyWrite w \ /
      SamePhysicalAddress w i \\
      \ ~SameMicroop i w) =>  
    AddEdge ((i, Execute), (w, (0, MemoryHierarchy)))).

if a load reads the initial value of a memory location, it must execute before any write to that location reaches Mem.
Enforce that the load executes before all writes to its address in the test.
Finding Axioms

Memory Hierarchy

Core 0
- Fetch
- Execute
- Writeback \(i_1\)

Core 1
- Fetch
- Execute
- Writeback

Memory Hierarchy
Finding Axioms

A load must execute either before or after any write to its address reaches memory.
Finding Axioms

A load must execute either before or after any write to its address reaches memory.

w: Store y=val1

i: Load y=val2

Memory Hierarchy
Finding Axioms

A load must execute either before or after any write to its address reaches memory.

Core 1
Fetch
Execute
Writeback

w: Store y=val1

OR

Core 0
Execute
Writeback

i: Load y=val2

Memory Hierarchy
The Before Or After Every SameAddrWrite Macro

DefineMacro "Before_Or_After_Every_SameAddrWrite":
   forall microop "w", (  
      (IsAnyWrite w \ SamePhysicalAddress w i) =>  
      (AddEdge ((w, (0, MemoryHierarchy)), (i, Execute)) \  
         AddEdge ((i, Execute), (w, (0, MemoryHierarchy)))))

\[ Writeback_i,1 \] OR \[ Writeback \]
\[ w: Store y=val1 \] OR \[ i: Load y=val2 \]
Memory Hierarchy
The Before Or After Every SameAddrWrite Macro

DefineMacro "Before Or After Every SameAddrWrite":
forall microop "w", ( 
(IsAnyWrite w \ SamePhysicalAddress w i) => 
(AddEdge ((w, (0, MemoryHierarchy)), (i, Execute)) \/
AddEdge ((i, Execute), (w, (0, MemoryHierarchy))))).
Finding Axioms

Core 0
- Fetch
- Execute
- $\text{Writeback}_{w}$

Core 1
- Fetch
- Execute
- Writeback

$Y = 0$
Finding Axioms

A load must read from the latest write to that address to reach memory.

Y = 0
Finding Axioms

Alternatively:
1) The load must **execute after** the write it reads from
2) No writes to that address **between** the source write and the read

\[ Y = 0 \]
Finding Axioms

Alternatively:
1) The load must **execute after** the write it reads from
2) No writes to that address **between** the source write and the read

\[ w: \text{St } y=1 \]

\[ Y = 1 \]
Alternatively:
1) The load must execute after the write it reads from
2) No writes to that address between the source write and the read

w: St y=1

i: Load y=1

Y = 1
Finding Axioms

Alternatively:
1) The load must **execute after the write it reads from**
2) **No writes to that address between the source write and the read**

\[ w: \text{St } y=1 \]
\[ w': \text{St } y=2 \]
\[ i: \text{Load } y=1 \]
\[ Y = 2 \]
The No_SameAddrWrites_Btwn_Src_And_Read Macro

DefineMacro "No_SameAddrWrites_Btwn_Src_And_Read":
exists microop "w", (?
  IsAnyWrite w \[\]
  AddEdge ((w, (0, MemHier)), (i, Execute)) \[\]
~(exists microop "w'",
  IsAnyWrite w' \[\]
  ~SameMicroop w w' \[\]
  EdgesExist [((w, (0, MemHier)), (w', (0, MemHier)));
              ((w', (0, MemHier)), (i, Execute))])).

1) The load must execute after the write it reads from
2) No writes to that address between the source write and the read
DefineMacro "No_SameAddrWrites_Btwn_Src_And_Read":
exists microop "w", ( 
  IsAnyWrite w \ SamePhysicalAddress w i \ SameData w i \\ AddEdge ((w, (0, MemoryHierarchy)), (i, Execute)) \~(exists microop "w'", 
    IsAnyWrite w' \ SamePhysicalAddress i w' \~SameMicroop w w' \\ EdgesExist [((w, (0,MemHier)), (w', (0,MemHier))); ((w', (0,MemHier)), (i, Execute))]]).
DefineMacro "No_SameAddrWrites_Btwn_Src_And_Read": exists microop "w", (IsAnyWrite w \ SamePhysicalAddress w i \ SameData w i /\ AddEdge ((w, (0, MemoryHierarchy)), (i, Execute)) \~(exists microop "w'", IsAnyWrite w' \ SamePhysicalAddress i w’ /\ ~SameMicroop w w’ /\ EdgesExist [((w, (0,MemHier)), (w’, (0,MemHier)))); ((w’, (0,MemHier)), (i, Execute))])).

Read i executes after its source write w reaches memory…
DefineMacro "No_SameAddrWrites_Btwn_Src_And_Read":
exists microop "w", ( 
  IsAnyWrite w \ SamePhysicalAddress w i \ SameData w i \\
  \ AddEdge ((w, (0, MemoryHierarchy)), (i, Execute)) \ \\
  ~(exists microop "w'", 
  IsAnyWrite w' \ SamePhysicalAddress i w' \ \\
  ~SameMicroop w w' \\
  \ EdgesExist [((w, (0,MemHier)), (w', (0,MemHier))); 
  ((w', (0,MemHier)), (i, Execute))]]).
Putting the Macros together: the Read_Values axiom

Axiom "Read_Values":
forall microops "i",
IsAnyRead i =>
(ExpandMacro BeforeAllWrites \/
  (ExpandMacro No_SameAddrWrites_Btwn_Src_And_Read \/
   ExpandMacro Before_Or_After_Every_SameAddrWrite )).

\[ A \text{: Load } y = 1 \]
\[ w: St y = 1 \]
\[ w': St y = 2 \]
\[ i: Load y = 2 \]
\[ Y = \emptyset \]
Axiom "Read Values":

forall microops "i", IsAnyRead i =>

(ExpandMacro BeforeAllWrites /

( ExpandMacro No_SameAddrWrites_Btwn_Src_And_Read /

ExpandMacro Before_Or_After_Every_SameAddrWrite )).

