Module 15: "Memory Consistency Models"

Lecture 35: "Release Consistency and Delayed Consistency"

Memory Consistency Models

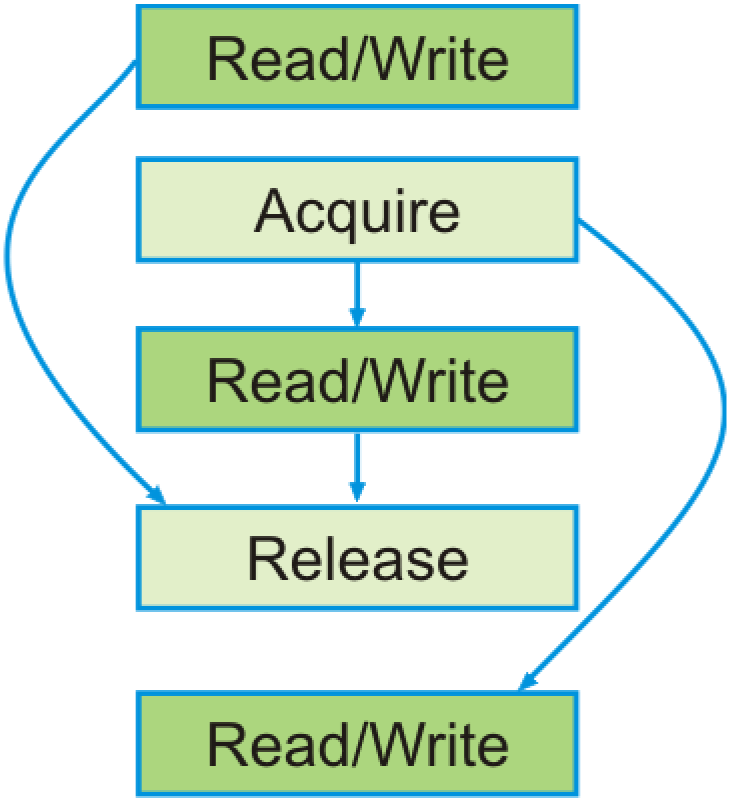
- Release consistency
- Hardware support
- Industry situation
- Eager-exclusive reply
- Compilers' job
- Delayed consistency
- Conclusions

[From Chapters 9 and 11 of Culler, Singh, Gupta]
[Additional reading: Adve and Gharachorloo , WRL Tech Report, 1995]

Previous Next

Release consistency

- Relaxes WO even further
    - Categorize synchronization operations into acquire and release
    - Acquire is a read (e.g., a normal load) or an atomic read-modify-write (e.g., LL/SC pair) used to get access to certain set of protected variables; popular examples of acquire include LOCK operations and waiting on a flag
    - Release is a write (e.g., a normal store) or an atomic read-modify-write (e.g., LL/SC pair) used to grant access to certain set of protected variables to others; popular examples of release include UNLOCK operations and setting a flag
    - Barrier is release (arrival) as well as acquire (departure)
- Couple of observations
    - There is no harm in issuing the acquire operation and the instructions after it, and committing them before all instructions preceding acquire have committed
    - There is no harm in executing and committing the instructions after release even before the release is committed
    - Of course, such re-ordering applies to only those instructions that access different memory addresses
    - Must not issue any instruction following an acquire before the acquire commits (semantics of acquire must be maintained)
    - Must commit all instructions ahead of release before release can commit (at this point all new values become visible to others)



- Synchronization operations must be labeled properly
- Otherwise compiler must insert fence instructions at acquire and release boundaries (may lose some of the benefits of RC)

- Note that for barrier synchronization, RC does not offer any extra advantage over WO

Hardware support

- WO/RC on Alpha processors
    - The processor does not implement load-invalidate replay; not needed for WO or RC (but WO is not fully exploited)
    - Offers two fence instructions: memory barrier (MB) and write memory barrier (WMB)
    - MB is same as sync in R10000 i.e. disables issue of any further memory operations until all memory operations before MB have committed
    - WMB only enforces ordering among stores like stbar in SPARC i.e. a store after WMB cannot bypass it, but a load can
    - MB can be used to easily implement WO: insert MB at every synchronization operation
- Relaxed memory order (RMO) in SPARC v9
    - Offers four flavors of fence instructions (RR, RW, WR, WW): possible to synthesize an array of consistency models

