Multicore Computer Architecture Issues:
From hardware to software
18-613, Spring 2019
Technology Improves

Transistors get smaller, clock speeds go up, power stays roughly constant. Party!
Performance Scaling Hit a Wall!

Transistors still doubling, performance tapers off. Architects need to be creative!
• Willamette core
• 180 nm process
• 217 mm² die size
• 42,000,000 transistors
• **14 nm process**
• 11 metal layers
• ~1,750,000,000 transistors
• ~9.19 mm x ~11.08 mm
• ~101.83 mm² die size
• 4 CPU cores + 24 GPU EUs
• 14 nm process
• 11 metal layers
• ~1,750,000,000 transistors
• ~9.19 mm x ~11.08 mm
• ~101.83 mm² die size
• 4 CPU cores + 24 GPU EUs

Shared memory multi-threading
• 14 nm process
• 682.6 mm² die size
• 76 CPU cores
• 7,100,000,000 transistors
Amdahl’s Law

\[ \frac{1}{(1-p) + \frac{p}{s}} \]
Amdahl’s Corollary: Speedup is limited by fraction of the program that is parallelizable.
- 14 nm process
- 11 metal layers
- ~1,750,000,000 transistors
- ~9.19 mm x ~11.08 mm
- ~101.83 mm² die size
- 4 CPU cores + 24 GPU EUs
“Coherence seeks to make the caches of a shared-memory system as functionally invisible as the caches in a single-core system. Correct coherence ensures that a programmer cannot determine whether and where a system has caches by analyzing the results of loads and stores.”

Excerpt from “Primer on Memory Consistency and Cache Coherence”
Mark Hill, 2011
Cache Coherence
What is the behavior of this parallel program? (X initially 0)
What about this example?
(X initially 0)
(and the symmetric case)
What assumptions are we making about the system to produce the results 0, 1, and 2?
We assume the updates see one anothers’ results!
(Why wouldn’t they?)
So what the heck do we do now?
Never let this happen. Caches should be coherent.

“coherence ensures that a programmer cannot determine whether and where a system has caches by analyzing the results of loads and stores”
Informally Defining Coherence

“Coherence serializes all reads with all updates to the same location by different CPUs/caches, so that each read sees the result of the most recent update by any other”

“Single Writer/Multiple Reader (SWMR) Invariant + Data-Value Invariant”
Epoch Model

Read/Write Epoch for CPU1

[X]=0
X++
[X]=1

Read/Write Epoch for CPU2

[X]=1
X++
[X]=2

Read-only Epoch for all

[X]=2
Rd X=?
[X]=2
Yay! Corresponds to reality!

[X]=2
Rd X=?
Epoch Model

R/W vs. R-O Epochs directly enforce SWMR

Read/Write Epoch for CPU1

$[X]=0$

$X++$

$[X]=1$

Read/Write Epoch for CPU2

$[X]=1$

$X++$

$[X]=2$

Read-only Epoch for all

$[X]=2$

Yay! Corresponds to reality!

Epoch transitions assume data-value invariant
What do we need to implement the Epoch Model?

Need to add concept of R/W epoch vs. R-O epoch

Need to add gadget that correctly moves data between epochs
Cache Coherence Protocol

Add state to each cache line saying whether it is R-O or R/W

Add protocol actions to move lines from state to state based on (1) local memory operations; and (2) other CPUs’ memory operations

Add support to get data from (1) local cache; (2) a remote cache; or (3) main memory, depending on line’s protocol state
High-level sketch of protocol in action

(1) CPU1 says “I am is writing X”
    Others relinquish cached copies of X
    and reply “OK go for it”  
    <enter R/W epoch>

(2) (ditto (1) for CPU2)

(3) CPU1 replies “I have X. Use my copy or get it from memory after I write it back”

(4) CPU3 says “I want to read only”
    Others reply “OK, we all agree not to write without saying so”
    <enter R-O epoch>

(5) (ditto (3) for CPU 2)
Cache Coherence Protocol

Per-line coherence states

M

I

S
Cache Coherence Protocol

- Modified (R/W)
- Shared (R-O)
- Invalid (inaccessible)
Cache Coherence Protocol

Local operations perspective

Locally perform a read or write

Locally perform a write [send invalidations to other CPUs]

Locally perform a read [send requests to share to other CPUs]

Locally perform a write [send invalidations to other CPUs]

M

S

I
Cache Coherence Protocol

Remote operations perspective

M → S: Incoming request to share [reply with data or write back]

S → M: Incoming Invalidation [reply with invalidation acknowledgement]

I → M: Incoming Invalidation [reply with invalidation acknowledgement]

I → S: Incoming Invalidation [reply with invalidation acknowledgement]
Can we design another state?

