Chapter 10
Memory Consistency Models

Module 10.1 – Programmers' Expectation and Sequential Consistency Model

Cache Coherence vs. Memory Consistency

- Cache coherence
  - deals with ordering of writes to a **single** memory location
  - only needed for systems with caches
- Memory consistency
  - deals with ordering of reads/writes to **all** memory locations
  - concerns systems with or without caches

- Why is memory consistency model needed?

Programmer's Intuition

P0:	P1:
S1: datum = 5;	S3: while (!datumIsReady) ;
S2: datumIsReady = 1;	S4: … = datum

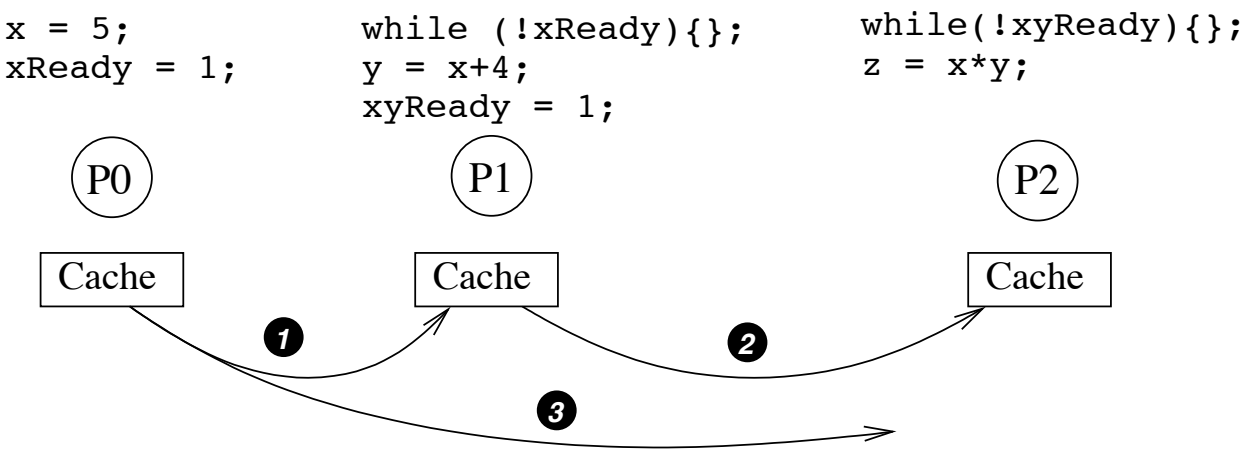
- Programmers expect that:
  - S4 reads the new value of datum (I.e., 5)
- Expectation violated if
  - S2 executed before S1
  - S4 executed before S3
- Hypothesis 1: **Program order expectation**
  - programmers expect that the order in which memory accesses are executed in a thread to follow the order in which they occur in the source code
- Note that the order of execution does not pertain to just one thread, but as seen by all other threads as well

Programmers' Intuition 2

P0:	P1:	P2:
S1: x = 5;	S3: while (!xReady) ;	S6: while (!xyReady) ;
S2: xReady = 1;	S4: y = x+4;	S7: z = x * y
	S5: xyReady = 1;	

- Let's say, initially, x=y=z=xReady=xyReady=0
- Programmers expect z to be 45 at S7
- Which implies that the new value of x:
  - if it has been propagated to P2
  - has also been propagated to P3
- Hypothesis 2: **Atomicity expectation**
  - An expectation that a read/write happens instantaneously with respect to all processors

Store Atomicity Violation



- How atomicity expectation is violated:
  - Step 1: New value of x and xReady propagated to P1, but has not reached P2
  - Step 2: New value of y and xyReady propagated to P2 before x is propagated to P2
  - Step 3: When x is propagated to P2, P2 has already read the old value of x, and z has been set to 0

Summary of Programmers' Expectation

- *Memory accesses coming out of a processor should be performed in program order, and each of them should be performed atomically*

- Lamport's definition of Sequential Consistency:
  - *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 occur in this sequence in the order specified by its program.*

- Empirically, programmers' expectations have been found to fit closely to Sequential Consistency (SC)

Figuring SC vs. Non-SC Outcome

P0:	P1:
S1: a = 1;	S3: … = b;
S2: b = 1;	S4: … = a;

