Hands-on: Moving from SC to TSO

- Reads must currently wait for prior writes to reach memory
  - EnforceWriteOrdering axiom

- Main motivation for TSO: store buffers to hide write latency

- Also want to allow reads to bypass value from store buffer (before value made visible to other cores)
  - Known as “read your own write early”

- How to model this in µSpec?
Moving from SC to TSO

Core 0:
- Fetch
- Execute
- Writeback

Core 1:
- Fetch
- Execute
- Writeback

Memory Hierarchy
Moving from SC to TSO
Moving from SC to TSO

Core 0
- Fetch
- Execute
- Writeback

Core 1
- Fetch
- Execute
- Writeback

Loads can bypass from SB

Memory Hierarchy
Moving from SC to TSO

Partially completed TSO uarch in
/home/check/pipecheck_tutorial/uarchs/TSO_fillable.uarch

Some axioms remain the same from SC.uarch
Hands-on: Moving from SC to TSO

- 7 changes needed to SC.uarch:
  1. Add store buffer stage
  2. Make writes go through SB before memory
  3. Ensure that same-core writes go through SB in order
  4. Enforce that write is released from SB only after all prior same-core writes have reached memory
  5. Ensure that if load is reading from memory, that core’s store buffer has no entries for address of load
  6. (Advanced) Allow a core to read value of a write from its store buffer before write is made visible to other cores
  7. (Advanced) Implement fence operation that flushes all prior writes to memory before any succeeding instructions can perform
Add StoreBuffer Stage

- “StoreBuffer” stage is between Writeback and MemoryHierarchy
- Solution:

```
StageName _ "__________".
StageName _ "MemoryHierarchy".
```
Add StoreBuffer Stage

- “StoreBuffer” stage is between Writeback and MemoryHierarchy
- Solution:

  StageName 3 "StoreBuffer".
  StageName 4 "MemoryHierarchy".
Hands-on: Moving from SC to TSO

- 7 changes needed to SC.uarch:
  1. Add store buffer stage
  2. **Make writes go through SB before memory**
  3. Ensure that same-core writes go through SB in order
  4. Enforce that write is released from SB only after all prior same-core writes have reached memory
  5. Ensure that if load is reading from memory, that core’s store buffer has no entries for address of load
  6. (Advanced) Allow a core to read value of a write from its store buffer before write is made visible to other cores
  7. (Advanced) Implement fence operation that flushes all prior writes to memory before any succeeding instructions can perform
Make Writes Go Through SB

- Modify Writes_Path axiom so stores go WB → SB → MemHier

Solution:

Axiom "Writes_Path":
forall microops "i",
IsAnyWrite i =>
AddEdges [((i, Fetch), (i, Execute), "path"));
 ((i, Execute), (i, Writeback), "path"));
 ((i, _________), (i, ___________), "path"));
 ((i, ___________), (i, (0, ____________)), "path")
].
Make Writes Go Through SB

- Modify Writes_Path axiom so stores go WB → SB → MemHier
- Solution:

Axiom "Writes_Path":
forall microops "i",
IsAnyWrite i =>
AddEdges [((i, Fetch), (i, Execute),  "path"));
   ((i, Execute), (i, Writeback),  "path"));
   ((i, Writeback), (i, StoreBuffer),  "path"));
   ((i, StoreBuffer), (i, (0, MemoryHierarchy)),
    "path")
].
Hands-on: Moving from SC to TSO

- 7 changes needed to SC.uarch:
  1. Add store buffer stage
  2. Make writes go through SB before memory
  3. **Ensure that same-core writes go through SB in order**
  4. Enforce that write is released from SB only after all prior same-core writes have reached memory
  5. Ensure that if load is reading from memory, that core’s store buffer has no entries for address of load
  6. (Advanced) Allow a core to read value of a write from its store buffer before write is made visible to other cores
  7. (Advanced) Implement fence operation that flushes all prior writes to memory before any succeeding instructions can perform
Same-Core Writes Go Through SB in order

- If same-core writes go through WB in order, they should go through SB in order too.
- Hint: Use Writeback_stage_is_in_order axiom as a starting point.
- Solution:

Axiom "StoreBuffer_stage_is_in_order":
for all microops "i1",
for all microops "i2",
IsAnyWrite i1 \ IsAnyWrite i2 \ _______ i1 i2 =>
EdgeExists ((i1, _______), (i2, _______), "") =>
AddEdge ((i1, ___________), (i2, ___________), "PPO", "darkgreen").
Same-Core Writes Go Through SB in order