What should we optimize?
Can we design another state?

(Benefit: no invalidation required to transition from E->M, like from S->M)
Implementing the Protocol

Snoopy Coherence

Shared bus for coherence messages
Implementing the Protocol

Invalidate

$X++$

CPU 1

$X++$

CPU 2

$X++$

CPU 3

Rd $X=?$

$
Implementing the Protocol

CPU 1

X++

$\$

CPU 2

X++

$\$

CPU 3

Rd X=?

$\$

Ack

Ack

X++
Implementing the Protocol

(M)

CPU 1

X++

X = 1

Entering CPU1’s write epoch

CPU 2

X++

$ $

CPU 3

Rd X = ?

$ $

X++
Implementing the Protocol

CPU 1

(M)

X=1

X++

CPU 2

X++

$\$

CPU 3

Rd Req

Rd X=?

X++

$\$

Rd X=?
Implementing the Protocol

CPU 1

X++

X=1

CPU 2

X++

$\$

CPU 3

Rd X=?

(M)

Don’t have it

Got it: X=1

X++

Rd X=?
Implementing the Protocol

CPU 1
X++
X=1
(S)

CPU 2
X++
$(S)$

CPU 3
Rd X=?
X=1
(S)

Entering R-O epoch

X++

Rd X=?
Implementing the Protocol

What sucks about Snoopy?
Implementing the Protocol

Bus limits scalability due to congestion and complex message arbitration
Sandybridge: bi-directional rings

Knight’s Landing 2017 2D mesh

Skylake Xeon 2017 2D mesh
Implementing the Protocol

Directory-based

Sharers of X
Implementing the Protocol

Directory-based Sharers of X

CPU 1

X++

$\$

CPU 2

X++

$\$

CPU 3

Rd X=?

$\$

Sharers of X
Implementing the Protocol

CPU 1  
X++  
$  

CPU 2  
X++  
$  

CPU 3  
Rd X=?  
$

Who has X?

Sharers of X

Directory-based
Implementing the Protocol

Directory-based

CPU 1
X++
$

CPU 2
X++
$

CPU 3
Rd X=?
$

No one does! Proceed!

Sharers of X

X++
Implementing the Protocol

Directory-based

CPU 1

X++

$ $

CPU 2

X++

X=1

CPU 3

Rd X=?

X=1

CPUs 2 and 3 do. Send them Invalidates!

Sharers of X

X++

Rd X=?
Implementing the Protocol

Benefit: No broadcast on shared bus
Implementing the Protocol

Drawbacks?
Implementing the Protocol

Centralized directory won’t scale
(In Practice: Distribute Directory)
Optimization: Non-binding Prefetch

Prefetch instruction preemptively changes coherence state

Sharers of X

CPU 2

CPU 3
Optimization: Non-binding Prefetch

Benefit?
Optimization: Speculation

Speculative operations that squash behave like non-binding pre-fetch
“computers execute operations in a **different order** than is specified by the program. A correct execution is achieved if the results produced are the same as would be produced by executing the program steps in order. For a multiprocessor computer, such a correct execution by each processor **does not guarantee the correct execution of the entire program.**”

Excerpt from “How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Program”
LESLE LAMPORT, 1979
Memory Consistency
Memory Consistency Model

Informal Definition:

“Defines the value a read operation may read at each point during the execution”

“Defines the set of legal observable orders of memory operations during an execution”

“Defines which reorderings of memory operations are permitted”
Review: Coherence

2 Invariants:

1) “One Writer or One or More Readers”

2) “Reading X gets the value of the last write to X”
Review: Coherence

2 Invariants:

1) “One Writer or One or More Readers”

2) “Reading X gets the value of the last write to X”
Without Coherence

(The coherence invariants prevent this from happening)

Without coherence, processors can't decide who wrote last. Green is hosed.
Coherence defines the set of legal orders of accesses to a single memory location.
Consistency is Ordering

Consistency defines the set of legal orders of accesses to multiple memory locations
Sequential Consistency (SC)

