Shared Memory and Memory Consistency Models

Daniel J. Sorin

Duke University

UPMARC 2016

Who Am I?

- Professor of ECE and Computer Science
- From Duke University
  - In Durham, North Carolina
- My research and teaching interests
  - Cache coherence protocols and memory consistency
  - Fault tolerance
  - Verification-aware computer architecture
  - Special-purpose processors

Who Are You?

- **People interested in memory consistency models**
  - **Important topic for computer architects and writers of parallel software**
- **People who could figure out the Swedish train system to get here from Arlanda Airport**
  - **SJ? SL?? UL???**

(C) Daniel J. Sorin

Optional Reading

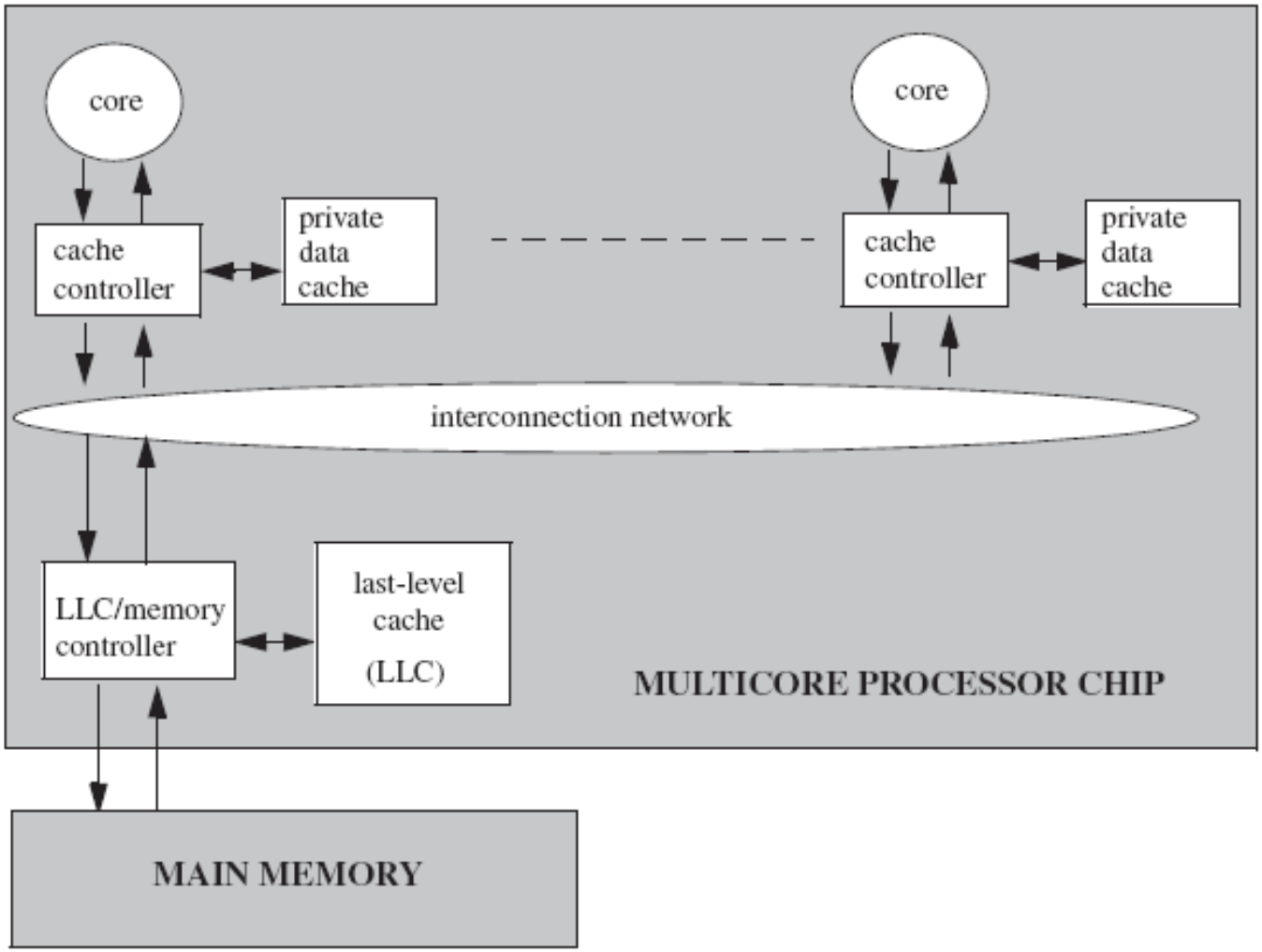
- **Daniel Sorin, Mark Hill, and David Wood. "A Primer on Memory Consistency and Cache Coherence." *Synthesis Lectures on Computer Architecture*, Morgan & Claypool, 2011.**

http://www.morganclaypool.com/doi/abs/10.2200/S00346ED1V01Y201104CAC016

Outline

- **Overview: Shared Memory & Coherence**
  - **Chapters 1-2 of book that you don't have to read**
- **Intro to Memory Consistency**
- **Weak Consistency Models**
- **Case Study in Avoiding Consistency Problems**
- ***Litmus Tests for Consistency***
- ***Including Address Translation***
- ***Consistency for Highly Threaded Cores***

Baseline Multicore System Model



What is (Hardware) Shared Memory?

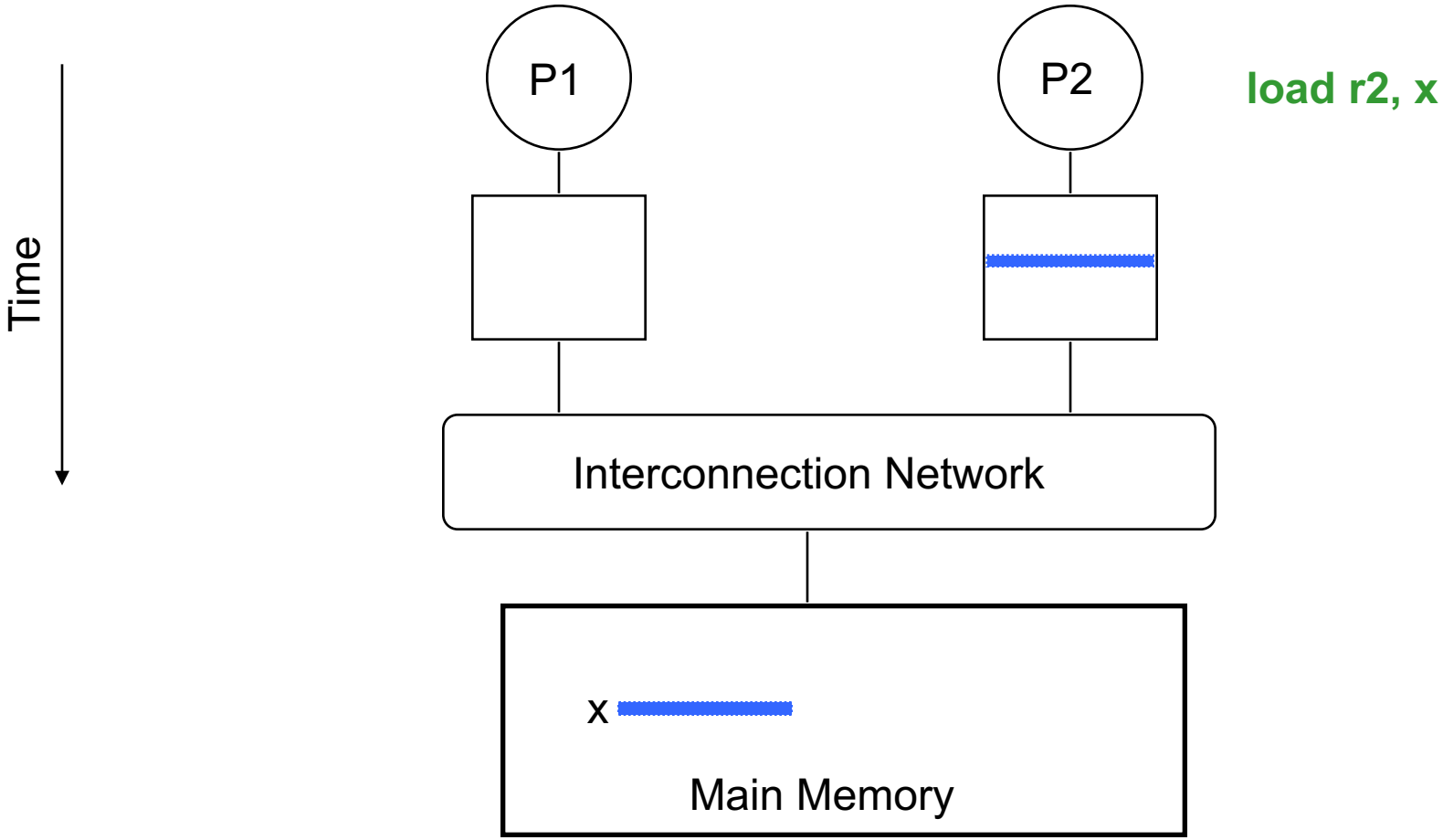
- Take multiple microprocessor cores
- Implement a memory system with a single global physical address space
- Hardware provides illusion of single shared address space
  - Even when cores have caches (next slide)

- Single address space sounds great …
  - … but what happens when cores have caches?

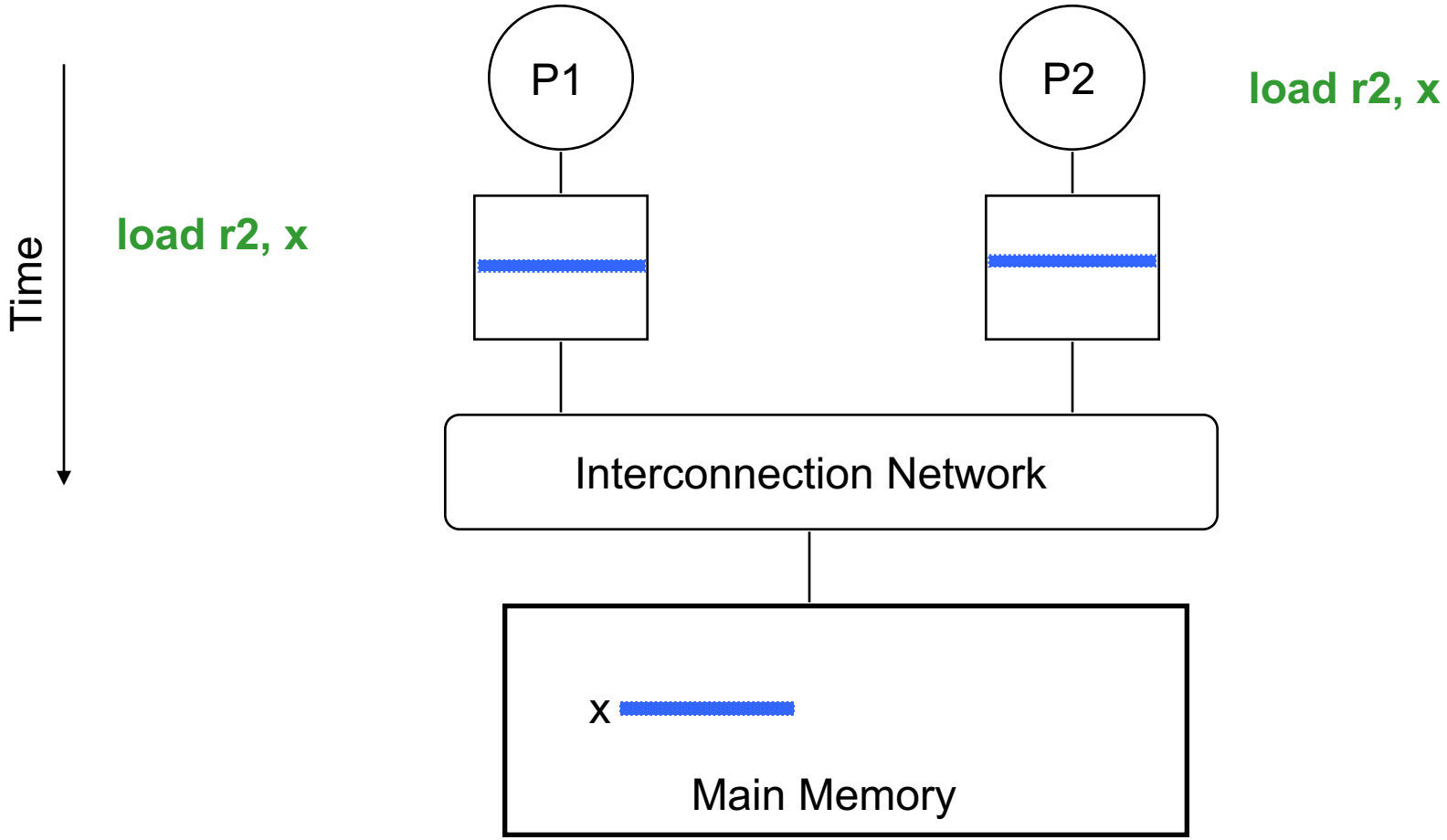
Let's see how caches can lead to incoherence …

