Shared Memory Consistency Models

Dimitris Nikolopoulos

Shared-memory abstraction

Programmers expect latest data written by P1

Several real systems would return old value of data!

```
    Initially all pointers = null, all integers = 0.

P1                                  P2, P3, ..., Pn

while (there are more tasks) {      while (MyTask == null) {
  Task = GetFromFreeList();           Begin Critical Section
  Task → Data = ...;                  if (Head != null) {
  insert Task in task queue             MyTask = Head;
}                                       Head = Head → Next;
Head = head of task queue;            }
                                      End Critical Section
                                    }
                                    ...  = MyTask → Data;
```

Memory consistency

- Formal specification of how updates of memory locations appear to the programmer
- Read returns value of last write
  - Easy in uniprocessors: program order
  - Harder on multiprocessors: what is the last write?
- Sequential consistency
  - All memory operations atomic (although they are not in reality)
  - All operations from the same processor in program order

Memory consistency model

- Enables programmers to reason about correctness in their programs
- Affects performance since it enables (or disables) several compiler and hardware optimizations
- Affects portability, since it defines what changes (or not) need to be performed to preserve correctness across platforms

Sequential consistency

- Close to programmer intuition
- Restricts many compiler and hardware optimizations
- Relaxed consistency models
  - Allow reordering of certain sequences of memory operations
  - Improve performance compared to sequential consistency

Uniprocessor consistency

- Memory operation ordering should preserve true data and control dependencies
- Enables optimizations such as:
  - Register allocation
  - Code motion
  - Loop transformations
  - Pipelining, multiple instruction issue
  - Write buffer bypassing, forwarding
  - Lockup-free (non-blocking) caches

Sequential consistency view

Memory seen as a global memory with a single switch enabling access to processors

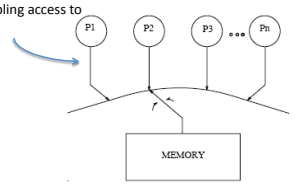


Figure 3: Programmer's view of sequential consistency.

Example: Dekker's Algorithm

If P1 sees Flag2=0, this implies that the update of Flag1 has happened before the update of Flag2

Sequential consistency guarantees that both processors can not see both Flags equal to 0 at the same time

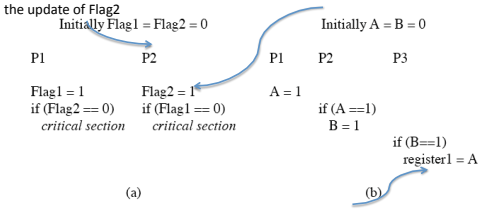


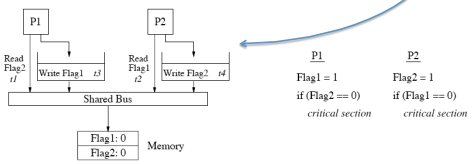
Figure 4: Examples for sequential consistency.

If p3 sees P2's write, then it also sees P1's write.

Violations of SC (no caches)

- Write buffers
  - Reads can bypass pending writes

Need to block reads while writes are pending. Writes can wait for an acknowledgement from the destination

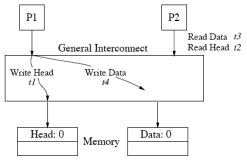


P1	P2
Flag1 = 1	Flag2 = 1
if (Flag2 == 0)	if (Flag1 == 0)
critical section	*critical section*

(a) write buffer

Violations of SC (no caches)

- Bypassing in interconnection network

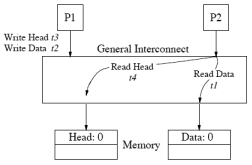


P1	P2
Data = 2000	while (Head == 0) {;}
Head = 1	... = Data

(b) overlapped writes

Violations of SC (no caches)

- Non-blocking reads



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

(c) non–blocking reads

Figure 5: Canonical optimizations that may violate sequential consistency.

Violation of SC (with caches)

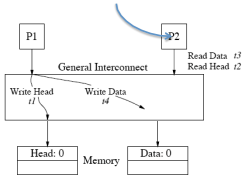
- Similar violations with systems without caches
- Read by a processor that hits in the cache
  - May not allow completion to prevent bypassing!
- Other issues
  - Cache coherence protocol required to propagate writes
  - Detecting when a write is complete requires more transactions in the presence of caching
  - Propagating changes to multiple copies of data is inherently non-atomic

Cache consistency = SC?