Industry situation

- A fairly debated issue
    - Chip designers naturally want to make hardware simple and that points to relaxed models
    - Processors designed by MIPS Technology implement SC
    - Processors from Sun Microsystems support TSO and/or PSO
    - Intel processors come with PC
    - Alpha and IBM PowerPC processors support WO; Power4, Power5 do not guarantee write atomicity
- Multiprocessors normally follow the model supported by the underlying microprocessor
    - Pentium Pro Quad SMP allows writes and subsequent interventions to complete even before all invalidation acknowledgments are collected (violates write atomicity)
- Delayed-exclusive replies in Origin 2000
    - In Origin 2000 the memory controller at the requester node does not send the upgrade ack or PUTX reply to the local processor until all inval acks are collected
    - Note that memory controller actually could fool the processor by sending the exclusive reply as soon as it arrives; this is called eager-exclusive reply and is exercised by most Alpha servers (they can do it because they are not SC)
    - Eager-exclusive reply complicates memory controller design: must hold on to the OTT entry until all inval acks are collected; must block subsequent interventions from proceeding and must block writeback for that line until all inval acks are collected (relaxing these two would violate write atomicity)

Previous Next

Eager-exclusive reply

- Programmers' interface
    - P0: A=1; flag=1; P1: while (!flag); print A;
    - Programmer thinks A will be printed as 1
    - A machine that implements eager-exclusive replies can produce a value of 0 (even if the processor is SC); how?
    - P0: A=1; WMB; flag=1; P1: while (!flag); print A;
    - Formally, at every release boundary (e.g., before UNLOCK, BARRIER, flag set) a WMB must be inserted; WMB will wait for all pending inval acks to be collected before allowing any further issue from store queue
    - Eager-exclusive replies essentially require the same treatment at the programmers' interface as if RC allowed only write re-ordering (is it similar to PSO? No)
- The write atomicity is anyway broken
    - P0: A=1; flag=1; P1: while(!flag); print A;
    - Why bother to complicate OTT by holding back writebacks and interventions?

Compilers' job

- Must be careful while carrying out code motion
    - Usually it is very hard to formally argue about correctness of a re-ordering
    - The compiled code must not follow a model that is weaker than whatever is advertised to the programmer
    - The underlying microprocessor normally supports a model that is at least as strong as whatever the compiler offers; it could be stronger, but cannot be weaker (that would be unfaithful to the programmer)
    - A piece of program compiled for RC will still work fine on an SC microprocessor, but will fail to exploit some of the optimizations done by the compiler (here working fine means it will produce an RC-compliant output and that is the programmers' interface)
    - PC, PSO, TSO do not offer much opportunity for compiler optimization
- Synchronized or properly labeled programs
    - A program that labels all synchronization operations with some directives is called a synchronized program
    - For locks and barriers it is easy to do; how to identify all event synchronizations through flags?
        - Often the programmer can label these manually
    - Formally, it is necessary to label all competing operations; two conflicting operations (i.e. to the same location and one is a write) from two different processes are said to be competing if they appear back-to-back in at least one possible SC execution
    - A program is synchronized if all competing operations are labeled as synchronization operations
    - Once this (hard part) is done, compiler inserts fence instructions at proper places depending on how relaxed the model is
- Current state of automatically producing synchronized programs is quite sad
    - Requires enumerating all SC-compliant re-orderings
    - The existing analysis is either conservative or very expensive
    - So normally it becomes programmers' job
    - One extreme would be to label all operations as competing; this will lead to

performance poorer than SC (why?)