(C) Daniel J. Sorin

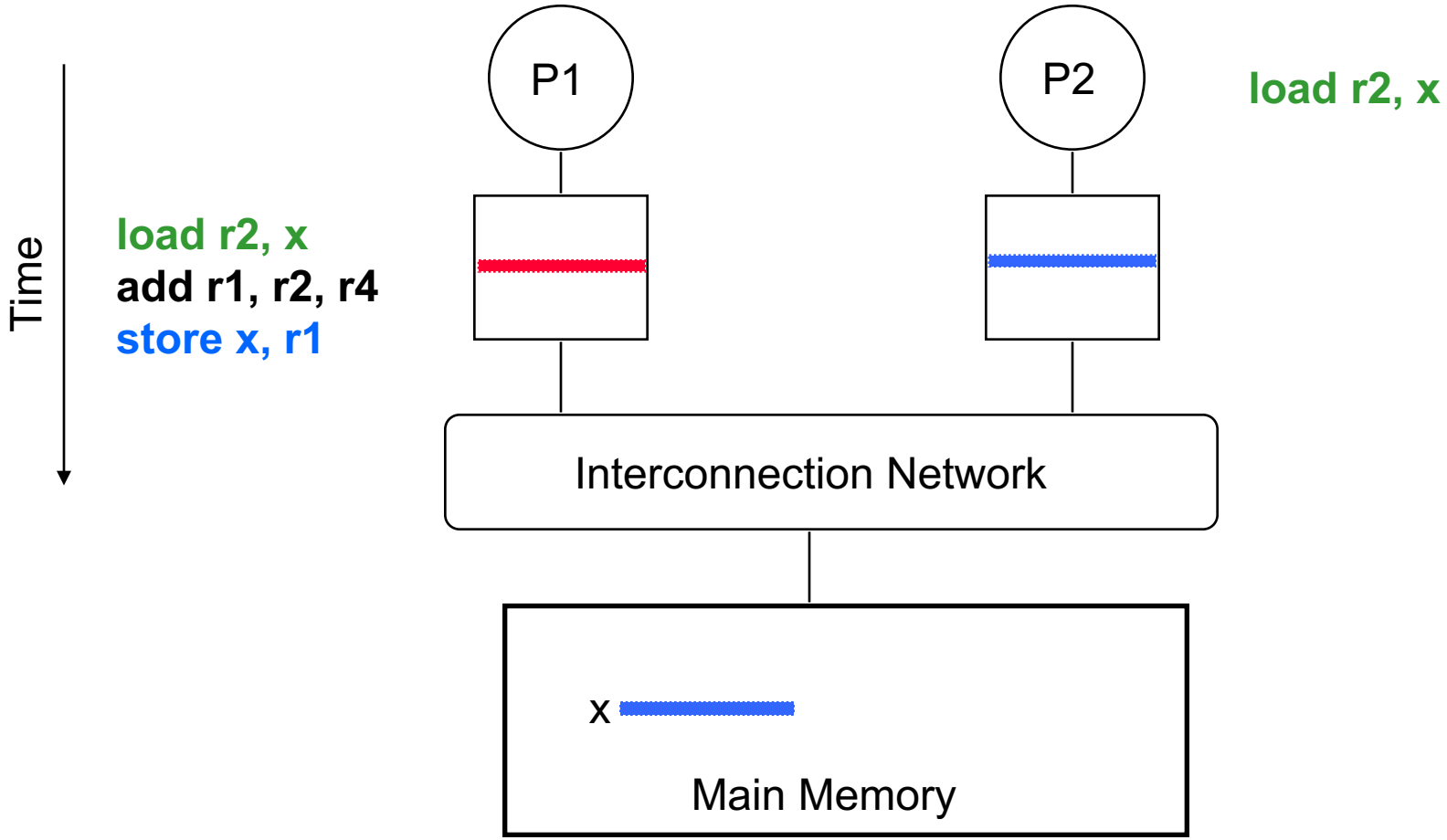
Cache Coherence Problem (Step 1)



Cache Coherence Problem (Step 2)

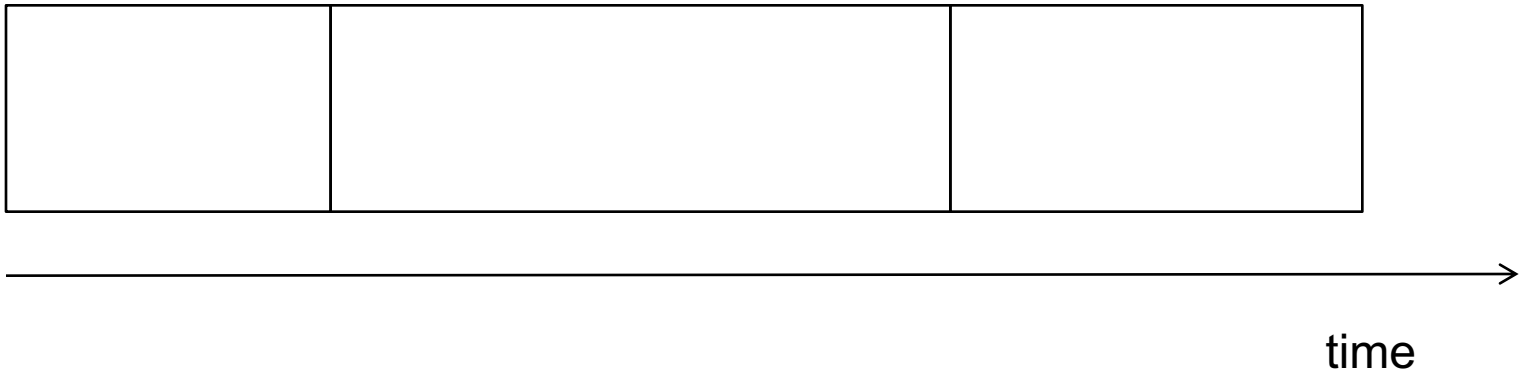


Cache Coherence Problem (Step 3)



Cache Coherence Protocol

- Cache coherence protocol (hardware) enforces two invariants with respect to every block
- We'll think about both invariants in terms of epochs
  - Divide lifetime of each block into epochs of time
- So what are the two invariants?

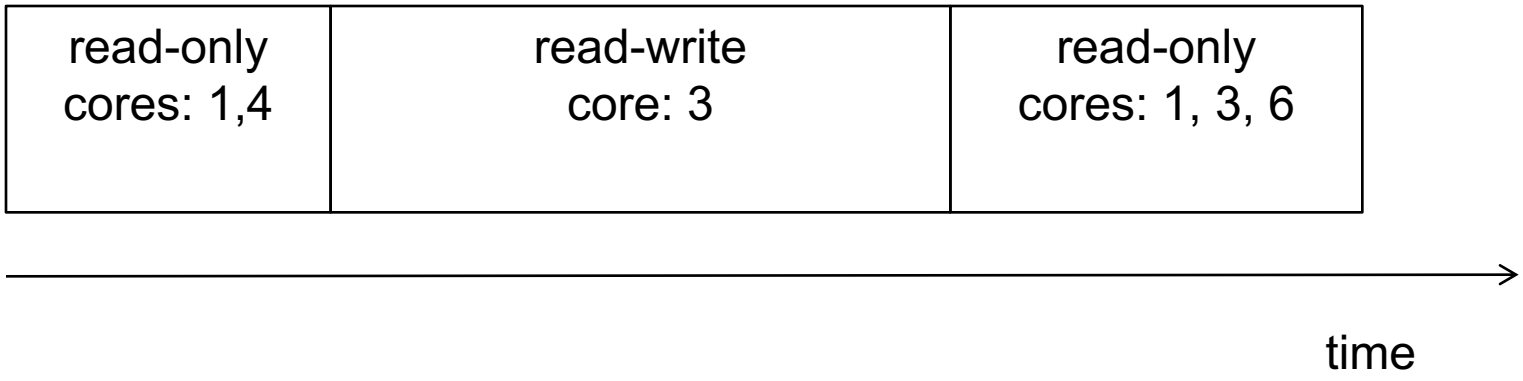


Cache Coherence Invariant #1

1. Single Writer Multiple Reader (SWMR) invariant

SWMR: at any time, a given block either:

- **has one writer → one core has read-write access**
- **zero or more readers → some cores have read-only access**

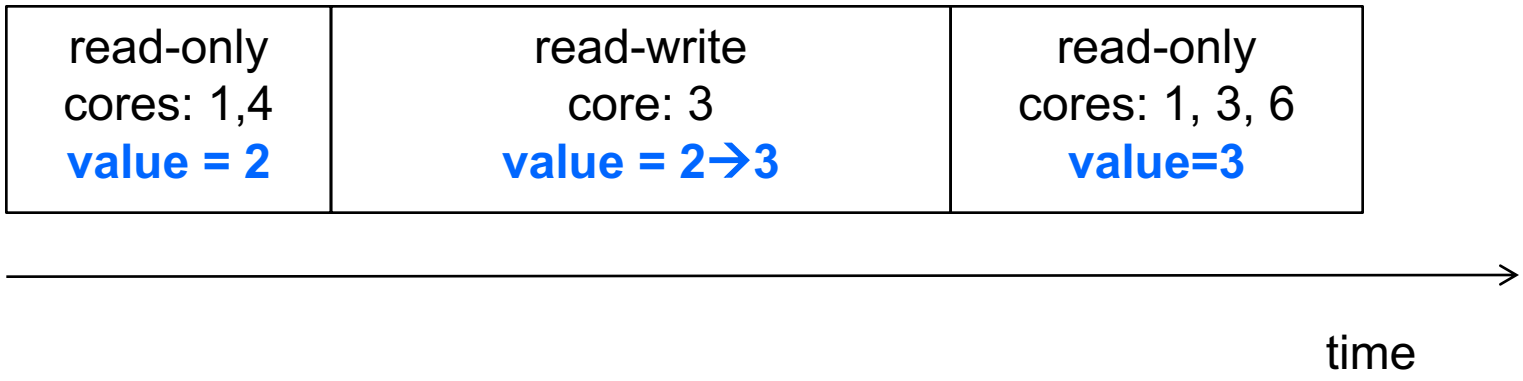


(C) Daniel J. Sorin

Cache Coherence Invariant #2

2. Data invariant: up-to-date data transferred

The value at the beginning of each epoch is equal to the value at the end of the most recently completed read-write epoch



Cache Coherence Protocols

- All any coherence protocol does is enforce these two invariants at runtime
- Many possible ways to do this
- Tradeoffs between performance, scalability, power, cost, etc.

Implementing Cache Coherence Protocols

- But fundamentally all protocols do same thing
- Cache controllers and memory controllers send messages to coordinate who has each block and with what value
- For now, just assume we have a coherence protocol
  - This is one of my favorite topics, so I'll have to refrain for now

Why Cache-Coherent Shared Memory?

- **Pluses**
  - For applications - looks like multitasking uniprocessor
  - For OS - only evolutionary extensions required
  - Easy to do inter-thread communication without OS
  - Software can worry about correctness first and then performance
- **Minuses**
  - Proper synchronization is complex
  - Communication is implicit so may be harder to optimize
  - More work for hardware designers (i.e., me!)
- **Result**
  - Most modern multicore processors provide cache-coherent shared memory

Outline

- Overview: Shared Memory & Coherence
- Intro to Memory Consistency
  - Chapter 3
- Weak Consistency Models
- Case Study in Avoiding Consistency Problems
- Litmus Tests for Consistency
- Including Address Translation
- Consistency for Highly Threaded Cores

Coherence vs. Consistency

- **Programmer's intuition says load should return most recent store to same address**
  - **But which one is the "most recent"?**

- **Coherence concerns each memory location independently**
- **Consistency concerns apparent ordering for ALL memory locations**

Why Coherence != Consistency

```
// initially, A = B = flag = 0
```

Thread 1	Thread 2
Store A = 1;	while (Load flag==0); // spin
Store B = 1;	Load A;
Store flag = 1;	Load B;
	print A and B;

- Intuition says Thread 2 should print A = B = 1
- Yet, in some consistency models, this isn't required!
- Coherence doesn't say anything … why?