- If same-core writes go through WB in order, they should go through SB in order too
- Hint: Use Writeback_stage_is_in_order axiom as a starting point
- Solution:

Axiom "StoreBuffer_stage_is_in_order":
forall microops "i1",
forall microops "i2",
IsAnyWrite i1 /\ IsAnyWrite i2 /\ SameCore i1 i2 =>
EdgeExists ((i1, Writeback), (i2, Writeback), "") =>
AddEdge ((i1, StoreBuffer), (i2, StoreBuffer), "PPO", "darkgreen").
Hands-on: Moving from SC to TSO

- 7 changes needed to SC.uarch:
  1. Add store buffer stage
  2. Make writes go through SB before memory
  3. Ensure that same-core writes go through SB in order
  4. **Enforce that write is released from SB only after all prior same-core writes have reached memory**
  5. Ensure that if load is reading from memory, that core’s store buffer has no entries for address of load
  6. (Advanced) Allow a core to read value of a write from its store buffer before write is made visible to other cores
  7. (Advanced) Implement fence operation that flushes all prior writes to memory before any succeeding instructions can perform
Same-Core Writes Reach Memory In Order

- For two same-core writes in program order, first write must reach memory before second can leave store buffer

- Hint: Axiom should only apply to pairs of writes!

- Solution:

Axiom "EnforceWriteOrdering":
forall microop "w",
forall microop "w'",
(IsAnyWrite w /\ __________ w' /\ __________ w w') =>
AddEdge ((w, (0, _____________)), (w', _____________),
"one_at_a_time", "green").
Same-Core Writes Reach Memory In Order

- For two same-core writes in program order, first write must reach memory before second can leave store buffer
- Hint: Axiom should only apply to pairs of writes!
- Solution:

Axiom "EnforceWriteOrdering":
forall microop "w",
forall microop "w'",
(IsAnyWrite w \ IsAnyWrite w' \ ProgramOrder w w') =>
AddEdge ((w, (0, MemoryHierarchy)), (w', StoreBuffer), "one_at_a_time", "green").
Hands-on: Moving from SC to TSO

- 7 changes needed to SC.uarch:
  1. Add store buffer stage
  2. Make writes go through SB before memory
  3. Ensure that same-core writes go through SB in order
  4. Enforce that write is released from SB only after all prior same-core
     writes have reached memory
  5. Ensure that if load is reading from memory, that core’s store buffer has
     no entries for address of load
  6. (Advanced) Allow a core to read value of a write from its store buffer
     before write is made visible to other cores
  7. (Advanced) Implement fence operation that flushes all prior writes to
     memory before any succeeding instructions can perform
Only read from Mem if SB has no same addr writes

- Create a **macro** enforcing that all writes before instr “i” in program order to address of “i” have reached mem before “i” **Executes**

- Solution:
Only read from Mem if SB has no same addr writes

- Create a **macro** enforcing that all writes before instr “i” in program order to address of “i” have reached mem before “i” **Executes**

- Solution:

```defineMacro "STBEmpty":
  % Store buffer is empty for the address we want to read.
  forall microop "w", (  
    (_________ w \__ \________________________ w i \/ 
    ___________ w i) =>  
    AddEdge ((w, (0, _______________)), (i, ______),  
    "STBEmpty", "purple")).```
Only read from Mem if SB has no same addr writes

- Create a **macro** enforcing that all writes before instr “i” in program order to address of “i” have reached mem before “i” **Executes**

- **Solution:**