For all reads i (same identifier used in the macros)…
Putting the Macros together: the Read_Values axiom

Axiom "Read_Values":
forall microops "i",
IsAnyRead i =>
(ExpandMacro BeforeAllWrites) /

(ExpandMacro No_SameAddrWrites_Btwn_Src_And_Read /
ExpandMacro Before_or_After_Every_SameAddrWrite)).

...either the read executes before all writes (expand macro defined earlier)...

\[ w \text{: St } y=1 \]
\[ w' \text{: St } y=2 \]
\[ l \text{: Load } y \]
Putting the Macros together: the Read\_Values axiom

Axiom "Read\_Values":
forall microops "i",
IsAnyRead i =>
(ExpandMacro BeforeAllWrites /
 (ExpandMacro No_SameAddrWrites_Btwn_Src_And_Read /
 ExpandMacro Before_Or_After_Every_SameAddrWrite ))).

...or the read reads from the latest write to that address
μhb Graphs for co-mp Using Axioms

Each column represents an instruction flowing through the pipeline

(i1) (i2)
Fetch

(i3) (i4)
Execute

Thread 0

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>i1: Store [x]</td>
<td>1</td>
</tr>
<tr>
<td>i2: Store [x]</td>
<td>2</td>
</tr>
</tbody>
</table>

Thread 1

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>i3: r1 = Load [x]</td>
<td>i4: r2 = Load [x]</td>
</tr>
</tbody>
</table>

Initially, Mem[x] = 0

SC Forbids: r1=2, r2=1, Mem[x] = 2
μhb Graphs for co-mp Using Axioms

Each column represents an instruction flowing through the pipeline

(i1) (i2) (i3) (i4)

Fetch
- i1: Store \( [x] \) ← 1
- i2: Store \( [x] \) ← 2

Execute
- i3: \( r1 = \text{Load} [x] \)
- i4: \( r2 = \text{Load} [x] \)

Writeback
- SC Forbids: \( r1=2, r2=1, \text{Mem}[x] = 2 \)

MemHier

Initially, \( \text{Mem}[x] = 0 \)

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>i1: Store ([x]) ← 1</td>
<td>i3: ( r1 = \text{Load} [x] )</td>
</tr>
<tr>
<td>i2: Store ([x]) ← 2</td>
<td>i4: ( r2 = \text{Load} [x] )</td>
</tr>
</tbody>
</table>

Initially, \( \text{Mem}[x] = 0 \)
μhb Graphs for co-mp Using Axioms

Each column represents an instruction flowing through the pipeline

(i1) (i2)

(i3) (i4)

Fetch

Execute

Writeback

MemHier

Thread 0

(i1): Store [x] ← 1
(i2): Store [x] ← 2

Thread 1

(i3): r1 = Load [x]
(i4): r2 = Load [x]

Initially, Mem[x] = 0

• i4 must be sourced from i1
• But i2 intervenes!
=> Constraint unsatisfiable

SC Forbids: r1=2, r2=1, Mem[x] = 2
Cannot find an **acyclic** graph that **satisfies** all constraints => 
**Forbidden** Execution of co-mp is **NOT observable** on µarch!
Test your completed SC uarch!

# Assuming you are in ~/pipecheck_tutorial/uarches/
$ check -i ../tests/SC_tests/co-mp.test -m SC_fillable.uarch

# If your uarch is valid, the above will create co-mp.pdf in your
# current directory (open pdfs from command line with evince)
# To run the solution version of the SC uarch on this test:
# (Note: this will overwrite the co-mp.pdf in your current folder)
$ check -i ../tests/SC_tests/co-mp.test -m SC.uarch -d solutions/

# If you get an error (cannot parse uarch, ps2pdf crashes, etc),
# examine your syntax or ask for help.
# If the outcome is observable (“BUG”), compare the graphs
# generated by the solution uarch to those of your uarch.

# To compare the uarches themselves:
$ diff SC_fillable.uarch solutions/SC.uarch
Run the entire suite of SC litmus tests!

# Assuming you are in ~/pipecheck_tutorial/uarches/
$ run_tests -v 2 -t ../tests/SC_tests/ -m SC_fillable.uarch

# The above will generate *.gv files in ~/pipecheck_tutorial/out/
# for all SC tests, and output overall statistics at the end. If
# the count for “Buggy” is non-zero, your uarch is faulty. Look for
# the tests that output “BUG” to find out which tests fail.

# You can use gen_graph to convert gv files into PDFs:
$ gen_graph -i <test_gv_file>

# Compare your uarch with the solution SC uarch using diff to find
# discrepancies:
$ diff SC_fillable.uarch solutions/SC.uarch
Coffee Break!

After the break: Extending SC uarch. to TSO