Why Memory Consistency is Important

- **Memory consistency model defines correct behavior**
  - **It is contract between system and programmer**
  - **Analogous to ISA specification**
  - **Consistency is part of architecture → software-visible**
- **Coherence protocol is only a means to an end**
  - **Coherence is not visible to software (i.e., not architectural)**
  - **Enables new system to present same consistency model despite using newer, fancier coherence protocol**
  - **Systems maintain backward compatibility for consistency (like ISA)**
- **Reminder to architects: consistency model restricts ordering of loads/stores**
  - **Does NOT care at all about ordering of coherence messages**

(C) Daniel J. Sorin

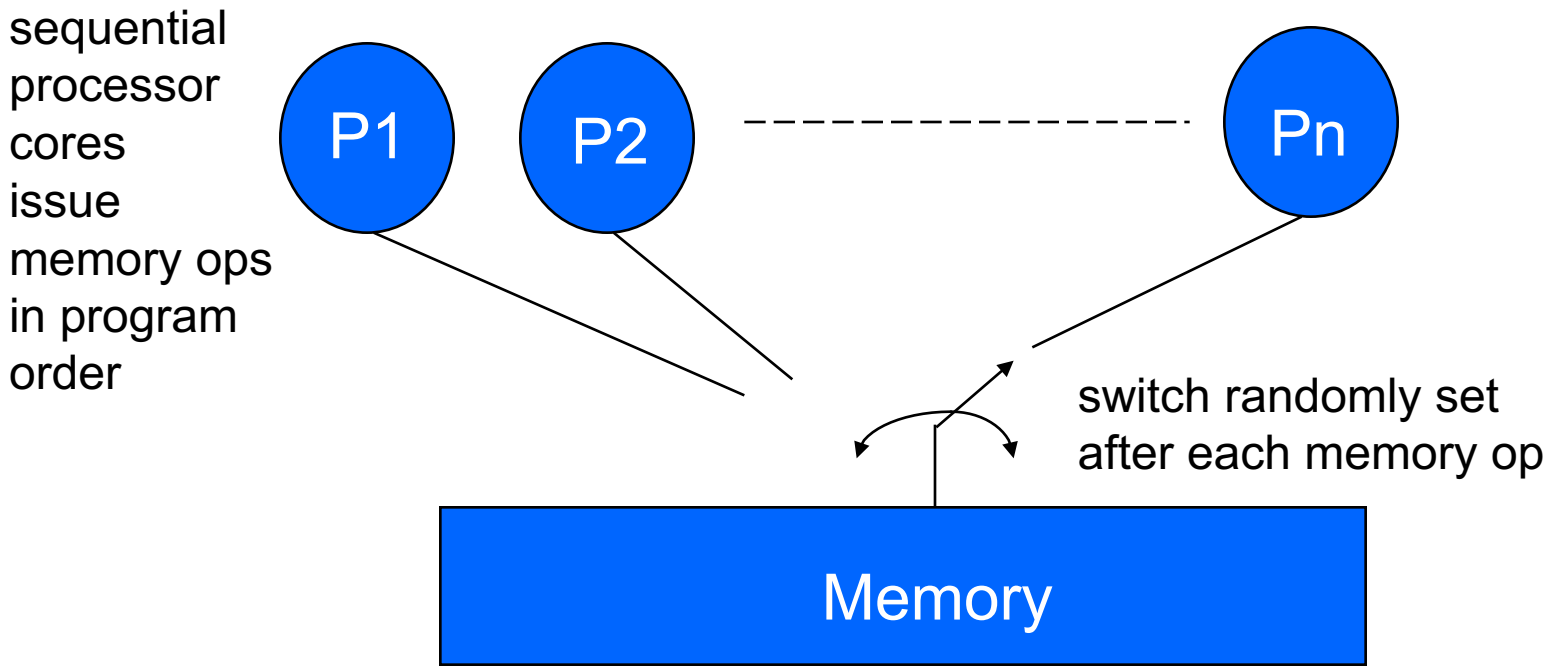
Sequential Consistency (SC)

- **Leslie Lamport 1979:**

"A multiprocessor is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program"

- **First precise definition of consistency**
- **Most restrictive consistency model**
- **Most intuitive model for (most) humans**

The Memory Model



SC: Definitions

- **Sequentially consistent execution**
  - **Result is same as one of the possible interleavings on uniprocessor**
- **Sequentially consistent system**
  - **Any possible execution corresponds to some possible total order**

- **Preferred (and equivalent) definition of SC**
  - **There exists a total order of all loads and stores (across all threads), such that the value returned by each load equals the value of the most recent store to that location**

SC: More Definitions

- **Memory operation**
  - Load, store, or atomic **read-modify-write (RMW)** to memory location
- **Issue (different from "issue" within core!)**
  - An operation is issued when it leaves core and is presented to memory system (usually the L1 cache or write-buffer)
- **Perform**
  - A store is performed wrt to a processor core P when a load by P returns value produced by that store or a later store
  - A load is performed wrt to a processor core when subsequent stores cannot affect value returned by that load
- **Complete**
  - A memory operation is complete when performed wrt all cores.
- **Program execution**
  - Memory operations for specific run only (ignore non-memory-referencing instructions)

(C) Daniel J. Sorin

SC: Table-Based Definition

- **I like tabular definitions of models**
  - Specify which program orderings are enforced by consistency model
    - » Remember: program order defined per thread
  - Includes loads, stores, and atomic read-modify-writes (RMWs)
  - "X" denotes ordering enforced

		Operation 2		
		Load	Store	RMW
Operation 1	Load	X	X	X
	Store	X	X	X
	RMW	X	X	X

SC and Out-of-Order (OOO) Cores

- At first glance, SC seems to require in-order cores
- Conservative way to support SC
  - Each core issues its memory ops in program order
  - Core must wait for store to complete before issuing next memory operation
  - After load, issuing core waits for load to complete, and store that produced value to complete before issuing next op
  - Easily implemented if cores connected with shared (physical) bus
- But remember: SC is an abstraction
  - Difference between architecture and micro-architecture
- Can do whatever you want, if illusion of SC!

(C) Daniel J. Sorin

Optimized Implementations of SC

- **Famous paper by Gharachorloo et al. [ICPP 1991] shows two techniques for optimization of OOO core**
  - **Both based on consistency speculation**
  - **That is: speculatively execute and undo if violate SC**
  - **In general, speculate by issuing loads early and detecting whether that can lead to violations of SC**
- **MIPS R10000-style speculation**
  - **Non-speculatively issue & commit stores at Commit stage (in order)**
  - **Speculatively issue loads at Execute stage (out-of-order)**
  - **Track addresses of loads between Execute and Commit**
  - **If other core does store to tracked address (detected via coherence protocol) → mis-speculation**
  - **Why does this work?**

Optimized Implementations of SC, part 2

- **Data-replay speculation**
  - **Non-speculatively issue & commit stores at Commit stage (in order)**
  - **Speculatively issue loads at Execute stage (out-of-order)**
  - **Replay loads at Commit**
  - **If load value at Execute doesn't equal value at Commit → mis-speculation**
  - **Why does this work?**

- **Key idea: consistency is interface (illusion)**
  - **If software can't tell hardware violated consistency, it's OK**
  - **Analogous to cores that execute out-of-order while presenting in-order (von Neumann) illusion**

Outline

- Overview: Shared Memory & Coherence
- Intro to Memory Consistency
- Weak Consistency Models
  - Chapters 4-5
- Case Study in Avoiding Consistency Problems
- Litmus Tests for Consistency
- Including Address Translation
- Consistency for Highly Threaded Cores

Why Relaxed Memory Models?

- **Recall SC requires strict ordering of reads/writes**
  - **Each processor generates a local total order of its reads and writes (R→R, R→W, W→W, & R→W)**
  - **All local total orders are interleaved into global total order**

		Operation 2		
		Load	Store	RMW
Operation 1	Load	X	X	X
	Store	X	X	X
	RMW	X	X	X

Why Relaxed Memory Models?

- **Relaxed models relax some of these constraints**
  - **TSO: Relax ordering from writes to reads (to diff addresses)**
  - **XC: Relax all read/write orderings (but add "fences")**
- **Why do we care?**
  - **May allow hardware optimizations prohibited by SC**
  - **May allow compiler optimizations prohibited by SC**
- **Many possible models weaker than SC**

Let's start with Total Store Order (TSO) …

(C) Daniel J. Sorin

TSO/x86

- **Total Store Order (TSO)**
  - **First defined by Sun Microsystems**
  - **Later shown that Intel/AMD x86 is nearly identical → "TSO/x86"**
- **Less restrictive than SC**
  - **Tabular ordering of loads, stores, RMWs, and FENCEs**
  - **X=ordered**
  - **B=data value bypassing required if to same address**

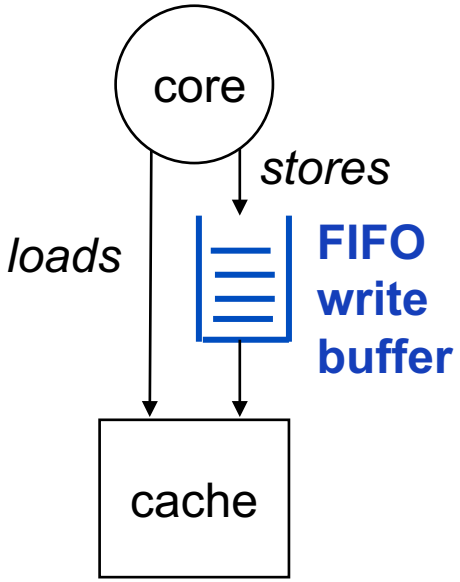
		Operation 2			
		Load	Store	RMW	FENCE
Operation 1	Load	X	X	X	X
	Store	B	X	X	X
	RMW	X	X	X	X
	FENCE	X	X	X	X

TSO/x86: Relax Write to Read Order

// initially, A = B = 0

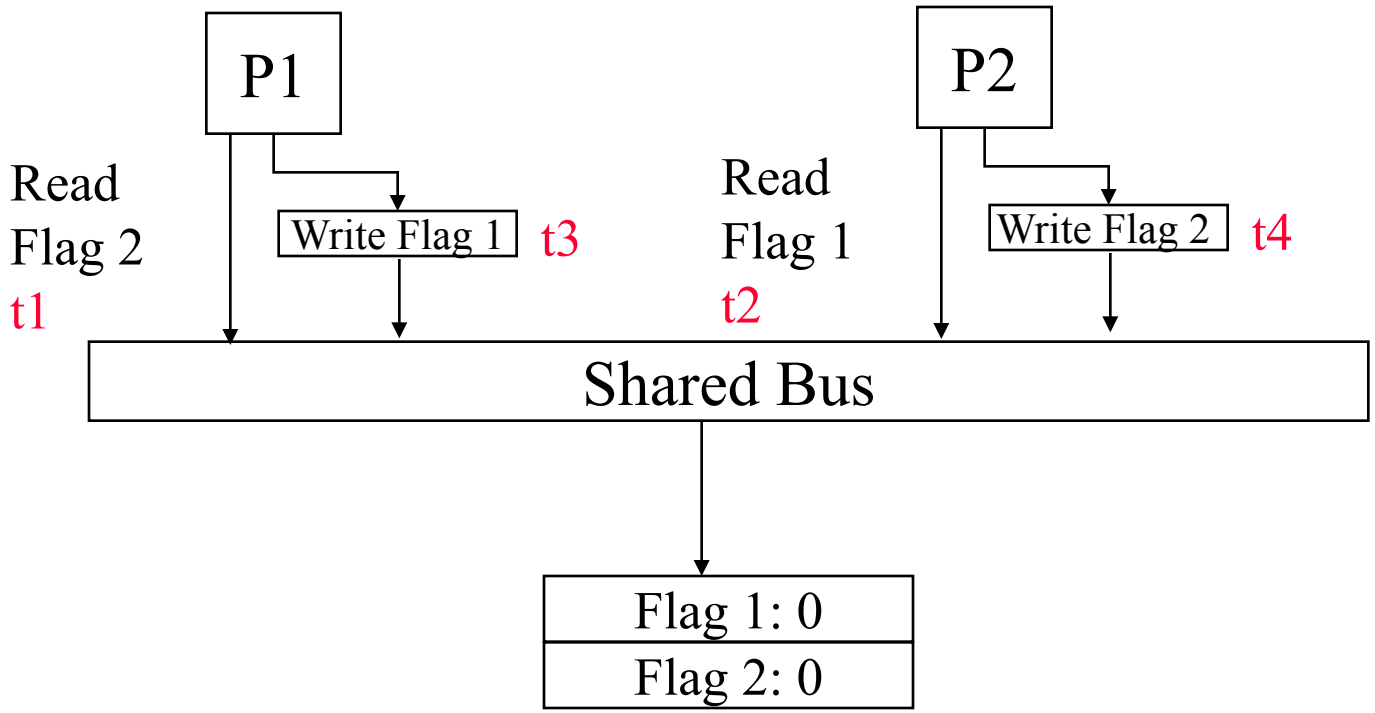
Thread 1	Thread 2
Store A = 1;	Store B = 1
Load r1 = B;	Load r2 = A;