- Note that program is **non-deterministic** due to a lack of synchronization
- In SC, S1 → S2 and S3 → S4 are guaranteed
  - S1, S2, S3, S4 cause a,b = 1,1
  - S3, S4, S1, S2 cause a,b = 0,0
  - S1, S3, S2, S4 cause a,b = 1,0
  - but a,b = 0,1 is impossible in SC
    - for a to be 0, it must be that S4 → S1: S3, S4, S1, S2
    - for b to be 1, it must be that S2 → S3: S1, S2, S3, S4
    - they cannot be both correct

Non-SC Machine

- The outcome of a,b = 0,1 is possible
  - S4, S1, S2, S3
    - In this case, S3 → S4 is violated
    - The order between two reads is violated
  - S2, S3, S4, S1
    - In this case, S1 → S2 is violated
    - The order between two writes is violated

Figuring SC vs. Non-SC Outcome

P0:	P1:
S1: a = 1;	S3: b = 1;
S2: … = b;	S4: … = a;

- Work on this:
  - Figure out what results are possible under SC
  - Figure out what results are not possible under SC
  - Prove that the impossible results can only occur when SC is violated

Answer: Possible and Impossible Results

- Note that program is non-deterministic due to a lack of synchronization
- In SC, S1 → S2 and S3 → S4 are guaranteed
  - S1, S2, S3, S4 cause a,b = 1,0
  - S3, S4, S1, S2 cause a,b = 0,1
  - S1, S3, S2, S4 cause a,b = 1,1
  - but a,b = 0,0 is impossible in SC
    - for a to be 0, it must be that S4 → S1: S3, S4, S1, S2
    - for b to be 0, it must be that S2 → S3: S1, S2, S3, S4
    - they cannot be both correct

For Non-SC Machine

- The outcome of a,b = 0,0 is possible
  - S4, S1, S2, S3
    - In this case, S3 → S4 is violated
    - The order between a write and a younger read is violated
  - S2, S3, S4, S1
    - In this case, S1 → S2 is violated
    - The order between a write and a younger read is violated

Non-deterministic program

- Both previous codes are non-deterministic
  - Not the common case, notoriously hard to debug
  - Non-determinism may have legitimate use (See Code 3.14 and 3.16 in Chapter 3)
- So, does preserving ordering of memory accesses matter?
- Probably no if non-determinism is intentional
- Otherwise, yes, because:
  - Programs may be non-deterministic by accidents (forgetting to insert synchronizations)
  - Keep programmers sane during debugging
  - Even properly synchronized programs need ordering for the synchronization to work properly
- How to build a system that ensures SC?

Building an SC System

- Program order
  - Ensure compiler does not reorder memory accesses
  - Declare critical variables as volatile (to avoid register allocation, code elimination, etc.)
- Atomicity
  - Execute one memory access one at a time, in program order
- In the processor pipeline, they can be overlapped/ reordered
  - But must be exposed to the cache in program order
  - A load is complete when the block has been read from the cache
  - A store is complete when invalidation has been posted (on a bus) or acknowledged (refer to Section 10.2.1 for more details)

Example of SC Ordering

```
S1: ld R1, A
S2: ld R2, B
S3: st R3, C
S4: st R4, D
S5: ld R5, D
```

- S1 must complete before S2, S2 before S3, etc.
- Implications
  - If S1 is a cache miss, S2 is a cache hit, S2 still must wait until S1 is completed. Same with S3 & S4
  - S5 must wait for S4 to complete, even though stores are often retired early and exposed to the cache late
  - S5 must wait for S4 to complete, even though they are to the same location!

Improving SC Performance - 1

- We still have to obey ordering,
- but we can make each load/store completion faster, e.g. by converting cache misses into cache hits:
  - Employ load prefetching
    - As soon as address is known/predictable, issue a prefetch request to fetch the block in Exclusive/Shared state
  - Employ store prefetching
    - As soon as address is known/predictable, issue a prefetch request to fetch the block in Modified state
- But not perfect
  - Prefetch too late: does not improve performance by much
  - Prefetch too early: cause pollution, block may be stolen, etc.