```
DefineMacro "STBEmpty":
  % Store buffer is empty for the address we want to read.
  forall microop "w", ( (IsAnyWrite w \ SamePhysicalAddress w i \ ProgramOrder w i) => AddEdge ((w, (0, MemoryHierarchy)), (i, Execute), "STBEmpty", "purple")).
```
Only read from Mem if SB has no same addr writes

- Now expand the macro in Read_Values axiom to ensure that SB has no entries for a load’s address if it is reading from memory
Axiom "Read_Values":
forall microops "i",
IsAnyRead i => (  
% Uncomment the commented lines if you add the (advanced) store buff forwarding.
% ExpandMacro ______ \/
% (   
  ExpandMacro ______ \/
   
  (   
    ExpandMacro BeforeAllWrites  
    \/
    (   
      ExpandMacro No_SameAddrWrites_Btwn_Src_And_Read  
      \/
      ExpandMacro Before_Or_After_Every_SameAddrWrite  
    )
  )   
)  
% )  
).
Only read from Mem if SB has no same addr writes

Axiom "Read_Values":
forall microops "i",
IsAnyRead i => (  
% Uncomment the commented lines if you add the (advanced) store buff forwarding.  
% ExpandMacro ______ \/
% (         
    ExpandMacro STBEmpty \/
      (     
        ExpandMacro BeforeAllWrites \
        (         
          ExpandMacro No_SameAddrWrites_Btwn_Src_And_Read \
          (         
            ExpandMacro Before_Or_After_Every_SameAddrWrite
          )
        )
      )
    )
) 
).
Hands-on: Moving from SC to TSO

- 7 changes needed to SC.uarch:
  1. Add store buffer stage
  2. Make writes go through SB before memory
  3. Ensure that same-core writes go through SB in order
  4. Enforce that write is released from SB only after all prior same-core writes have reached memory
  5. Ensure that if load is reading from memory, that core’s store buffer has no entries for address of load
  6. (Advanced) Allow a core to read value of a write from its store buffer before write is made visible to other cores
  7. (Advanced) Implement fence operation that flushes all prior writes to memory before any succeeding instructions can perform
Forward Value from SB (Advanced)

- Create a **macro** that checks for a write on the same core to forward from (Execute stage -> Execute stage), and ensures the forwarding occurs **before** the write reaches memory

- Macro must also check that forwarding occurs from the latest write in program order (no intervening writes)

- Solution:
DefineMacro "STBFwd":

% Forward from the store buffer
exists microop "w", (  
  __________ w /
  _______ w i /
  ______________ w i /
  _______ w i /
  AddEdges [((w, Execute), (i, Execute), "STBFwd", "red"),
    ((i, Execute), (w, (0, MemoryHierarchy)), "STBFwd", "purple")]
  /
% Ensure the STB entry is the latest one.
~exists microop "w'",
  __________ w' /
  ______________ w w' /
  __________ w w' /
  __________ w' i.
DefineMacro "STBFwd":
  % Forward from the store buffer
  exists microop "w", (  
    IsAnyWrite w \\ 
    SameCore w i \\ 
    SamePhysicalAddress w i \\ 
    SameData w i \\ 
    AddEdges [((w, Execute), (i, Execute), "STBFwd", "red");  
      ((i, Execute), (w, (0, MemoryHierarchy)), "STBFwd", 
        "purple")]) \\ 
  % Ensure the STB entry is the latest one.
  ~exists microop "w'",
  IsAnyWrite w' \\ SamePhysicalAddress w w' \\ 
  ProgramOrder w w' \\ ProgramOrder w' i.
Forward Value from SB (Advanced)

- Expand the macro in the Read_Values axiom so that forwarding from the SB is an alternative choice to reading from memory

- Solution:
Axiom "Read_Values":
forall microops "i",
IsAnyRead i =>
(
    ExpandMacro ______ \/
    (ExpandMacro STBEmpty \/
    (ExpandMacro BeforeAllWrites
    \/
    (ExpandMacro No_SameAddrWrites_Btwn_Src_And_Read
    \/
    ExpandMacro Before_Or_After_Every_SameAddrWrite
    ))
    )
).
Axiom "Read_Values":
forall microops "i",
IsAnyRead i =>
  (ExpandMacro STBFwd /
   (ExpandMacro STBEmpty /
     (ExpandMacro BeforeAllWrites /
      (ExpandMacro No_SameAddrWrites_Btwn_Src_And_Read /
       ExpandMacro Before_Or_After_Every_SameAddrWrite)))).
Hands-on: Moving from SC to TSO

- 7 changes needed to SC.uarch:
  1. Add store buffer stage
  2. Make writes go through SB before memory
  3. Ensure that same-core writes go through SB in order
  4. Enforce that write is released from SB only after all prior same-core writes have reached memory
  5. Ensure that if load is reading from memory, that core’s store buffer has no entries for address of load
  6. (Advanced) Allow a core to read value of a write from its store buffer before write is made visible to other cores
  7. (Advanced) Implement fence operation that flushes all prior writes to memory before any succeeding instructions can perform
Fence Instruction Orders Write-Read pairs

▪ Add a fence instruction that flushes all prior writes in program order to memory before the fence's execute stage

▪ Solution:
Fence Instruction Orders Write-Read pairs

- Add a fence instruction that flushes all prior writes in program order to memory before the fence's execute stage.

Solution:

Axiom "Fence_Ordering":
forall microops "f",
IsAnyFence f =>
AddEdges [((f, Fetch), (f, Execute), "path"));
((f, Execute), (f, Writeback), "path")]
/
(
forall microops "w",
(__________ w /\
__________ w f) =>
AddEdge ((w, (0, ______________)), (f, _______),
"fence", "orange") ).
Fence Instruction Orders Write-Read pairs

- Add a fence instruction that flushes all prior writes in program order to memory before the fence’s execute stage.

Axiom "Fence_Ordering":
forall microops "f",
IsAnyFence f =>
AddEdges [((f, Fetch),      (f, Execute),      "path"));
   ((f, Execute),     (f, Writeback), "path")]
/
(forall microops "w",
   (IsAnyWrite w /\ ProgramOrder w f) =>
    AddEdge ((w, (0, MemoryHierarchy)), (f, Execute),
       "fence", "orange")
).
Initially, $\text{Mem}[x] = \text{Mem}[y] = 0$

### µhb Graph for sb On TSO µarch.

- **Thread 0**
  - i1: Store $[x] \leftarrow 1$
  - i2: $r1 = \text{Load}[y]$

- **Thread 1**
  - i3: Store $[y] \leftarrow 1$
  - i4: $r2 = \text{Load}[x]$

**SC Forbids:** $r1=0$, $r2=0$

### Memory Hierarchy (MemHier)

- **StoreBuffer**
  - Writeback
  - Execute
  - Fetch

The graph illustrates the flow of data and dependencies between different stages of memory operations.
Initially, Mem[x] = Mem[y] = 0

\[ \mu hb \text{ Graph for } sb \text{ On TSO } \mu arch. \]

Thread 0
- i1: Store [x] $\leftarrow$ 1
- i2: r1 = Load [y]

Thread 1
- i3: Store [y] $\leftarrow$ 1
- i4: r2 = Load [x]

SC Forbids: r1=0, r2=0
Initially, Mem\[x\] = Mem\[y\] = 0

\[\mu hb\] Graph for sb On TSO \(\mu\)arch.

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

SC Forbids: \(r1=0, r2=0\)

Loads no longer need to wait for prior writes to reach memory => **acyclic graph**

**sb** is **observable** on TSO \(\mu\)arch!
Test your completed TSO uarch!