- Cache consistency requires that a write is *eventually* seen by all processors
- Cache consistency requires that writes by all processors to the *same memory location* are seen in the same order by all processors
- Sequential consistency requires that writes to *all memory locations* are seen in the same order by all processors

Detecting completion of writes

Assume P2 has "Data" in its cache. P1 writes head before write to data is propagated to P2 (memory commit happens before invalidation)



(b) overlapped writes

Possible for P2 to read the new value of Head, but the old value of Data (from its cache), violation of SC!

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

Need a mechanism to confirm completion of invalidations before proceeding with the second write in P1!

Illusion of write atomicity

Writes to A may arrive out of order to P3 and P4, therefore violating sequential consistency

May happen in interconnection networks with no guarantees in the order of delivery

Initially A = B = C = 0

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

Figure 6: Example for serialization of writes.

Need to ensure that writes to the same memory location are strictly serialized (write atomicity)

Illusion of write atomicity

All variables in caches, all memory operations by the same processor in order and atomic

P2 reads A before A's update reaches P3, B's update from P2 reaches P3, P3 sees old value of A (from cache).

Initially Flag1 = Flag2 = 0

P1	P2
Flag1 = 1	Flag2 = 1
if (Flag2 == 0)	if (Flag1 == 0)
critical section	*critical section*

(a)

Initially A = B = 0

P1	P2	P3
A = 1		
	if (A ==1)	
	B = 1	
		if (B==1)
		register1 = A

(b)

Figure 4: Examples for sequential consistency.

Violation of SC because P2 sees and uses the write to A before the write is propagated to P3

Illusion of write atomicity

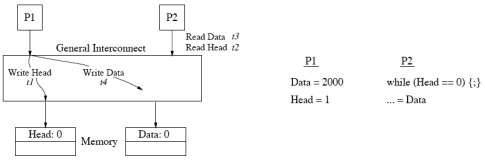
- Prevent read to return a newly written value before all invalidations have been acknowledged
- Easy in invalidate-based protocols
- Harder in update-based protocols
  - Two-phase update
  - Send updates to all caching processors
  - Receive acknowledgment
  - Send acknowledgement to all caching processors

Similarity between compilers and hardware

- Consistency prevents reordering of instructions
- Register allocation can be hazardous (equivalent to caching)
- Software pipelining can be hazardous (recall midterm!)
- Code motion can be hazardous (recall dynamic scheduling processors)

Compiler example: Register allocation

Register allocation of Head by P2 may prevent P2 from seeing the update From P1. Sequential consistency violation!



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

(b) overlapped writes

SC explained

- Program order
  - A processor must ensure that a memory operation is complete before proceeding to the next. Detecting completion for writes requires ack from memory and invalidations or updates in cache-based systems
- Write atomicity
  - Writes to a single location are serialized, no read returns a new value of a write until all invalidations/ updates are finished and acknowledged

Hardware optimizations for SC

- Prefetching ownership on a write
  - Prefetch exclusive requests for any writes pending in the write buffer
  - Applicable only in invalidation-based systems
- Speculative service of reads
  - Rollback if SC is violated due to a write
  - Good for superscalar processors that already have rollback machinery (e.g. to handle branch mispredictions)

Relaxed consistency

- Characterization
  - How do we relax program order?
    - From write to read
    - From write to write
    - From read to read or write
  - How do we relax write atomicity?
    - Do we allow reads to return another processor's write before all updates/invalidations are sent?
    - Processor allowed to see its own write before the write is made visible to other processors
- Mechanisms for overriding relaxations to programmers

RC summary

Relaxation	W — R Order	W — W Order	R — RW Order	Read Others' Write Early	Read Own Write Early	Safety net
SC [16]					√	
IBM 370 [14]	√					serialization instructions
TSO [20]	√				√	RMW
PC [13, 12]	√			√	√	RMW
PSO [20]	√	√			√	RMW, STBAR
WO [5]	√	√	√		√	synchronization
RCsc [13, 12]	√	√	√		√	release, acquire, nsync, RMW
RCpc [13, 12]	√	√	√	√	√	release, acquire, nsync, RMW
Alpha [19]	√	√	√		√	MB, WMB
RMO [21]	√	√	√		√	various MEMBAR's
PowerPC [17, 4]	√	√	√	√	√	SYNC