Improving SC Performance - 2

- Violate ordering, but undo effect if atomicity is violated
  - Ability to undo and re-execute is already present
    - (ILP technique - refer to ECE 521)
  - So, only need to know when atomicity has been violated
- Consider load A, followed by load B
  - In strict SC, load B must wait until load A completes
  - With speculation, load B accesses the cache anyway
  - Mark load B as speculative
  - If B is invalidated before it retires, atomicity has been violated
  - Then, cancel B and re-execute it
- Store speculation is harder (store cannot be canceled)
- Hence, only load speculation is employed

Module 10.2 – Relaxed Consistency Models

Relaxed Consistency Models

- Allows “some” memory accesses to be reordered or overlapped
- But, violating SC may violate programmers’ intuition
- So, provide safety nets
- Memory fence/memory barrier instructions
  - Fence instruction ensures ordering between preceding load/store and following load/store
  - Responsibility of programmers to insert fences
- Relaxed consistency models
  - Processor Consistency (PC)
  - Weak ordering (WO)
  - Release Consistency (RC)
  - Lazy Release Consistency (LRC)
  - etc.

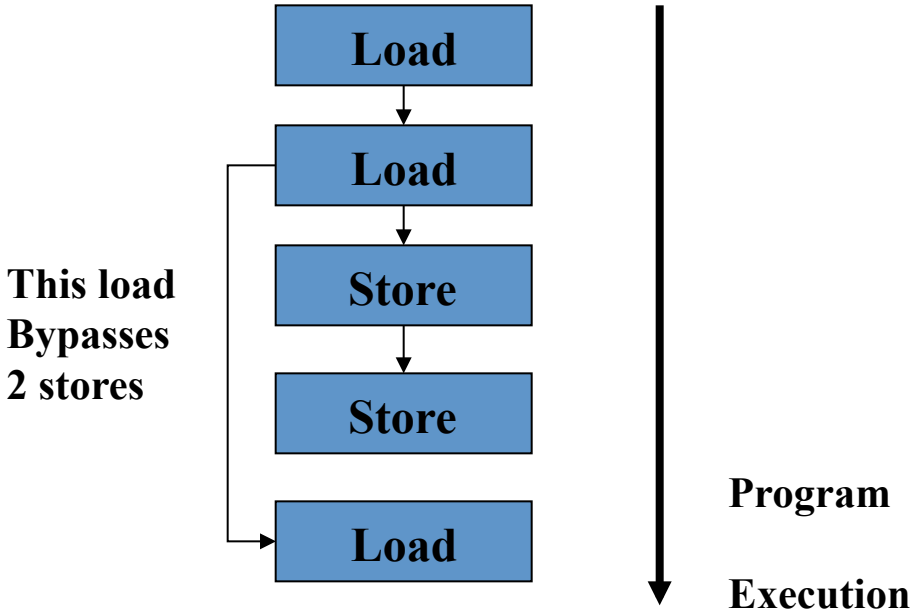
Implementation of Memory Fence

- Fence ensures that mem ops that are younger are not issued until the older mem ops have globally performed, I.e.
  - Wait until all older writes have been posted on the bus (or received InvAck)
  - Wait until all older reads have completed
  - Flush the pipeline to avoid issuing younger mem ops early
- Thread library or synchronization library programmers have to insert fences
- What if amateur programmers make their own synchronization, and forget fences?
  - Machine does not guarantee correctness

Processor Consistency

- SC ensures ordering of
  - LD ➔ LD
  - LD ➔ ST
  - ST ➔ LD
  - ST ➔ ST
- PC removes the ST➔LD constraint, with significant implications for ILP:
  - Stores can be placed in write buffer when committed, exposed to caches later
  - Loads do not wait for stores to perform, they access the cache right away (without being speculative!)
  - Load dependent on older store can “bypass” (directly obtain the store value)
- PC also removes write atomicity

Processor Consistency (PC)



Implications

P1:
data = 2000;
flag = 1;

P2:
while (flag == 0) {};
print data;

PC produces SC results

P1:
flag1 = 1;
if (flag2 == 0)
…

P2:
flag2 = 1;
if (flag1 == 0)
…

PC fails to produce SC results, but if programmers write such non-deterministic code, it's their fault, right? Hopefully