- Sometimes a programmer may choose to leave some competing operations unlabeled (i.e. intentionally introduce data races); leads to better performance if either it does not compromise correctness (due to some peculiar program behavior) or it is okay to have non-deterministic outcome (e.g., red-black ordering)

Previous Next

Delayed consistency

- Designed especially for update-based protocols
  - Delay the updates until write buffer becomes full or until the next synchronization point or some other execution point
  - Aims at reducing update traffic
  - Possible to merge many updates to the same cache line
  - Reasoning about program may become difficult
  - Same thing can be done for invalidation-based protocols also:
    - Keep stores in the store buffer (need a large one) until the store buffer fills up or the next synchronization point is reached; at this point issue these stores to the memory system (i.e. send to home, send invalidations, etc.); TSO, PSO, PC, WO, RC can take advantage of this (invalidation traffic becomes bursty; but false sharing is reduced) [Transactional consistency]

Conclusions

- Why consistency models?
  - Server designers don't deal with these notorious issues just for fun
  - The problem is that some people feel SC is too restrictive; relaxed models are needed to get good performance possibly compromising some simplicity on the programmers' side (of course, you should not invent a consistency model that is so unintuitive that it becomes unusable)
  - Pipelining with sufficient buffering (to support many in-flight instructions) and with some extra "fix-up" hardware, SC can provide fairly good performance
  - Store buffers provide some write latency overlap in SC microprocessors, but the ROB size is pathetically small to overlap the entire invalidation sending and acknowledgment collection phases; TSO, PC, PSO offer better choices
  - TSO, PC, PSO allow stores to be removed from the head of the ROB as soon as they issue, thereby unclogging the ROB and allowing subsequent instructions (including loads) to commit i.e. loads can bypass stores
  - In TSO and PC, a remote store at the head of store buffer may not allow subsequent stores (which may be local and even cache hits) to leave; PSO solves this problem by allowing stores to bypass previous stores
  - Soon designers realized that loads are even bigger problem because they are more frequent than stores on RISC processors; so the capability to bypass loads (and stores) was formalized in RC and WO (taking full advantage of this in hardware is very difficult, though)
  - Surprisingly, all microprocessors today implement all these re-orderings inside the pipeline; then why not take a little more pain and implement SC

Previous Next

Exercise : 3

These problems should be tried after module 15 is completed.

1. [5+5] Consider a 512-node system. Each node has 4 GB of main memory. The cache block size is 128 bytes. What is the total directory memory size for (a) bitvector, (b) DiriB with i=3?

2. [5+5+5+10] For a simple two-processor NUMA system, the number of cache misses to three virtual pages X, Y, Z is as follows.

Page X: P0 has 14 misses, P1 has 11 misses. P1 takes the first miss.
Page Y: P0 has zero misses, P1 has 18 misses.
Page Z: P0 has 15 misses, P1 has 9 misses. P0 takes the first miss.

The remote to local miss latency ratio is four. Evaluate the aggregate time spent in misses by the two processors in each of the following policies. Assume that a local miss takes 400 cycles.

(a) First touch placement.
(b) All three pages on P0.
(c) All three pages on P1.
(d) Best possible application-directed static placement i.e. a one-time call to the OS to place the three pages.

3. [30] Suppose you want to transpose a matrix in parallel. The source matrix is A and the destination matrix is B. Both A and B are decomposed in such a way that each node gets a chunk of consecutive rows. Application-directed page placement is used to map the pages belonging to each chunk in the local memory of the respective nodes. Now the transpose can be done in two ways. The first algorithm, known as "local read algorithm", allows each node to transpose the band of rows local to it. So naturally this algorithm involves a large fraction of remote writes to matrix B. Assume that the band is wide enough so that there is no false sharing when writing to matrix B. The second algorithm, known as "local write algorithm", allows each node to transpose a band of columns of A such that all the writes to B are local. Naturally, this algorithm involves a large number of remote reads in matrix A. Assume that the algorithms are properly tiled so that the cache utilization is good. In both the cases, before doing the transpose, every node reads and writes to its local segment in A and after doing the transpose every node reads and writes to its local segment in B. Assuming an invalidation-based cache coherence protocol, briefly but clearly explain which algorithm is expected to deliver better performance. How much synchronization does each algorithm require (in terms of the number of critical sections and barriers)? Assume that the caches are of infinite capacity and that a remote write is equally expensive in all respects as a remote read because in both cases the retirement is held up for a sequentially consistent implementation.

4. [5+5] If a compiler reorders accesses according to WO and the underlying

processor is SC, what is the consistency model observed by the programmer? What if the compiler produces SC code, but the processor is RC?