The simplest, most intuitive memory consistency model

Two Invariants to SC:

Instructions are executed in program order

All processors agree on a total order of executed instructions
The SC “Switch”

Execution

- Write X
- Read Y

- Write Y
- Read X

- Read X
The SC “Switch”

Execution

Wr X

Rd Y

Wr Y

Rd X

Rd X

Wr X

Execute
The SC “Switch”

Execution
Wr X
Rd Y

Execute

Wr X
Rd Y

Wr Y
Rd X

Rd X
The SC “Switch”

- Wr X
- Rd Y

- Wr Y
- Rd X

- Rd X

- Execution
  - Wr X
  - Rd Y
  - Wr Y
The SC “Switch”

**Execution**
- Wr X
- Rd Y
- Wr Y
- Rd X

Execute

- Wr X
- Rd Y
- Wr Y
- Rd X
The SC “Switch”

Execution
- Wr X
- Rd Y
- Wr Y
- Rd X
- Rd X
Why is SC Important?

Who cares?.... You care!

SC is the most complex model that we can ask programmers to think about.

SC prohibits all reordering of instructions (Invariant 1)
Why are Instructions Reordered?

And when does it matter?
Why are Instructions Reordered?

Optimization. Elsewhere?
Reordering #1: Write Buffers

CPU can read its write buffer, but not others’

Buffered writes eventually end up in coherent shared memory
Reordering #1: Write Buffers

Program
Initially $X == Y == 0$

$X=1$ \hspace{1cm} $Y=1$

$r1=Y$ \hspace{1cm} $r2=X$

Is $r1==r2==0$ a valid result?
Reordering #1: Write Buffers

Program
Initially $X == Y == 0$

$X=1$  $Y=1$

$r1=Y$  $r2=X$

Is $r1==r2==0$ a valid result?

$r1 == r2 == 0$ is not SC, but it can happen with write buffers
Reordering #1: Write Buffers

Program
Initially X == Y == 0

Y=1
r1=Y
r2=X

Execution
Reordering #1: Write Buffers

Program
Initially X == Y == 0

r1=Y  r2=X

Execution
Reordering #1: Write Buffers

Program
Initially X == Y == 0

Execution

$\text{r1}=Y \quad \text{r2}=X$
Reordering #1: Write Buffers

Program
Initially X == Y == 0

Execution
Reordering #1: Write Buffers

Initial variables:
- Initially X == Y == 0

Operations:
- X = 1
- Y = 1
- r1 = Y
- r2 = X

Program:

Execution:
Reordering #1: Write Buffers

**Program**
Initially X == Y == 0

**Execution**
r1=Y [r1 <- 0]
Reordering #1: Write Buffers

Program
Initially $X = Y = 0$

Execution
$r1 = Y$ [r1 <- 0]
$r2 = X$ [r2 <- 0]
Reordering #1: Write Buffers

Program
Initially X == Y == 0

Execution
r1 = Y [r1 <- 0]
r2 = X [r2 <- 0]
X = 1
Y = 1
(Not SC!)

WBs let reads finish before older writes
Reordering #2: Write Combining

Coalescing Write Buffer

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4 word cache line

Program

X,Z in same $ line

X=1
Y=1
Z=1
Reordering #2: Write Combining

Coalescing Write Buffer

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>X=1</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Program

X,Z in same $ line

X=1
Y=1
Z=1
Reordering #2: Write Combining

<table>
<thead>
<tr>
<th>Coalescing Write Buffer</th>
<th>Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>X=1</td>
<td>X=1</td>
</tr>
<tr>
<td></td>
<td>Y=1</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

X,Z in same $ line

Y=1

Z=1
Reordering #2: Write Combining

Coalescing Write Buffer

<table>
<thead>
<tr>
<th>X=1</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Y=1</td>
</tr>
<tr>
<td></td>
<td>Z=1</td>
<td></td>
</tr>
</tbody>
</table>

Program

X,Z in same $ line

| X=1 | Y=1 | Z=1 |
Reordering #2: Write Combining

Combining the write to X & Z saves bandwidth, but reorders Z=1 and Y=1
Reordering #3: Interconnect

Variable time cost traversing routed on-chip network

Program

<table>
<thead>
<tr>
<th>X=1</th>
<th>r1=X</th>
<th>Y=1</th>
<th>r3=Y</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>r2=Y</td>
<td></td>
<td>r4=X</td>
</tr>
</tbody>
</table>