```
# Assuming you are in ~/pipecheck_tutorial/uarches/
$ check -i ../tests/TSO_tests/sb.test -m TSO_fillable.uarch

# If your uarch is valid, the above will create sb.pdf in your current directory (open pdfs from command line with evince)
# To run the solution version of the TSO uarch on this test:
# (Note: this will overwrite the sb.pdf in your current folder)
$ check -i ../tests/TSO_tests/sb.test -m TSO.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 not observable (“Strict”), compare the graphs # generated by the solution uarch to those of your uarch.

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

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

# The above will generate *.gv files in ~/pipecheck_tutorial/out/
# for all TSO 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 TSO uarch using diff to find
# discrepancies:
$ diff TSO_fillable.uarch solutions/TSO.uarch
PipeCheck Verification Time

![Graph showing runtime comparison between different benchmarks]

- **FiveStage (No SB)**
- **FiveStage (w/ SB)**
- **gem5 O3**
- **OpenSPARC T2**

**Runtime (s)**

- FiveStage (No SB)
- FiveStage (w/ SB)
- gem5 O3
- OpenSPARC T2
Covered the basics of what PipeCheck can do…

- But there’s more!

- PipeCheck can handle heterogeneous pipelines:
Covered the basics of what PipeCheck can do...

- ...and microarchitectural optimizations...

**Left:** Speculative Load Reordering

**Right:** Speculative Fence Retirement
Covered the basics of what PipeCheck can do…

- ...and the methodology is extensible to other types of orderings!

**CCICheck:** Coherence orderings that affect consistency

**COATCheck:** Addr Translation/Virtual Memory orderings that affect consistency
PipeCheck Summary

- Fast, automated verification
- Check implementation against ISA spec
- Decompose verification into smaller per-axiom sub-problems
- Open-Sourced:
  
  https://github.com/daniellustig/coatcheck

Repo from this tutorial:

https://github.com/ymanerka/pipecheck_tutorial