- **TSO/x86**
  - Allows r1==r2==0 (not allowed by SC)
- **Why do this?**
  - Allows FIFO write buffers → performance!
  - Does not confuse programmers (too much)



(C) Daniel J. Sorin

Write Buffers w/ Read Bypass



Thread 1

```
Flag 1 = 1
if (Flag 2 == 0)
   critical section
```

Thread 2

```
Flag 2 = 1
if (Flag 1 == 0)
  critical section
```

TSO/x86: Adding Order When Needed

// initially, A = B = 0

Thread 1 (T1)	Thread 2 (T2)
Store A = 1;	Store B = 1
FENCE	FENCE
Load r1 = B;	Load r2 = A;

- Need to add explicit ordering if you want it
  - Unlike SC, where everything ordered by default
- FENCE instruction provides ordering
  - FENCE is part of ISA
  - No instructions can be reordered across FENCE
    - » E.g., FENCE prohibits Ld r1=B from occurring before St A=1

TSO Also Provides "Causality" (Transitivity)

// initially all locations are 0

T1	T2	T3
St A = 1;	while (Ld flag1==0) {};	while (Ld flag2==0) {};
St flag1 = 1;	St flag2 = 1;	Ld r3 = A;

- We expect T3's Ld r3=A to get value 1
- All commercial versions of TSO guarantee causality

So Why Not Relax All Order?

```
// initially all 0
 T1                T2
                   L1: Ld r1 = flag;          // spin
St A=1;                if (r1 != 1) goto L1   // loop
St B=1;                Ld r1 = A;
St flag = 1;           Ld r2 = B;
```

- SC and TSO always order red ops & order green ops
  - But that's overkill → we don't need to order them
  - Reordering could allow for OOO processors, non-FIFO write buffers, some coherence optimizations, etc.
- Opportunity: instead of ordering everything by default, only order when you need it

But What's the Catch?

```
// initially all 0
 T1                T2
                   L1: Ld r1 = flag;          // spin
St A=1;                if (r1 != 1) goto L1   // loop
St B=1;                Ld r1 = A;
St flag = 1;           Ld r2 = B;
```

- What if St flag=1 can be reordered before St A=1?
- Or if Ld r1=A can be reordered before loading flag=1?
- We want some order
  - Red ops before St flag=1
  - Green ops after loading flag=1

Order with FENCE Operations

```
// initially all 0
 T1                          T2
                        L1: Ld r1 = flag;          // spin
St A=1;                     if (r1 != 1) goto L1   // loop
St B = 1;                   FENCE;
FENCE;                      Ld r1=B;
St flag = 1;                Ld r2 = B;
```

- **FENCE orders everything above it before everything after it**
  - **T1's FENCE: If thread sees flag=1, must also see A=1, B=1**
  - **T2's FENCE: T2 can't do loads of r1,r2 before seeing flag set**

Many Flavors of Weak Models

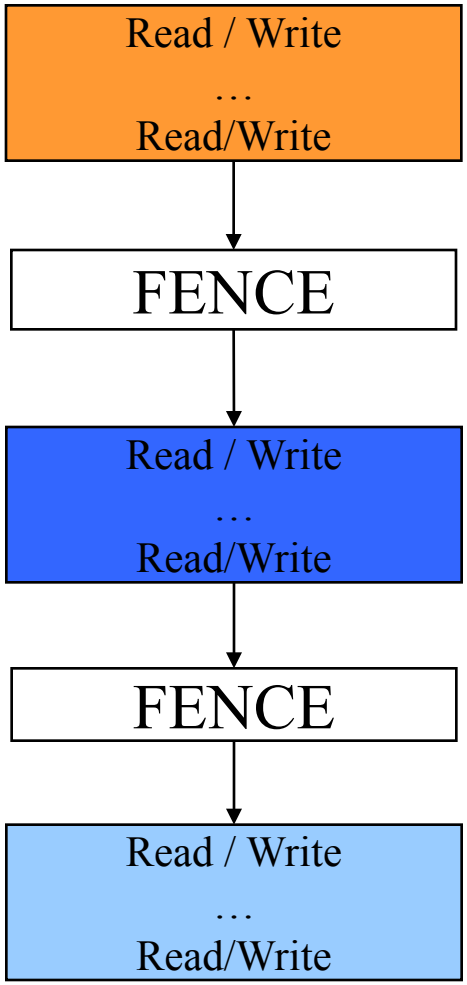
- **Many possible models weaker than SC and TSO**
  - **Most differences pretty subtle**
- **XC in primer (like what is often called Weak Ordering)**
  - **One type of FENCE**
  - **X=order, A=order if same address, B=bypassing if same address**

		Operation 2			
		Load	Store	RMW	FENCE
Operation 1	Load	A	A	A	X
	Store	B	A	A	X
	RMW	A	A	A	X
	FENCE	X	X	X	X

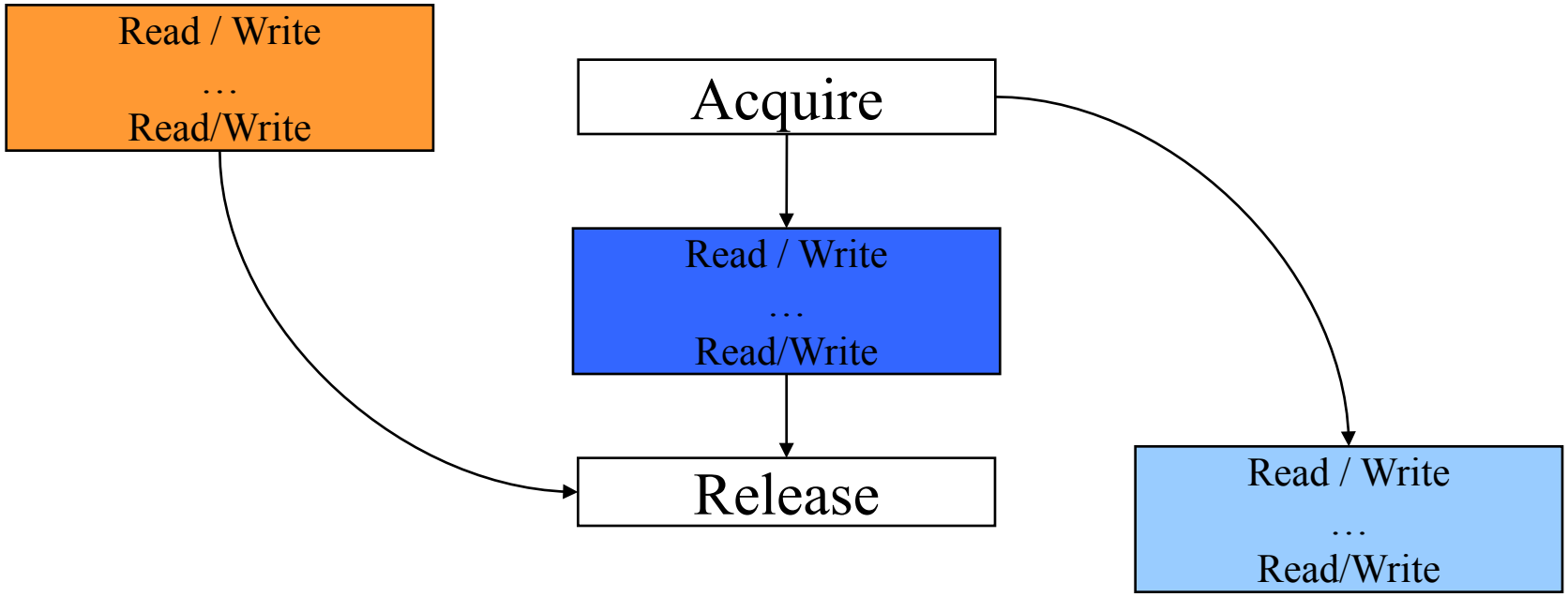
Release Consistency (RC)

- **Like XC but two types of one-way FENCEs**
  - **Acquire and Release**
- **Acquire: Acquire→Ld, St**
- **Release: Ld, St → Release**

XC Example


Read / Write
…
Read/Write
FENCE
Read / Write
…
Read/Write
FENCE
Read / Write
…
Read/Write

Release Consistency Example



(C) Daniel J. Sorin

The Programming Interface

- **XC and RC require synchronized programs**
- **All synchronization operations must be labeled and visible to the hardware**
  - **Easy (easier!) if synchronization library used**
  - **Must provide language support for arbitrary Ld/St synchronization (event notification, e.g., flag)**
- **Program written for weaker model OK on stricter**
  - **E.g., SC is a valid implementation of TSO, XC, or RC**

(C) Daniel J. Sorin

SC for Data-Race-Free

- **Data race: two accesses by two threads where:**
  - **At least one is a write**
  - **They're not separated by synchronization operations**
- **Data-race-free (DRF) program has no data races**
  - **Most correct programs are DRF – can you think of counter-examples?**

IMPORTANT RESULT

- **TSO, XC, and RC all provide "SC for DRF"**
  - **If program is DRF, then behavior is sequentially consistent**
  - **Allows programmer to reason about SC system!**
- **But what if program isn't DRF (i.e., has a bug)?**
  - **Debugging becomes much more … interesting**

Outline

- Overview: Shared Memory & Coherence
- Intro to Memory Consistency
- Weak Consistency Models
- Case Study in Avoiding Consistency Problems
- Litmus Tests for Consistency
- Including Address Translation
- Consistency for Highly Threaded Cores

(C) Daniel J. Sorin

Why Architects Must Understand Consistency: A Case Study

- **What happens when memory consistency interacts with value prediction?**
- **Hint: it's not obvious!**

- **Note: this is not an important problem in itself → the key is to show you how you must think about consistency when designing multicore processors**

(C) Daniel J. Sorin

Informal Example of Problem, part 1

- **Student #2 predicts grades are on bulletin board B**
- **Based on prediction, assumes score is 60**

Bulletin Board B

Grades for Class

Student ID	score
1	75
2	60
3	85

(C) Daniel J. Sorin

Informal Example of Problem, part 2

- **Professor now posts actual grades for this class**
  - **Student #2 actually got a score of 80**
- **Announces to students that grades are on board B**

Grades for Class

Student ID	score	
1	75	50
2	60	80
3	85	70

Informal Example of Problem, part 3

- Student #2 sees prof's announcement and says,

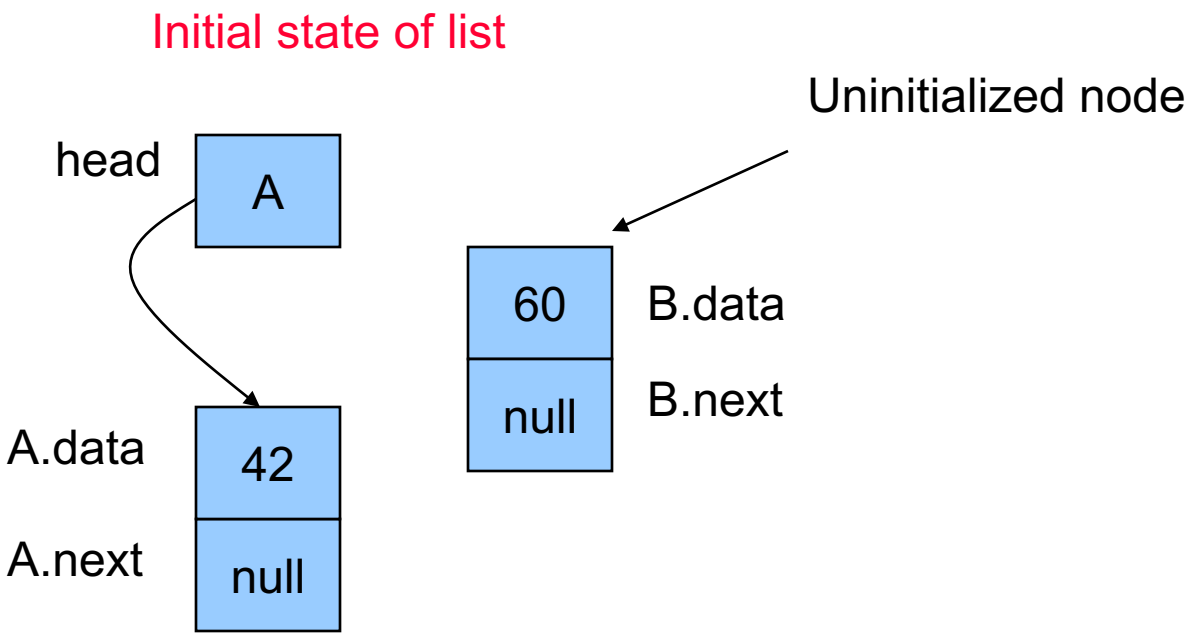
  " I made the right prediction (bulletin board B), and my score is 60"!
- Actually, Student #2's score is 80