Figure 8: Simple categorization of relaxed models. A √ indicates that the corresponding relaxation is allowed by straightforward implementations of the corresponding model. It also indicates that the relaxation can be detected by the programmer (by affecting the results of the program) except for the following cases. The "Read Own Write Early" relaxation is not detectable with the SC, WO, Alpha, and PowerPC models. The "Read Others' Write Early" relaxation is possible and detectable with complex implementations of RCsc.

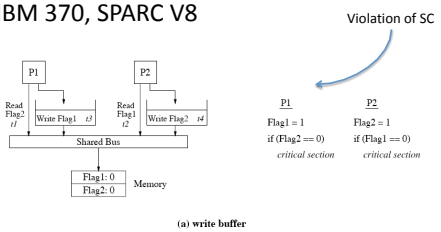
RC in commercial systems

Relaxation	Example Commercial Systems Providing the Relaxation
W — R Order	AlphaServer 8200/8400, Cray T3D, Sequent Balance, SparcCenter1000/2000
W — W Order	AlphaServer 8200/8400, Cray T3D
R — RW Order	AlphaServer 8200/8400, Cray T3D
Read Others' Write Early	Cray T3D
Read Own Write Early	AlphaServer 8200/8400, Cray T3D, SparcCenter1000/2000

Figure 9: Some commercial systems that relax sequential consistency.

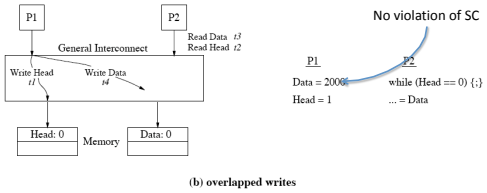
Relaxing Write-to-Read

- Relax ordering of write followed by read to *different* memory location (performance)
- IBM 370, SPARC V8



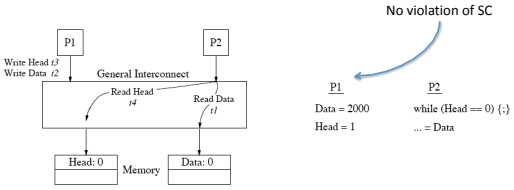
(a) write buffer

Relaxing Write-to-Read



(b) overlapped writes

Relaxing Write-to-Read



(c) non–blocking reads

Figure 5: Canonical optimizations that may violate sequential consistency.

Relaxing W➔R implementations

- IBM 370
  - Prohibits a read to return before a write to the same location is visible to all processors
- TSO
  - Allows a read to return before a write from the same processor, before the write from the processor becomes visible to other processors
- PC
  - Relaxes both conditions on read

Example

Initially A = Flag1 = Flag2 = 0

P1	P2
Flag1 = 1	Flag2 = 1
A = 1	A = 2
register1 = A	register3 = A
register2 = Flag2	register4 = Flag1

Result: register1 = 1, register3 = 2, register2 = register4 = 0

(a)

Initially A = B = 0

P1	P2	P3
A = 1		
	if (A == 1)	
	B = 1	
		if (B == 1)
		register1 = A

Result: B = 1, register1 = 0

(b)

Figure 10: Differences between 370, TSO, and PC. The result for the program in part (a) is possible with TSO and PC because both models allow the reads of the flags to occur before the writes of the flags on each processor. The result is not possible with IBM 370 because the read of A on each processor is not issued until the write of A on that processor is done. Consequently, the read of the flag on each processor is not issued until the write of the flag on that processor is done. The program in part (b) is the same as in Figure 4(b). The result shown is possible with PC because it allows P2 to return the value of P1's write before the write is visible to P3. The result is not possible with IBM 370 or TSO.