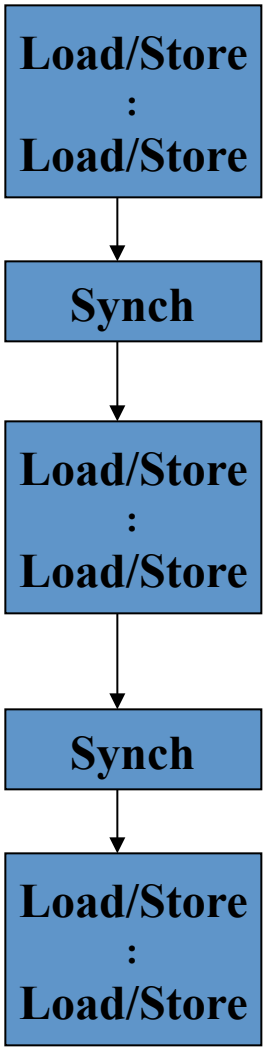
More

- How close is PC with programmers' expectation?
  - Most of the time, very close (e.g. post-wait synchronization works correctly)
  - Major OSes ported to PC with relative ease
- Others that cause errors in PC usually are due to races that also happen in SC
  - However, debugging races in PC is more difficult
- Non-atomic writes cause a problem (recall the 3-way synchronization
  - Against programmers' intuition

Weak Ordering (WO)

- Key observation:
  - If programmers want a particular ld/st to be executed after a different ld/st, they would have inserted synchronizations
- Most (bug-free) programs are synchronized
  - So, handle them correctly
  - And blame programmers for the remaining programs
- Key idea:
  - It is safe to reorder or overlap all memory operations except at synchronization points
- WO relies on 2 assumptions
  - Programs are properly synchronized
  - Programmers correctly express which ld/st act as synchronization accesses

Further Rationale for WO



Synch may be a lock acquire/release

Before a synch, all previous ops must finish
Before any ld/st, all previous synch must finish

Why safe? Typically within a critical section, we have made sure that only one process is inside, thus safe to reorder anything in the critical section.

Outside a critical section, we usually do not care about the order of mem ops (we would have used synchronization if we had cared)

Challenges for WO

- How to know whether a particular ld/st serves as a synchronization point?
  - Assume all atomic instructions are synchronization points
    - fetch-and-op, test-and-set
  - Assume all load linked (LL) and store conditional (SC) are synchronization points
- What if there are regular ld/st that are synchronization points?
- Expect programmers to
  - Insert safety nets, e.g. fence instructions
  - Label all ld/st as synchronization accesses
    - e.g., by converting the ld or st to atomic instructions

WO vs. PC

- More complex, need to distinguish normal ld/st with synch operations
- Ordering between read->write, read->read, and write->write not enforced
  - More overlap, higher performance
- In some cases, though, PC may outperform WC
  - E.g., a lock release is followed by a read

Release Consistency (RC)

- Distinguish synch operation into acquire and release
  - Acquire: acquires access to shared variable
    - Lock acquisition
    - Wait() in signal-wait synchronization
  - Release:
    - Lock release
    - Post() or Signal() in signal-wait synchronization