- What went wrong here?
  - Intuition: predicted value from future
- Problem is concurrency
  - Interaction between student and professor
  - Just like multiple threads, cores, or devices

(C) Daniel J. Sorin

Linked List Example of Problem (initial state)

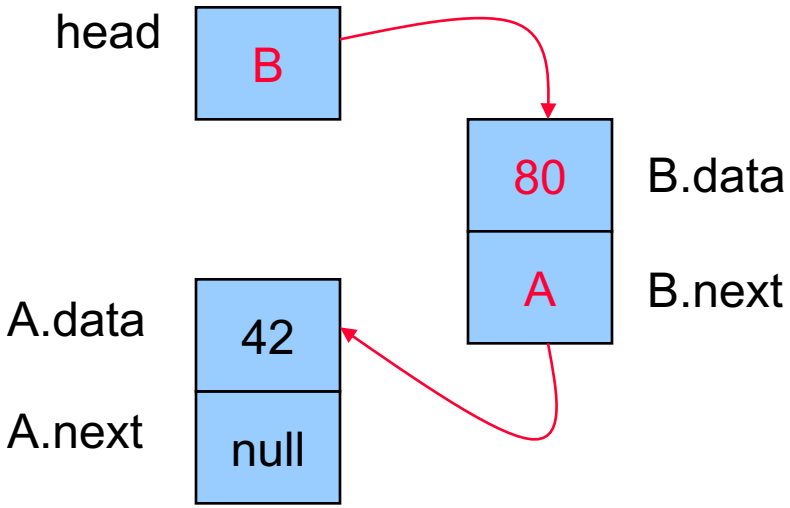
- Linked list with single writer and single reader
- No synchronization (e.g., locks) needed



(C) Daniel J. Sorin

Linked List Example of Problem (Writer)

- Writer sets up node B and inserts it into list



Code For Writer Thread

Setup node

W1: store mem[B.data] ← 80

W2: load reg0 ← mem[Head]

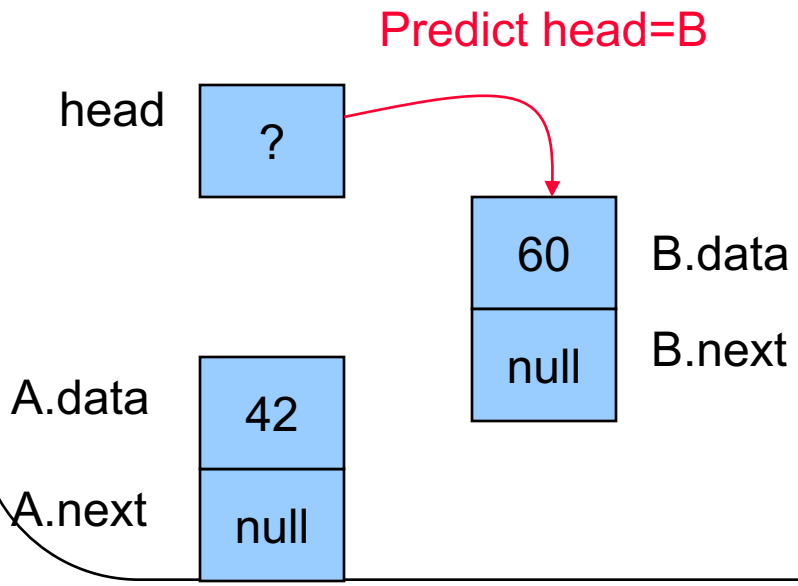
W3: store mem[B.next] ← reg0

Insert

W4: store mem[Head] ←B

Linked List Example of Problem (Reader)

- Reader cache misses on head and value predicts head=B.
- Cache hits on B.data and reads 60.
- Later "verifies" prediction of B. Is this execution legal?



Code For Reader Thread

R1: load reg1 ← mem[Head] = B

R2: load reg2 ← mem[reg1] = 60

(C) Daniel J. Sorin

Why This Execution Violates SC

- **Recall Sequential Consistency**
  - **Must exist total order of all operations**
  - **Total order must respect program order at each processor**

- **Our example execution has a cycle**
  - **No total order exists**

(C) Daniel J. Sorin

Trying to Find a Total Order

- What orderings are enforced in this example?

Code For Writer Thread

Setup node

W1: store mem[B.data] ← 80

W2: load reg0 ← mem[Head]

W3: store mem[B.next] ← reg0

Insert

W4: store mem[Head] ← B

Code For Reader Thread

R1: load reg1 ← mem[Head]

R2: load reg2 ← mem[reg1]

(C) Daniel J. Sorin

Program Order

- Must enforce program order

Code For Writer Thread

W1: store mem[B.data] ← 80

W2: load reg0 ← mem[Head]

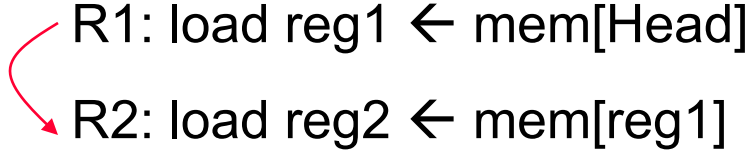
W3: store mem[B.next] ← reg0

W4: store mem[Head] ← B

Code For Reader Thread

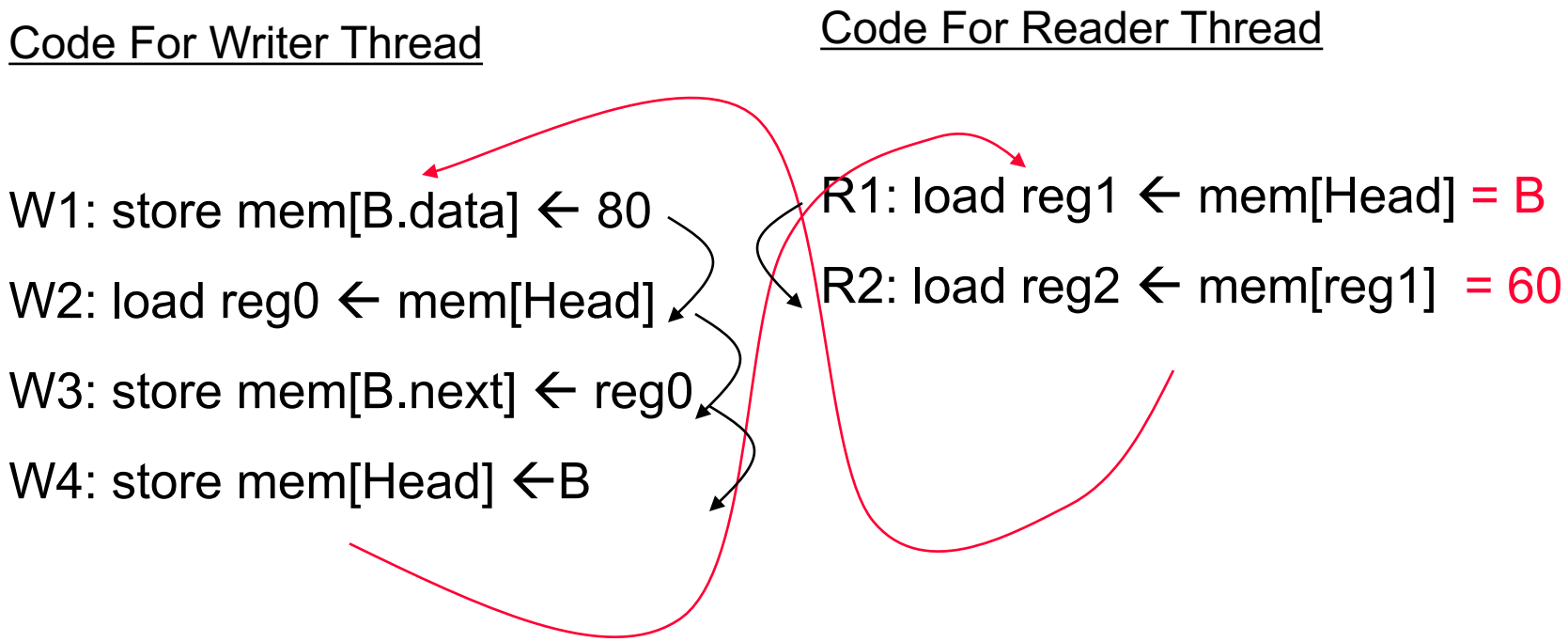
R1: load reg1 ← mem[Head]

R2: load reg2 ← mem[reg1]



Data Order

- If we predict that R1 returns the value B, we can violate SC



(C) Daniel J. Sorin

Value Prediction and Sequential Consistency

- Key: value prediction reorders dependent operations
  - Specifically, read-to-read data dependence order
- Execute dependent operations out of program order

- Applies to almost all consistency models
  - Models that enforce data dependence order

- Must detect when this happens and recover
- Similar to other optimizations that complicate SC

(C) Daniel J. Sorin

How to Fix SC Implementations w/Value Pred

- Two options from "Two Techniques for …"
  - Both adapted from ICPP '91 paper
  - Originally developed for out-of-order SC cores
- (1) Address-based detection of violations
  - Student watches board B between prediction and verification
  - Like existing techniques for out-of-order SC processors
  - Track stores from other threads
  - If address matches speculative load, possible violation
- (2) Value-based detection of violations
  - Student checks grade again at verification
  - Also an existing idea
  - Replay all speculative instructions at commit
  - Can be done with dynamic verification (e.g., DIVA [MICRO '99])

(C) Daniel J. Sorin

Outline

- Overview: Shared Memory & Coherence
- Intro to Memory Consistency
- Weak Consistency Models
- Case Study in Avoiding Consistency Problems
- Litmus Tests for Consistency
- Including Address Translation
- Consistency for Highly Threaded Cores

(C) Daniel J. Sorin

Litmus Tests

- **Goal: short code snippets to test consistency model**
- **Run litmus test many times (hoping for many different inter-thread interleavings)**
  - **Make sure no execution produces result that violates consistency model**
- **We've already seen a few litmus tests**

Litmus Test #1: SC vs. TSO

// initially, A = B = 0

T1	T2
Store A = 1;	Store B = 1
Load r1 = B;	Load r2 = A;

- SC: r1=r2=0 not allowed → cyclic dependence graph
- TSO/x86: all outcomes allowed, including r1=r2=0

Litmus Test #2: TSO vs. XC

```
// initially, A = B = flag = 0
 T1                      T2
St A = 1;                while (Ld flag == 0); // spin
St B = 1;                Ld A;
St flag = 1;             Ld B;
                         print A and B
```

- TSO requires T2 to print A = B = 1
- XC permits other results

Litmus Test #3: Transitivity

// initially all locations are 0

T1	T2	T3
St A = 1;	while (Ld flag1==0) {};	while (Ld flag2==0) {};
St flag1 = 1;	St flag2 = 1;	Ld r3 = A;

- We expect T3's Ld r3=A to get value 1
- All commercial versions of TSO guarantee causality

Litmus Test #4: IRIW

// Independent Read, Independent Write

// initially all locations are 0

T1	T2	T3	T4
St A = 1;	St B=1;	Ld A; // =1	Ld B; // =1
		FENCE;	FENCE;
		Ld B; // =1?	Ld A; // =1?