Execution

<table>
<thead>
<tr>
<th>X=1</th>
<th>Y=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1=X [r1 &lt;- 1]</td>
<td></td>
</tr>
<tr>
<td>r2=Y [r2 &lt;- 0]</td>
<td></td>
</tr>
<tr>
<td>r3=Y [r3 &lt;- 1]</td>
<td></td>
</tr>
<tr>
<td>r4=X [r4 &lt;- 0]</td>
<td></td>
</tr>
</tbody>
</table>
Reordering #4: Compilers

The compiler hoists the write out of the loop, permitting new (non-SC) results (e.g., “1 0 0 0 0 0 0...”)
When is Reordering a Problem?

When Executions Aren’t SC
When is an Execution Not SC?

When a memory operation happens before itself

<table>
<thead>
<tr>
<th>Execution</th>
<th>Happens-Before Graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r_1 = Y$ [r1 &lt;- 0]</td>
<td>$X=1$</td>
</tr>
<tr>
<td>$r_2 = X$ [r2 &lt;- 0]</td>
<td>$Y=1$</td>
</tr>
<tr>
<td>$X=1$</td>
<td></td>
</tr>
<tr>
<td>$Y=1$</td>
<td></td>
</tr>
<tr>
<td>$r_1 = Y$</td>
<td>$r_2 = X$</td>
</tr>
</tbody>
</table>
When is an Execution Not SC?

When a memory operation happens before itself

Execution

\[ r1=Y \quad [r1 \leftarrow 0] \]
\[ r2=X \quad [r2 \leftarrow 0] \]
\[ X=1 \]
\[ Y=1 \]

Happens-Before Graph

\[ X=1 \]
\[ r2=X \]
\[ Y=1 \]
\[ r1=Y \]

Program Order HB Edge
When is an Execution Not SC?

When a memory operation happens before itself

Execution

r1=Y [r1 <- 0]
r2=X [r2 <- 0]
X=1
Y=1

Happens-Before Graph

X=1
r1=Y

Y=1
r2=X

↓ Program Order HB Edge
↓ Causal Order HB Edge
When is an Execution Not SC?

When a memory operation happens before itself

If there is a cycle in the happens-before graph, the execution is not SC
When is an Execution Not SC?

When a memory operation happens before itself

If there is a cycle in the happens-before graph, the execution is not SC
So... are Computers Wrong?! 

SC is how programmers think.

SC prohibits all reordering of instructions

WBs let reads finish before older writes

Combining writes saves bandwidth but reorders writes
Relaxed Memory Consistency

Relaxed Memory Models permit reorderings, unlike SC
x86-TSO (intel x86s)

“The Write Buffer Memory Model”

Relaxes W->R order

**Total Store Order** - loads may complete before older stores to different locations complete.
PSO_{(SPARC)}

“The Write Combining Memory Model”

Partial Store Order - loads and stores may complete before older stores to different locations complete.

Relaxes W->W order
In General

Starting with PSO and relaxing $R \rightarrow R$ and $R \rightarrow W$ yields Weak Ordering or Release Consistency (alpha) depending on the implementation.
SC and Relaxed Consistency

SC is required for correctness and programmer sanity

+ 

Reordering is required* for performance

Goal: Ensure SC executions while permitting

Relaxed Consistency reorderings

*Usually; the MIPS memory model is SC (surprising!)
How to ensure SC, but permit reordering?
Synchronization Prevents Reordering

Memory fences are another type of synchronization.

Reordering prevented

Fence implementation depends on reordering implementation.

TSO: Stall reads until write buffer is empty.
Synchronization For Real Programmers

Memory fences are wrapped up in locks, etc.

Direct use of fences possible, but inadvisable.

USE A LIBRARY.
Data Races

Synchronization imposes happens-before on otherwise unordered operations

Data Race: Unordered operations to the same memory location, at least one a write
## Memory Models across the System Stack

<table>
<thead>
<tr>
<th>Language</th>
<th>Compiler</th>
<th>Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java/C++: SC</td>
<td>Conservative with reordering when d-r-f can’t be proved</td>
<td>Usually very weak for max optimization (lots of reordering)</td>
</tr>
<tr>
<td>for data-race-free programs</td>
<td></td>
<td>Note: fences from “above” ensure SC</td>
</tr>
</tbody>
</table>