

Lecture 13: Consistency Models

- Topics: sequential consistency, requirements to implement sequential consistency, relaxed consistency models

Coherence Vs. Consistency

- Recall that coherence guarantees (i) that a write will eventually be seen by other processors, and (ii) write serialization (all processors see writes to the same location in the same order)
- The consistency model defines the ordering of writes and reads to different memory locations – the hardware guarantees a certain consistency model and the programmer attempts to write correct programs with those assumptions

Example Programs

Initially, $A = B = 0$

P1

```
A = 1
if (B == 0)
  critical section
```

P2

```
B = 1
if (A == 0)
  critical section
```

P1

```
Data = 2000
Head = 1
```

P2

```
while (Head == 0)
  { }
... = Data
```

Initially, $A = B = 0$

P1

```
A = 1
```

P2

```
if (A == 1)
  B = 1
```

P3

```
if (B == 1)
  register = A
```

Consistency Example - I

- Consider a multiprocessor with bus-based snooping cache coherence and a write buffer between CPU and cache

Initially $A = B = 0$	
P1	P2
$A \leftarrow 1$	$B \leftarrow 1$
...	...
if ($B == 0$)	if ($A == 0$)
Crit.Section	Crit.Section

The programmer expected the above code to implement a lock – because of write buffering, both processors can enter the critical section

The consistency model lets the programmer know what assumptions they can make about the hardware's reordering capabilities

Consistency Example - 2

P1 Data = 2000 Head = 1	P2 while (Head == 0) { } ... = Data
--------------------------------------	--------------------------------------------------

Sequential consistency requires program order

- the write to Data has to complete before the write to Head can begin
- the read of Head has to complete before the read of Data can begin

Consistency Example - 3

P1	P2	P3	P4
A = 1	A = 2	while (B != 1) { }	while (B != 1) { }
B = 1	C = 1	while (C != 1) { }	while (C != 1) { }
		register1 = A	register2 = A

- register1 and register2 having different values is a violation of sequential consistency – possible if updates to A appear in different orders
- Cache coherence guarantees write serialization to a single memory location

Consistency Example - 4

Initially, $A = B = 0$

P1
 $A = 1$

P2
if ($A == 1$)
 $B = 1$

P3
if ($B == 1$)
 register = A

Sequential consistency can be had if a process makes sure that everyone has seen an update before that value is read – else, write atomicity is violated

Implementing Atomic Updates

- The above problem can be eliminated by not allowing a read to proceed unless all processors have seen the last update to that location
- Easy in an invalidate-based system: memory will not service the request unless it has received acks from all processors
- In an update-based system: a second set of messages is sent to all processors informing them that all acks have been received; reads cannot be serviced until the processor gets the second message

Sequential Consistency

- A multiprocessor is sequentially consistent if the result of the execution is achievable by maintaining program order within a processor and interleaving accesses by different processors in an arbitrary fashion
- The multiprocessors in the previous examples are not sequentially consistent
- Can implement sequential consistency by requiring the following: program order, write serialization, everyone has seen an update before a value is read – very intuitive for the programmer, but extremely slow

Performance Optimizations

- Program order is a major constraint – the following try to get around this constraint without violating seq. consistency
 - if a write has been stalled, prefetch the block in exclusive state to reduce traffic when the write happens
 - allow out-of-order reads with the facility to rollback if the ROB detects a violation
- Get rid of sequential consistency in the common case and employ relaxed consistency models – if one really needs sequential consistency in key areas, insert fence instructions between memory operations

Relaxed Consistency Models

- We want an intuitive programming model (such as sequential consistency) and we want high performance
- We care about data races and re-ordering constraints for some parts of the program and not for others – hence, we will relax some of the constraints for sequential consistency for most of the program, but enforce them for specific portions of the code
- Fence instructions are special instructions that require all previous memory accesses to complete before proceeding (sequential consistency)

Potential Relaxations

- Program Order: (all refer to *different* memory locations)
 - Write to Read program order
 - Write to Write program order
 - Read to Read and Read to Write program orders
- Write Atomicity: (refers to *same* memory location)
 - Read others' write early
- Write Atomicity and Program Order:
 - Read own write early

Relaxations

Relaxation	W → R Order	W → W Order	R → RW Order	Rd others' Wr early	Rd own Wr early
IBM 370	X				
TSO	X				X
PC	X			X	X
SC					X

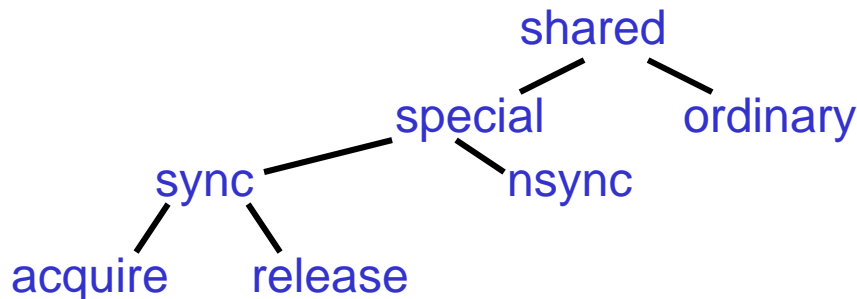
- IBM 370: a read can complete before an earlier write to a different address, but a read cannot return the value of a write unless all processors have seen the write
- SPARC V8 Total Store Ordering (TSO): a read can complete before an earlier write to a different address, but a read cannot return the value of a write by another processor unless all processors have seen the write (it returns the value of own write before others see it)
- Processor Consistency (PC): a read can complete before an earlier write (by any processor to any memory location) has been made visible to all