- Well-known litmus test to check for "write atomicity"
  - Store is logically seen by all cores at once
  - Some relaxed models enforce write atomicity
- What happens if last two loads both equal 0?
  - No order of stores exists → no write atomicity

(C) Daniel J. Sorin

More Litmus Tests

- **Many more litmus tests exist**
- **Useful for testing and debugging hardware**
- **Useful for reasoning about consistency models**

Outline

- Overview: Shared Memory & Coherence
- Intro to Memory Consistency
- Weak Consistency Models
- Case Study in Avoiding Consistency Problems
- Litmus Tests for Consistency
- Including Address Translation
- Consistency for Highly Threaded Cores

(C) Daniel J. Sorin

Translation-oblivious Memory Consistency

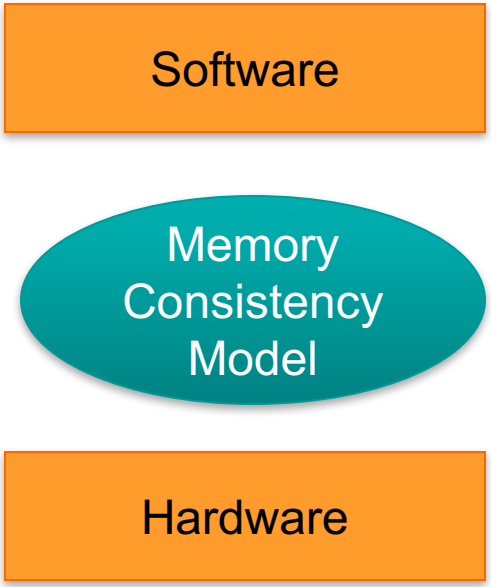
- **Lamport's definition of Sequential Consistency**
  - **Operations of individual processor appear in program order**

    on Physical or Virtual addresses?
  - **The total order of operations executed by different processors obeys some sequential order**
- **Memory system includes Address Translation (AT)**
  - **We need AT-aware specifications**

(C) Daniel J. Sorin

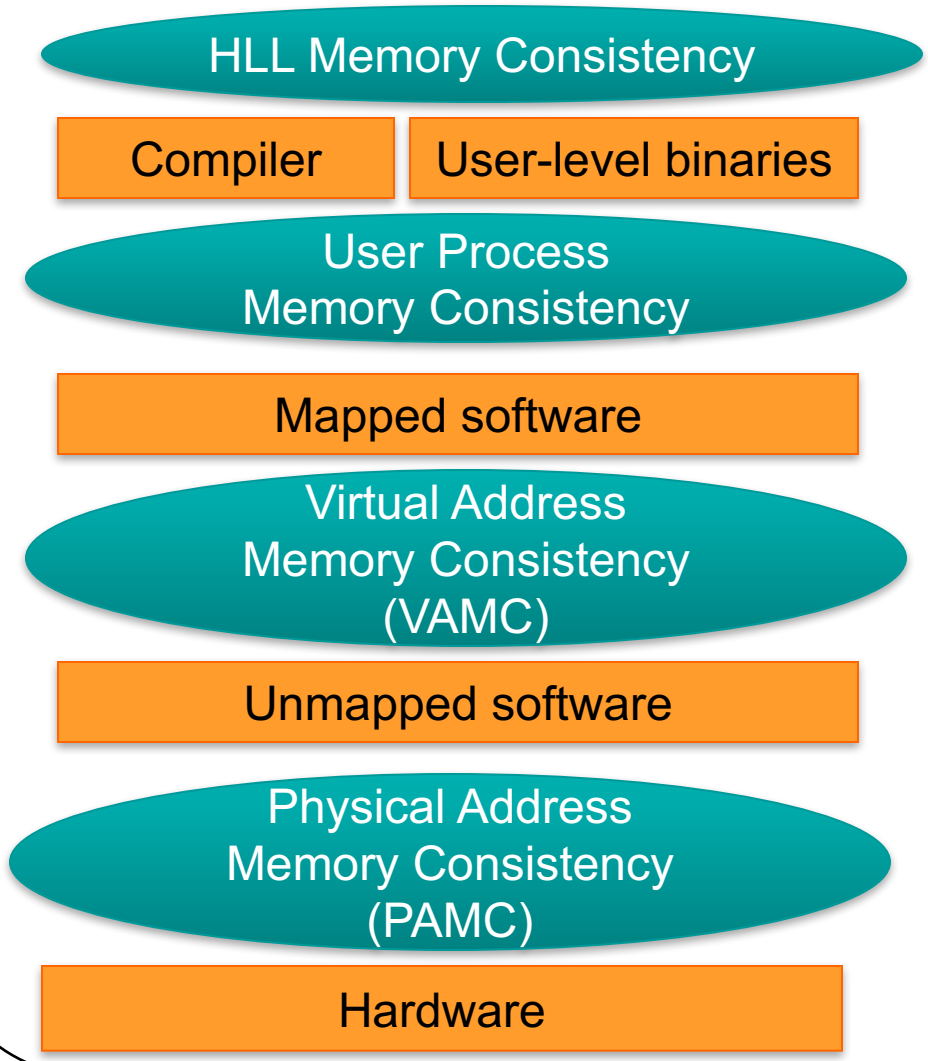
Memory Consistency – Traditional View

- **Monolithic interface between hardware and software**



(C) Daniel J. Sorin

Memory Consistency – Multi-level View



- **Memory consistency represents a set of interfaces**
  - **Supports different layers of software**
- **AT supports mapped software**
  - **Interacts with PAMC and VAMC**
  - **How does AT impact their specifications?**

(C) Daniel J. Sorin

PAMC – Physical Address Consistency

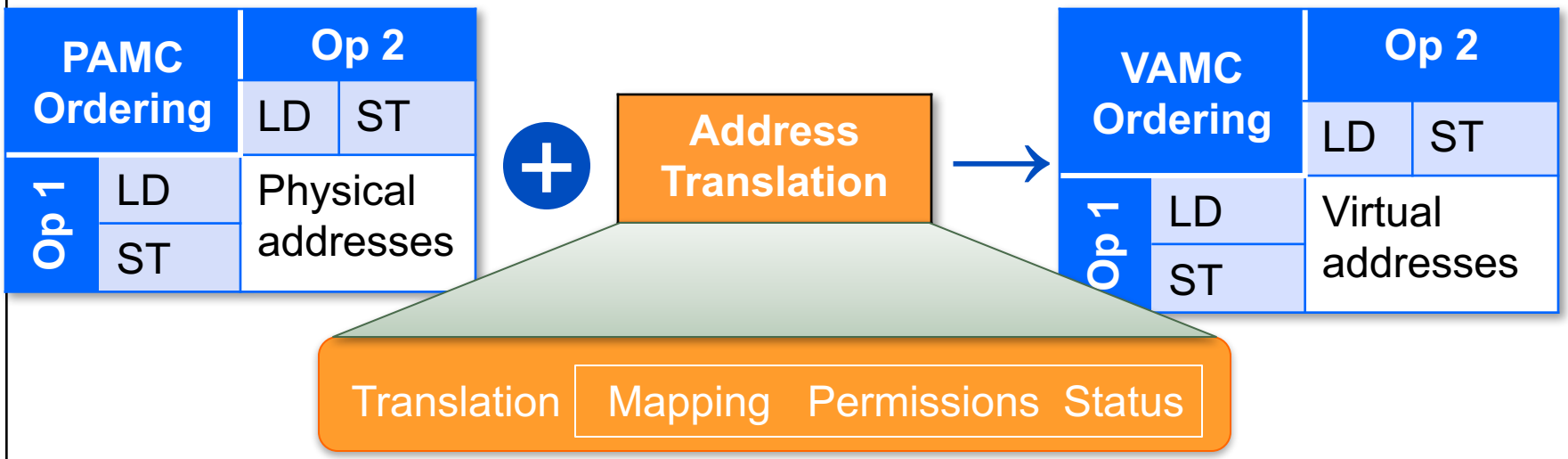
- **Supports unmapped software**
  - **Relies only on hardware**
  - **Fully specified by the architecture**
- **Adapting AT-oblivious specifications straightforward**
  - **All operations refer to physical addresses**

Weak Order PAMC		Operation 2		
		LD phys	ST phys	MemBar
Operation 1	LD phys		A	X
	ST phys	A	A	X
	MemBar	X	X	X

Legend
X = enforced order
A = order if same physical address

(C) Daniel J. Sorin

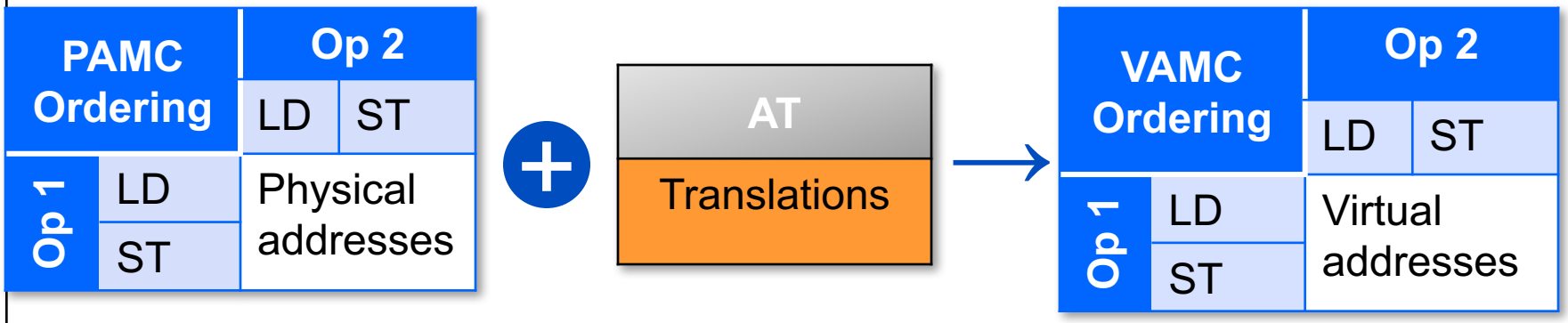
From PAMC to VAMC



- **Translations**
  - **Regulate Virtual→Physical address conversions through mappings**
  - **Include permissions and status bits**
  - **Defined in memory page table, cached in TLBs for expedited access**

(C) Daniel J. Sorin

AT's Impact on VAMC

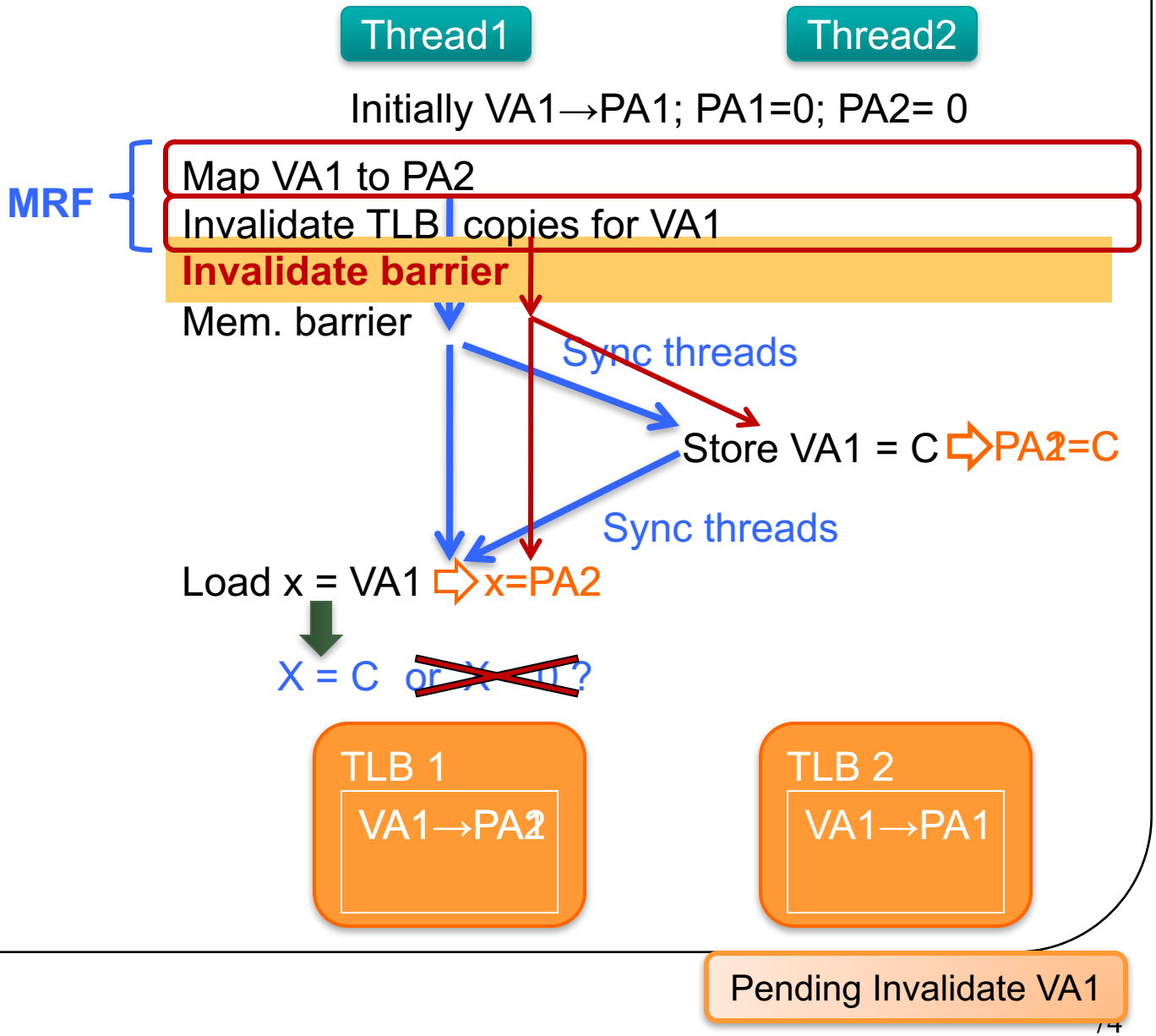


- **Intuitively, PAMC + AT = VAMC**
- **Three AT aspects impact VAMC**
  - **Synonyms - multiple virtual addresses for same data**
  - **Mappings/permissions changes**
    - » **Map/Remap Functions (MRFs)**
    - » **Maintain coherence between page table and TLBs**
  - **Status bit updates**