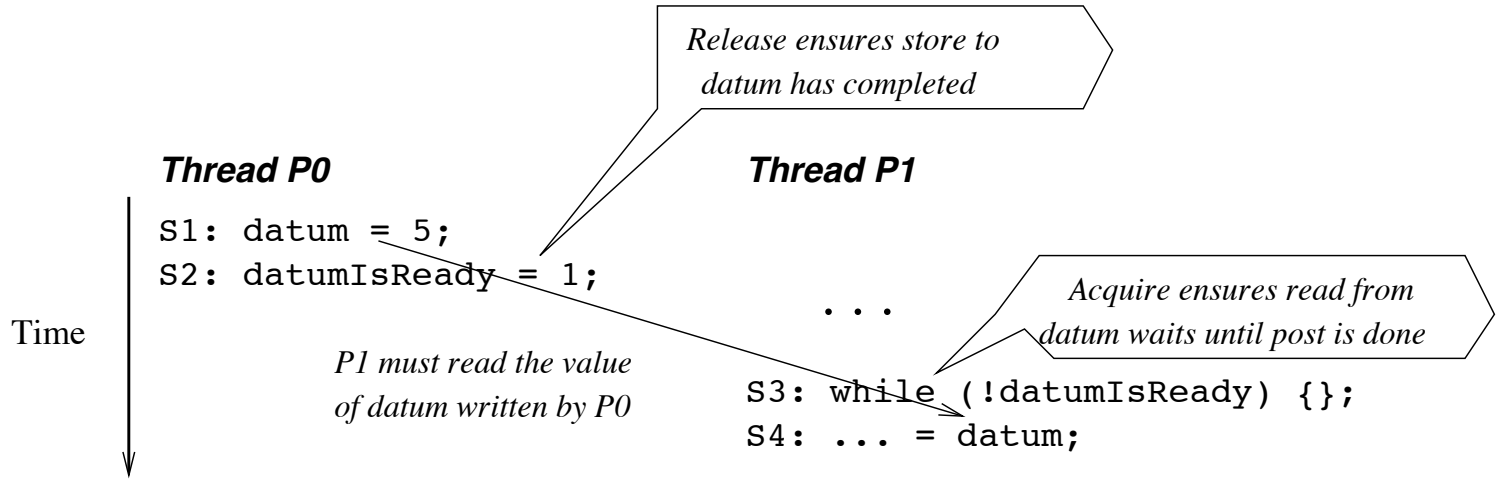
Acquire Semantics

- It ensures that no load/store instruction following sync_acquire is executed until the sync_acquire is complete. If we place it prior to entering the critical section and after obtaining the lock, we have made sure that no ld/st in the critical section is executed before the lock is fully obtained, and before preventing races with other processor inside the critical section. Hence, it prevents ***upward migration*** of instructions following the sync_acquire

Release Semantics

- It ensures that all ops before it have (globally) performed, hence when sync_release is executed before releasing the lock, we have made sure that the critical section's ld/st have been made visible to other processors. Hence, it prevents ***downward migration*** of instructions preceding the sync_release

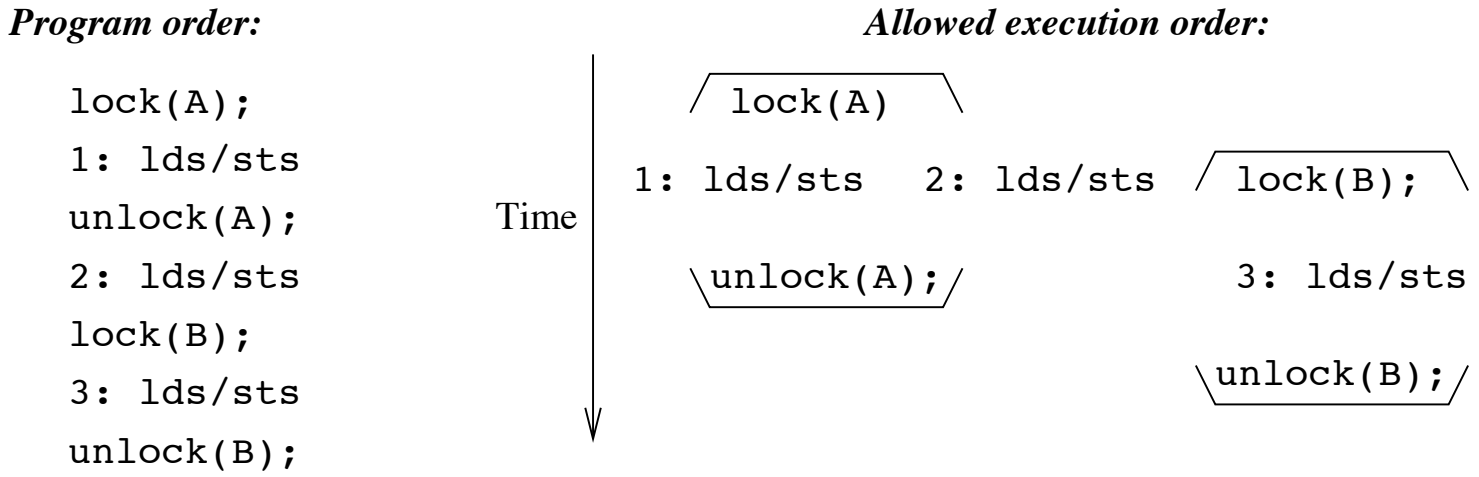
RC in Post-Wait Synchronization



- S2 is identified as release synchronization access
- Hence, S1 would have performed before S2 executes