Safety Nets

- To explicitly enforce sequential consistency, safety nets or fence instructions can be used
- Note that read-modify-write operations can double up as fence instructions – replacing the read or write with a r-m-w effectively achieves sequential consistency – the read and write of the r-m-w can have no intervening operations and successive reads or successive writes must be ordered in some of the memory models

Release Consistency

- RCsc relaxes constraints similar to WO, while RCpc also allows reading others' writes early
- More distinctions among memory operations
 - RCsc maintains SC between special, while RCpc maintains PC between special ops
 - RCsc maintains orders: acquire \rightarrow all, all \rightarrow release, special \rightarrow special
 - RCpc maintains orders: acquire \rightarrow all, all \rightarrow release, special \rightarrow special, except for sp.wr followed by sp.rd



Performance Comparison

- Taken from Gharachorloo, Gupta, Hennessy, ASPLOS'91
- Studies three benchmark programs and three different architectures:
 - MP3D: 3-D particle simulator
 - LU: LU-decomposition for dense matrices
 - PTHOR: logic simulator
- LFC: aggressive; lockup-free caches, write buffer with bypassing
- RDBYP: only write buffer with bypassing
- BASIC: no write buffer, no lockup-free caches

Performance Comparison

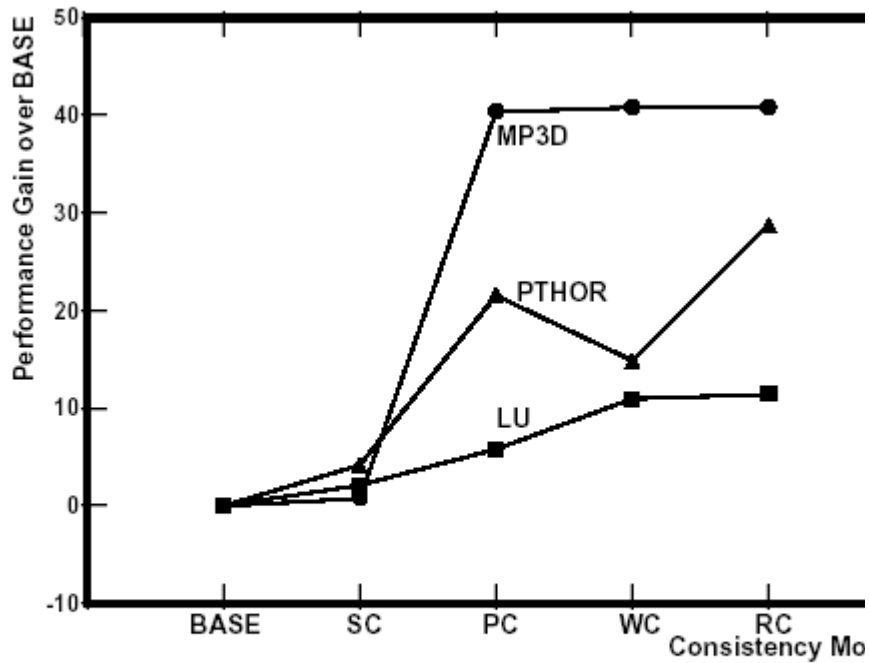


Figure 3: Relative performance of models on LFC

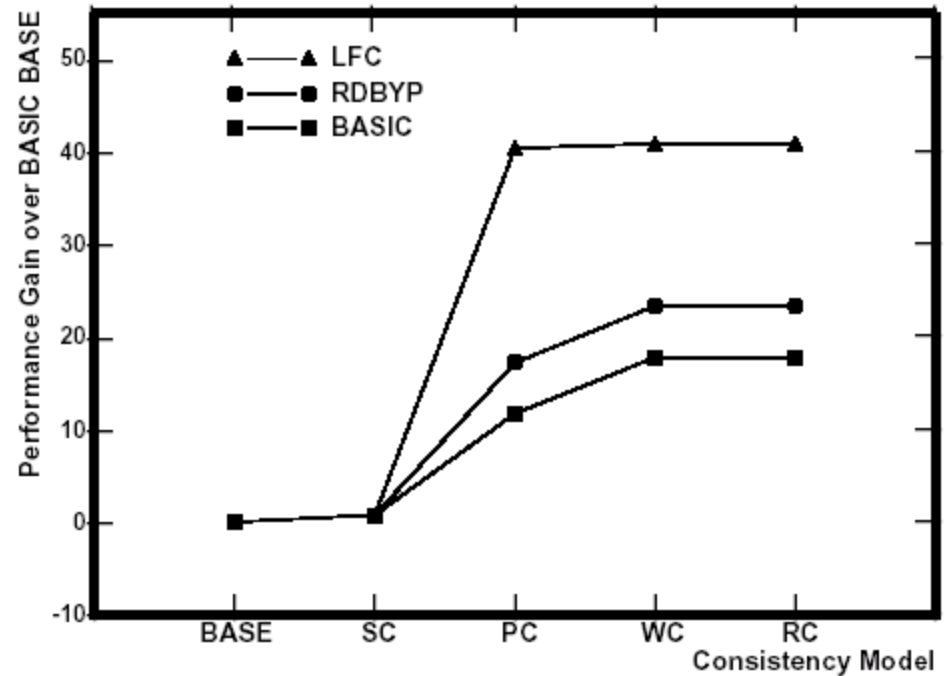


Figure 7: Performance of MP3D under LFC, RDBYP, and BASIC implementations.

Summary

- Sequential Consistency restricts performance (even more when memory and network latencies increase relative to processor speeds)
- Relaxed memory models relax different combinations of the five constraints for SC
- Most commercial systems are not sequentially consistent and rely on the programmer to insert appropriate fence instructions to provide the illusion of SC

Title

- Bullet