(C) Daniel J. Sorin

Why MRF Ordering Matters

- **Two threads operating on same virtual address VA1**
- **TLB Invalidation ordering impacts final result**
- **Enforcing MRF ordering eliminates ambiguity**



(C) Daniel J. Sorin

Specifying AT-Aware VAMC

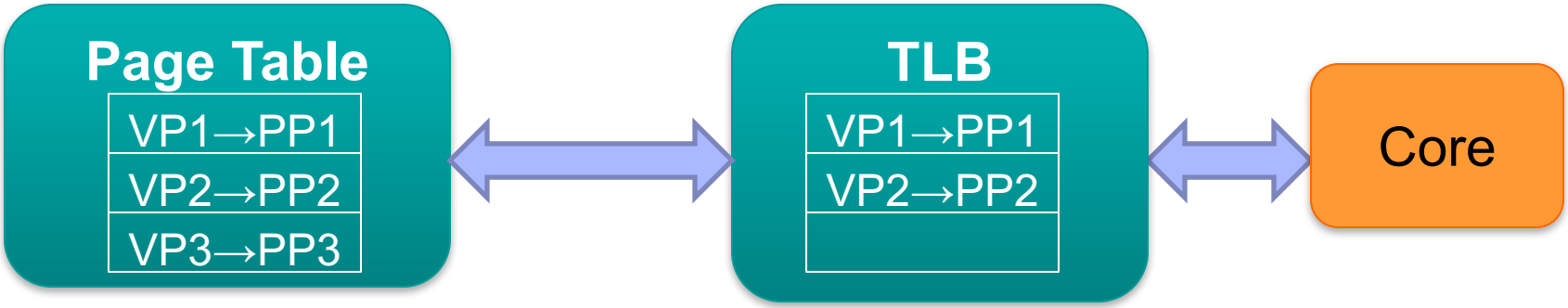
- **Possible VAMC specification based on Weak Order**
  - LD/ST refer to synonym sets of virtual addresses
  - MRFs are serialized wrt. any other operation
  - Status bits updates ordered only wrt. to MemBar and MRF
- Correct AT is critical for VAMC correctness

Weak Order VAMC		Operation 2				
		LD syn	ST syn	MemBar	MRF	SB
Operation 1	LD syn		A	X	X	
	ST syn	A	A	X	X	
	MemBar	X	X	X	X	X
	MRF	X	X	X	X	X
	SB			X	X	

Legend
X = enforced order
A = order if same address synonym set

Framework for AT Specifications

- **Framework characterizes AT state, not specific implementation**
- **Translations defined in page table, cached in TLBs**



- Invariant #1. Page table is correct
  - Software-managed data structure
- Invariant #2. Translations are coherent
  - Hardware/software managed

(C) Daniel J. Sorin

AT Model - ATsc

- **Sequential model of AT**
  - Similar, but not identical to AT models supported by x86 hardware running Linux
  - Translation accesses and status bit updates occur atomically with instructions
  - MRFs are logically atomic
    - » Implementation uses locks
- **Model supports PAMCsc + ATsc = VAMCsc**

(C) Daniel J. Sorin

Outline

- Overview: Shared Memory & Coherence
- Intro to Memory Consistency
- Weak Consistency Models
- Case Study in Avoiding Consistency Problems
- Litmus Tests for Consistency
- Including Address Translation
- Consistency for Highly Threaded Cores

(C) Daniel J. Sorin

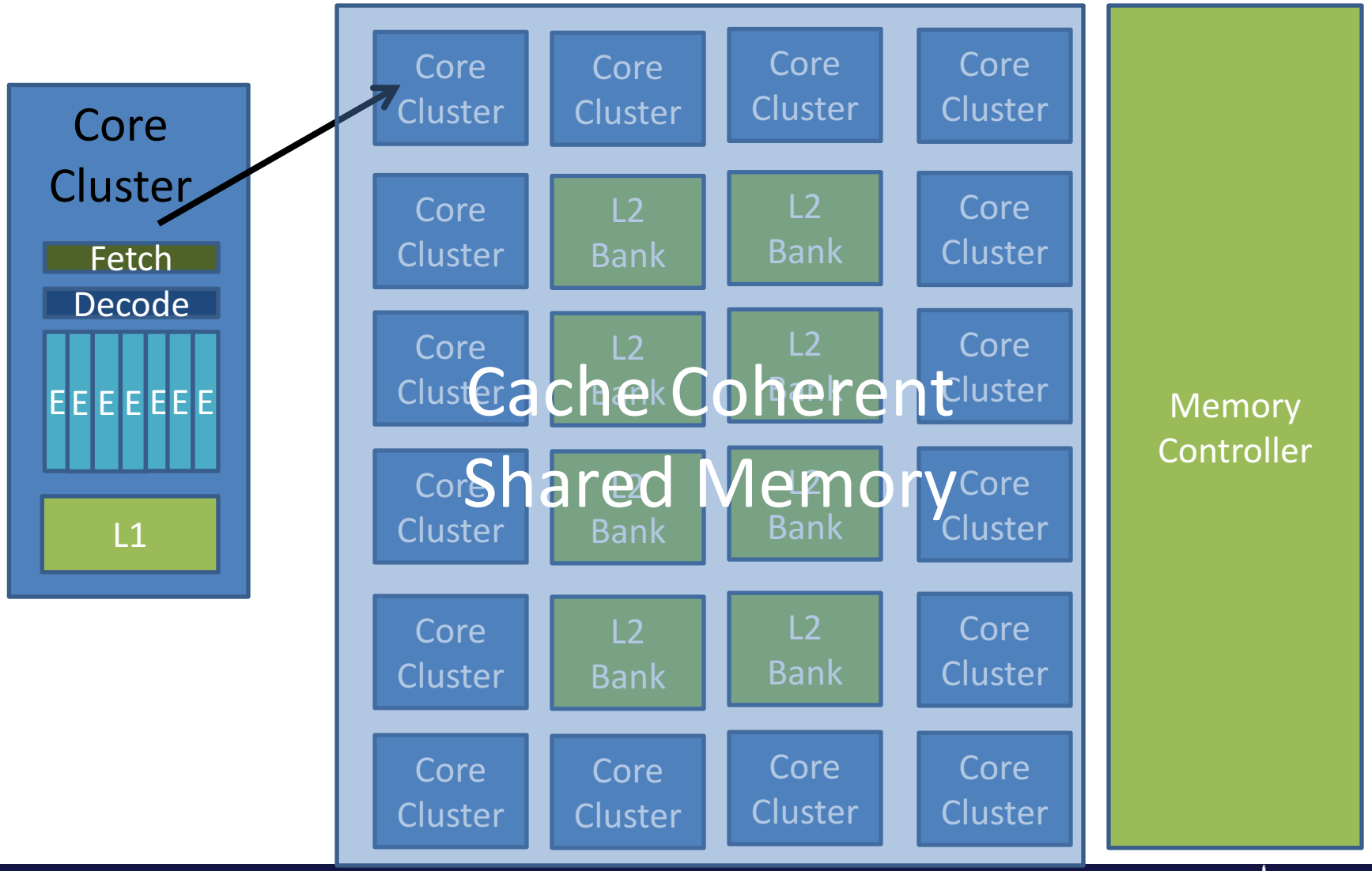
Overview

- **Massively Threaded Throughput-Oriented Processors** (MTTOPs) like GPUs are being integrated on chips with CPUs and being used for general purpose programming
- Conventional wisdom favors weak consistency on MTTOPs
- We implement a range of memory consistency models on MTTOPs
- We show that strong consistency is viable for MTTOPs

What is an MTTOP?

- **Massively Threaded** Throughput-Oriented
  - 4-16 core clusters
  - 8-64 threads wide SIMD
  - 64-128 deep SMT
  - ➢Thousands of concurrent threads
- Massively Threaded **Throughput-Oriented**
  - Sacrifice latency for throughput
    - Heavily banked caches and memories
    - Many cores, each of which is simple

Example MTTOP



(CPU) Memory Consistency Debate

	Strong Consistency	Weak Consistency
Performance	**Slower**	**Faster**
Programmability	**Easier**	**Harder**

- Conclusion for CPUs: trading off ~10-40% performance for programmability
  - "Is SC + ILP = RC?" (Gniady ISCA99)

But does this conclusion apply to MTTOPs?

Memory Consistency on MTTOPs

- GPUs have undocumented **hardware** consistency models
- Intel MIC uses x86-TSO for the full chip with directory cache coherence protocol
- MTTOP programming languages provide weak ordering guarantees
  - OpenCL does not guarantee store visibility without a barrier or kernel completion
  - CUDA includes a memory fence that can enable global store visibility

MTTOP Conventional Wisdom

- Highly parallel systems benefit from less ordering
  - Graphics doesn't need ordering
- Strong Consistency seems likely to limit MLP
- Strong Consistency likely to suffer extra latencies

Weak ordering helps CPUs, does it help MTTOPs?

It depends on how MTTOPs differ from CPUs …

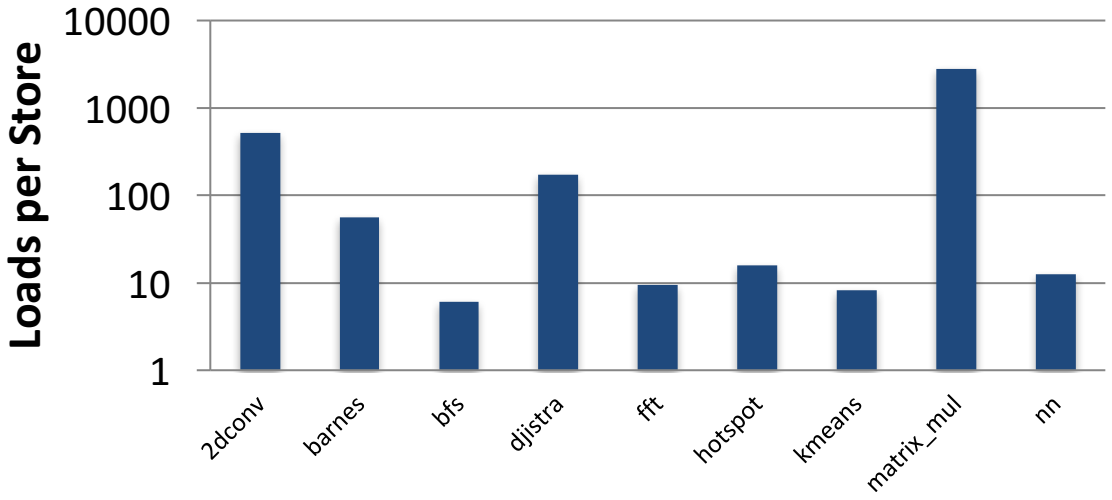
Diff 1: Ratio of Loads to Stores

Weak Consistency reduces the impact of store latency on performance

CPUs

Prior work shows CPUs perform 2-4 loads per store

MTTOPs



MTTOPs perform more loads per store→ store latency optimizations will not be as critical to MTTOP performance