- S3 identified as acquire synchronization access
- Hence, S3 would have performed before S4 executes

Overlapped Critical Sections in RC



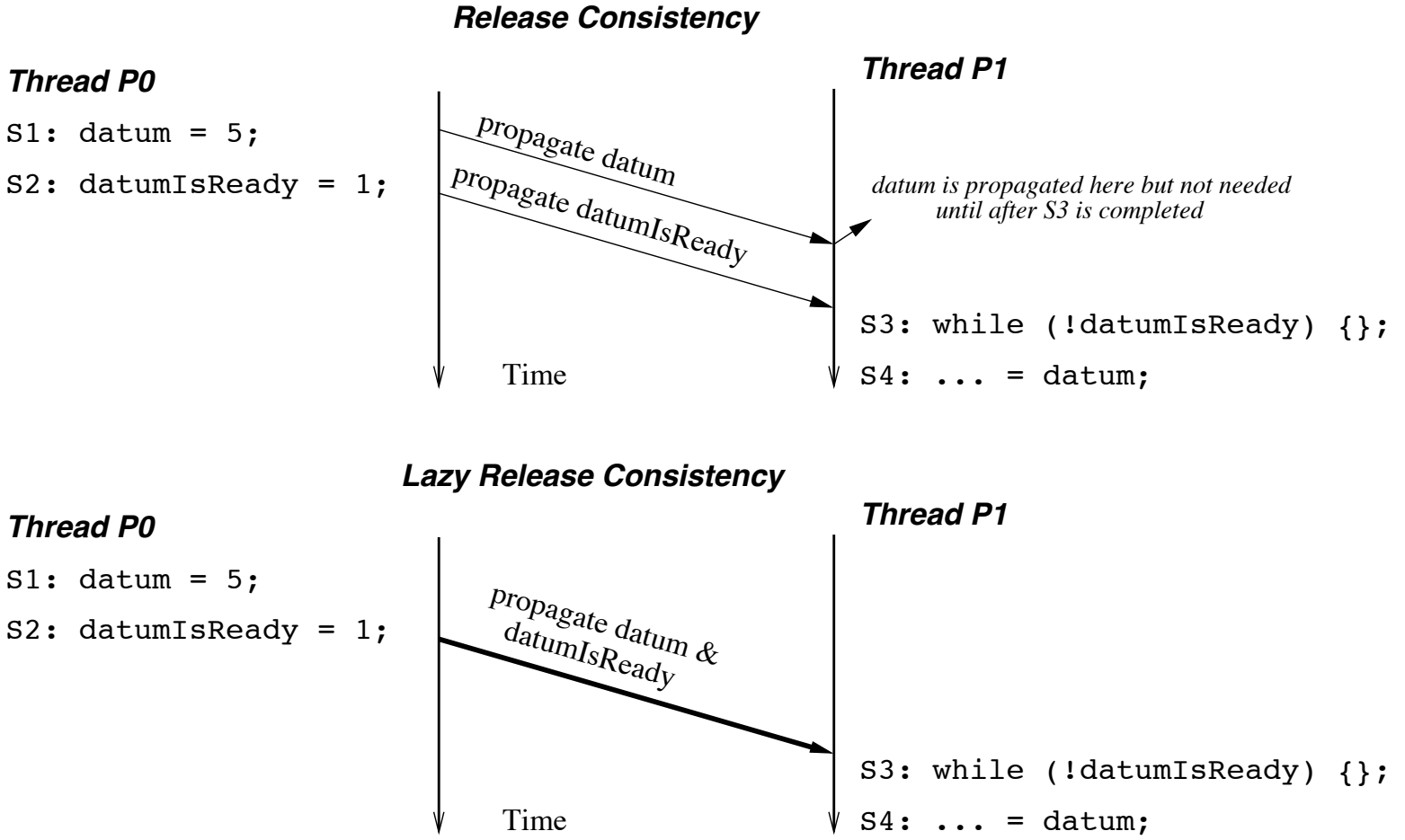
- Each of block 1, 2, 3, contains 1 or more loads or stores
- With WO,
  - only loads/stores in each block can be overlapped/reordered
  - loads/stores from different blocks cannot be overlapped/reordered
- With RC,
  - Block 1, 2, and 3 can be overlapped in their execution!
  - But none of them can be executed before lock(A) is performed

Lazy Release Consistency (LRC)

- Key observation
  - Values not needed until acquire synchronization is performed
- So why propagate them early?
  - We can propagate all new values when we hit the release sychronization points!