Safety nets

- IBM 370, none
- TSO
  - Safety net required from write to read to the same location on the same processor
  - Solved with an atomic RMW (read-modify-write) instruction (at extra cost, can be significant)

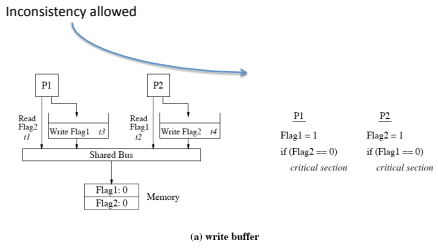
Performance implications

- Write-to-read relaxation improves substantially the performance of hardware implementations
- Little benefit for compilers, typically because writes are closely followed by reads to the same memory location, therefore no huge opportunities for reorderings
- Compilers typically benefit from full independence in large windows of instructions

Write-to-write, write-to-read

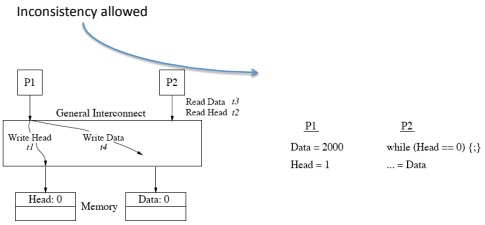
- Eliminate ordering between writes to *different* locations
- SPARV V8, PSO (partial store ordering)
  - Writes from the same processor can be pipelined, overlapped and reach destination caches out-of-order

Write-to-write relaxation



(a) write buffer

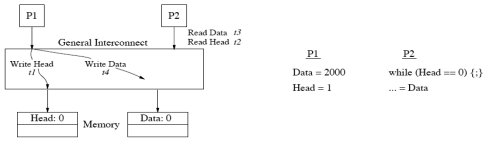
Write-to-write relaxation



(b) overlapped writes

Write-to-write relaxation

- Safety net
  - Store barrier
  - Inserted in the write buffer
  - Delays writes following the STBAR in the write buffer



(b) overlapped writes

Relaxing all orders

- Any read or write may be reordered with respect to any following read or write
- Memory operations following a read may be reordered with respect to the read (including writes)
- Hardware can hide latency of read operations (true non-blocking read implementation)
- All relaxed models allow processor to read its write early

Weak Ordering

- Data operations distinguished from synchronization operations
- To enforce ordering between two operations programmer needs to specify at least one as a synchronization operation
- Intuition: reordering operations outside synchronization regions typically does not affect performance (think mutual exclusion)

Weak Ordering

- Hardware implementation using counters
  - Processor increments counter when issuing operation, decrements when operation completes
  - Synchronization operation not issued until counter reaches zero (all operations complete)
  - Processor does not issue operations until after synchronization operation completes
  - Synchronization instructions serve as "fences" or "barriers"

Release Consistency

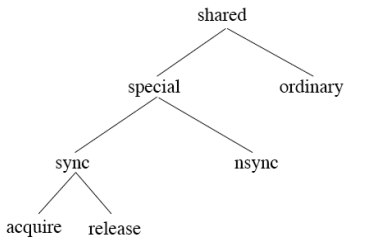


Figure 11: Distinguishing operations for release consistency.

Release Consistency

- *Ordinary* and *Special* operations: roughly correspond to data and synchronization operations in WO
- Special operations are either *sync* or *nsync*
  - Sync operations correspond to "real" synchronizations
  - "nsync" operations correspond to asynchronous operations that may be subject to races
  - Sync operations are further classified into *acquire* and *release*

Release Consistency

- *Ordinary* and *Special* operations: roughly correspond to data and synchronization operations in WO
- Special operations are either *sync* or *nsync*
  - Sync operations correspond to "real" synchronizations
  - "nsync" operations correspond to asynchronous operations that may be subject to races
  - Sync operations are further classified into *acquire* and *release*
    - Acquire and release correspond to locks used to grant exclusive access to a set of shared memory locations

Release consistency

- First implementation (RCsc): sequential consistency between special operations
- Second implementation (RCpc): processor consistency between special operations
- RCsc constraints:
  - Acquire to all, all to release, special to special
- RCpc constraints:
  - Acquire to all, all to release, special to special except from special write followed by special read

RC: imposing program order

- From a write to a read, we use an RMW instruction
- Write in RMW needs to be a release if the write being ordered is ordinary, otherwise it can be any special write
- Other primitives: write barrier, memory barrier