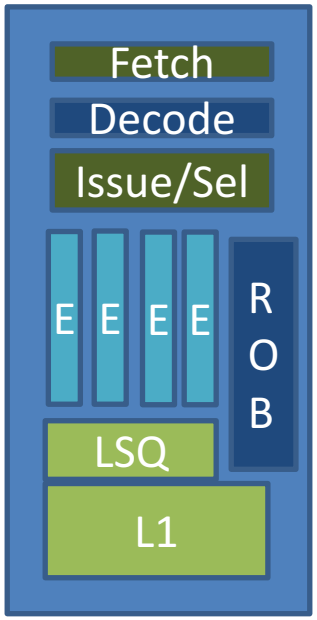
Diff 2: Outstanding L1 cache misses

Weak consistency enables more outstanding L1 misses per thread

CPU core

SIMD = 4
SMT = 4
MLP = 1-4
L1 Miss rate = .1

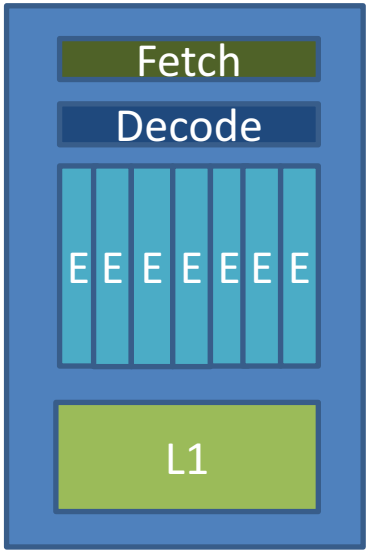
Misses = 1.6-6.4



MTTOP core cluster

SIMD = 64
SMT = 64
MLP = 1-4
L1 Miss rate = .5

Misses = 2048-8192

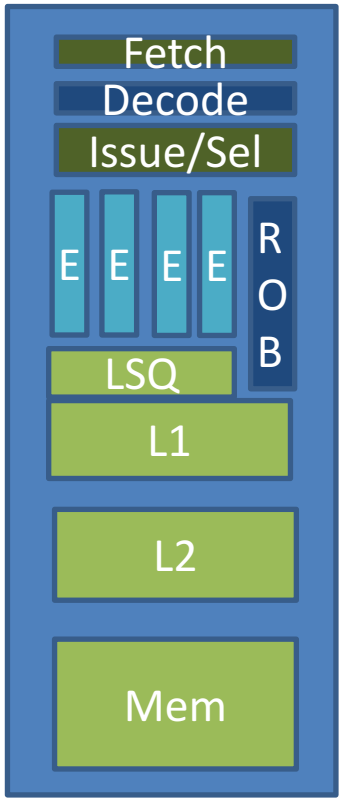


MTTOPs have more L1 cache misses → thread reordering enabled by weak consistency is less important to handle the latency of later memory stages

Diff 3: Memory System Latencies

Weak consistency enables reductions of store latencies

CPU core

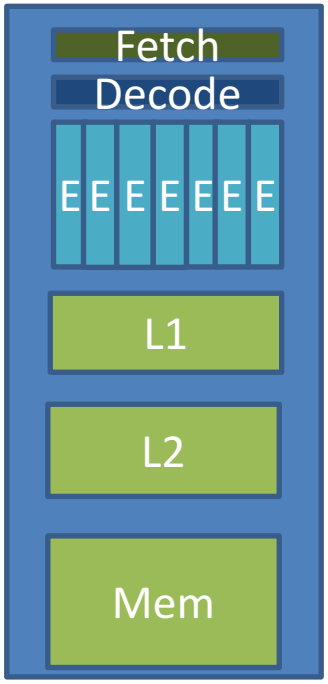


1-2 cycles

5-20 cycles

100-500 cycles

MTTOP core cluster



10-70 cycles

100-300 cycles

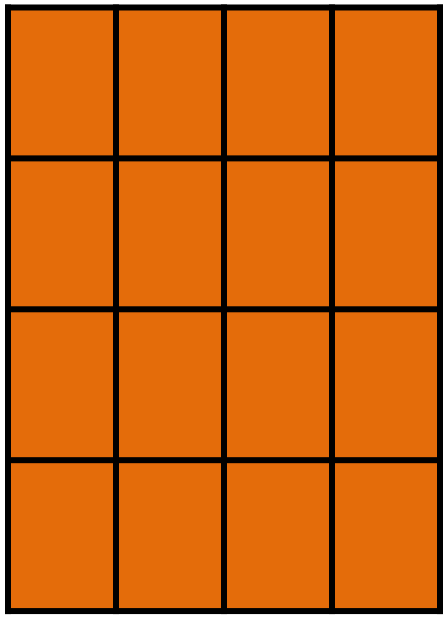
300-1000 cycles

MTTOPs have longer memory latencies → small latency savings will not significantly improve performance

Diff 4: Frequency of Synchronization

Weak consistency only re-orders memory operations between synchronization

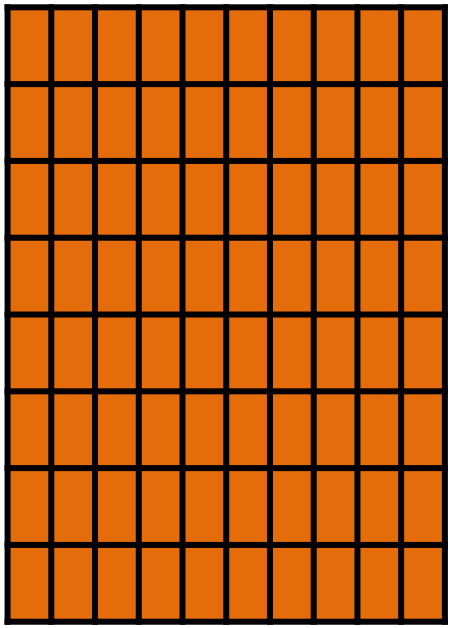
CPUs



```
spilt problem to regions

do:
      work on local region
      synchronize
```

MTTOPs



MTTOPs have more threads to compute a problem → each thread will have fewer independent memory operations between synchronization.

Diff 5: RAW Dependences Through Memory

Weak consistency enables store to load forwarding

CPUs

- Blocking for cache performance
- Frequent function calls
- Few architected registers
- ➢ Many RAW dependencies through memory

MTTOPs

- Coalescing for cache performance
- Inlined function calls
- Many architected registers
- ➢ Few RAW dependencies through memory

MTTOP algorithms have fewer RAW dependencies → there is little benefit to being able to read from a write buffer

MTTOP Differences & Their Impact

- Other differences are mentioned in the paper
- How much these differences affect performance of memory consistency?

Memory Consistency Implementations

No write buffer

Per-lane **FIFO** write buffer **drained on LOADS**

Per-lane **FIFO** write buffer **drained on FENCES**

Per-lane **CAM** for outstanding write addresses

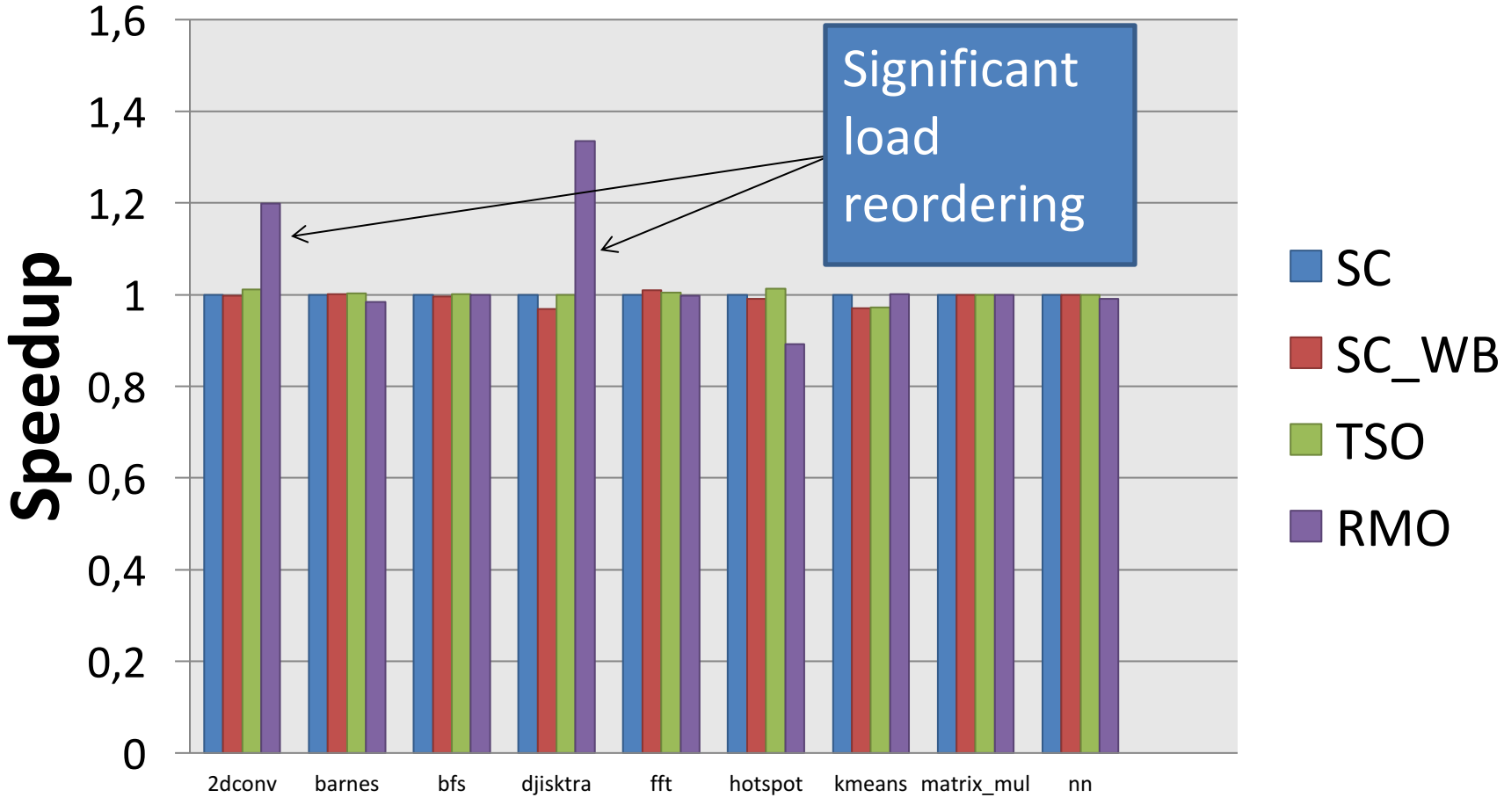
Methodology

- Modified gem5 to support SIMT cores running a modified version of the Alpha ISA
- Looked at typical MTTOP workloads
  - Had to port workloads to run in system model
- Ported Rodinia benchmarks
  - bfs, hotspot, kmeans, and nn
- Handwritten benchmarks
  - dijkstra, 2dconv, and matrix_mul

Target MTTOP System

Parameter	Value
core clusters	16 core clusters; 8 wide SIMD
core	in-order, Alpha-like ISA, 64 deep SMT
interconnection network	2D torus
coherence protocol	Writeback MOESI protocol
L1I cache (shared by cluster)	perfect, 1-cycle hit
L1D cache (shared by cluster)	16KB, 4-way, **20-cycle hit**, **no local memory**
L2 cache (shared by all clusters)	256KB, 8 banks, 8-way, **50-cycle hit**
consistency model-specific features (give benefit to weaker models)	
write buffer (SC_{wb} and TSO)	**perfect, instant access**
CAM for store address matching	**perfect, instant access**

Results


MTTOP Consistency Model Performance Comparison
Speedup
1,6
1,4
1,2
1
0,8
0,6
0,4
0,2
0
Significant load reordering
2dconv
barnes
bfs
djisktra
fft
hotspot
kmeans
matrix_mul
nn
SC
SC_WB
TSO
RMO

Upshot

- Improving store performance with write buffers is unnecessary
- MTTOP consistency model should not be dictated by performance or hardware overheads
- Graphics-like workloads can get significant MLP from load reordering (dijkstra, 2dconv)

Conventional wisdom may be wrong about MTTOPs

Outline

- Overview: Shared Memory & Coherence
- Intro to Memory Consistency
- Weak Consistency Models
- Case Study in Avoiding Consistency Problems
- Litmus Tests for Consistency
- Including Address Translation
- Consistency for Highly Threaded Cores