- This is where memory consistency issue mixes with cache coherence issue

RC vs. LRC (Lazy Release Consistency)



LRC

- Useful for systems in which many small write propagations is expensive compared to a few large write propagations
  - e.g. software shared memory multiprocessors

Implementations in Various ISAs

- See Section 10.4

An Alternative View

- Embed memory consistency model in the programming language (PL) specifications
- Compiler inserts fences as necessary to map the consistency model to what the hardware supports

- Was not done in early PL (C, C++, early Java)
- Very recently adopted in Java (see Java Memory Model paper by Pugh et al.)

- Pros: programmers get certainty
- Cons: compiler likely sacrifices performance for correctness

Example

Suppose you have a machine that does not guarantee any ordering with the exception of *fence* instruction that allows mem ops after it to execute only after mem ops before it globally complete. Insert fences in such a way to guarantee SC, PC, and WO/C

Original

Lock L1

Rd A

Rd B

Wr B

Wr D

Rd E

Unlock L1

Rd F

Example

Suppose you have a machine that does not guarantee any ordering with the exception of *fence* instruction that allows mem ops after it to execute only after mem ops before it globally complete. Insert fences in such a way to guarantee SC, PC, and WO/C

Original	**SC**	**PC**	**WO**
Lock L1	Lock L1	Lock L1	Lock L1
	fence	*fence*	*fence*
Rd A	Rd A	Rd A	Rd A
	fence	*fence*	
Rd B	Rd B	Rd B	Rd B
Wr B	Wr B	Wr B	Wr B
	fence	*fence*	
Wr D	Wr D	Wr D	Wr D
	fence		
Rd E	Rd E	Rd E	Rd E
	fence	*fence*	*fence*
Unlock L1	Unlock L1	Unlock L1	Unlock L1
	fence		*fence*
Rd F	Rd F	Rd F	Rd F

Example

Suppose you have a processor-consistent machine with a *fence* instruction. Insert fences in such a way to guarantee SC, PC, and WO/C

Original	**SC**	**PC**	**WO**
Lock L1	Lock L1	Lock L1	Lock L1
Rd A	Rd A	Rd A	Rd A
Rd B	Rd B	Rd B	Rd B
Wr B	Wr B	Wr B	Wr B
Wr D	Wr D	Wr D	Wr D
	fence		
Rd E	Rd E	Rd E	Rd E
Unlock L1	Unlock L1	Unlock L1	Unlock L1
	fence		*fence*
Rd F	Rd F	Rd F	Rd F

Latency Example

Given the following memory operations, determine their execution time assuming that each cache hit takes 1 clock cycle, cache miss takes 10 clock cycles, the cache is initially empty, and only <= 1 mem op is issued per cycle, on various processors that guarantee the following consistency models: SC, PC, and WO

Original	**SC**	**PC**	**WO**
Lock L1	Lock L1	Lock L1	Lock L1
	[0,10]	[0,10]	[0,10]
Rd A	Rd A	Rd A	Rd A
	[10,20]	[10,20]	[10,20]
Rd B	Rd B	Rd B	Rd B
	[20,30]	[20,30]	[11,21]
Wr B	Wr B	Wr B	Wr B
	[30,31]	[30,31]	[21,22]
Wr D	Wr D	Wr D	Wr D
	[31,41]	[31,41]	[12,22]
Rd E	Rd E	Rd E	Rd E
	[41,51]	[32,42]	[13,23]
Unlock L1	Unlock L1	Unlock L1	Unlock L1
	[51,52]	[42,43]	[23,24]
Rd F	Rd F	Rd F	Rd F
	[52,62]	[43,53]	[24,34]