5. [5+5] Consider the following piece of code running on a faulty microprocessor system which does not preserve any program order other than true data and control dependence order.

```
LOCK (L1)
load A
store A
UNLOCK (L1)
load B
store B
LOCK (L1)
load C
store C
UNLOCK (L1)
```

(a) Insert appropriate WMB and MB instructions to enforce SC. Do not over-insert anything.
(b) Repeat part (a) to enforce RC.

6. [5+10] Consider implementing a directory-based protocol to keep the private L1 caches of the cores in a chip-multiprocessor (CMP) coherent. Assume that the CMP has a shared L2 cache. The most natural way of doing it is to attach a directory to the tag of each L2 cache block. An L1 cache miss gets forwarded to the L2 cache. The directory entry is looked up in parallel with the L2 cache block. Does this implementation suit an inclusive hierarchy or exclusive hierarchy? Explain. If the cache block sizes of the L1 cache and the L2 cache are different, explain briefly how you will manage the state of the L2 cache block on receiving a writeback from the L1 cache. Do not assume per sector directory entry in this design.

Previous Next

Solution of Exercise : 3

1. [0 points] Please categorize yourself as either "CS698Z AND CS622" or "CS622 ONLY".

2. [5+5] Consider a 512-node system. Each node has 4 GB of main memory. The cache block size is 128 bytes. What is the total directory memory size for (a) bitvector, (b) DiriB with i=3?

Solution: (a) Number of cache blocks per node = 2^25 and 512 bits per directory entry. So total directory memory size = (2^25)*64*512 bytes = 1 TB.
(b) Each directory entry is 27 bits. So total directory memory size = 54 GB.

3. [5+5+5+10] For a simple two-processor NUMA system, the number of cache misses to three virtual pages X, Y, Z is as follows.

Page X: P0 has 14 misses, P1 has 11 misses. P1 takes the first miss.
Page Y: P0 has zero misses, P1 has 18 misses.
Page Z: P0 has 15 misses, P1 has 9 misses. P0 takes the first miss.

The remote to local miss latency ratio is four. Evaluate the aggregate time spent in misses by the two processors in each of the following policies. Assume that a local miss takes 400 cycles.

(a) First touch placement.
(b) All three pages on P0.
(c) All three pages on P1.
(d) Best possible application-directed static placement i.e. a one-time call to the OS to place the three pages.

Solution: (a) First touch: Page X is local to P1, page Y is local to P1, page Z is local to P0. P0 spends (14*1600+15*400) cycles or 28400 cycles. P1 spends (11*400+18*400+9*1600) cycles or 26000 cycles. Aggregate: 54400 cycles.

(b) All three pages on P0: P0 spends (14*400+15*400) cycles or 11600 cycles. P1 spends (11*1600+18*1600+9*1600) cycles or 60800 cycles. Aggregate: 72400 cycles.

(c) All three pages on P1: P0 spends (14*1600+15*1600) cycles or 46400 cycles. P1 spends (11*400+18*400+9*400) cycles or 15200 cycles. Aggregate: 61600 cycles.

(d) To determine the best page-to-node affinity, we need to compute the latency for both the choices for each page. This is shown in the following table. Since P0 doesn't access Y, it should be placed on P1. Therefore, Y is not included in the following table.

Page	Home	Latency of P0	Latency of P1	Aggregate
X	P0	5600	17600	23200

X	P1	22400	4400	26800
Z	P0	6000	14400	20400
Z	P1	24000	3600	27600

Thus X and Z both should be placed on P0. Y should be placed on P1. The aggregate latency of this is 50800 cycles. As you can see, this is about 7% better than first touch.

4. [30] Suppose you want to transpose a matrix in parallel. The source matrix is A and the destination matrix is B. Both A and B are decomposed in such a way that each node gets a chunk of consecutive rows. Application-directed page placement is used to map the pages belonging to each chunk in the local memory of the respective nodes. Now the transpose can be done in two ways. The first algorithm, known as "local read algorithm", allows each node to transpose the band of rows local to it. So naturally this algorithm involves a large fraction of remote writes to matrix B. Assume that the band is wide enough so that there is no false sharing when writing to matrix B. The second algorithm, known as "local write algorithm", allows each node to transpose a band of columns of A such that all the writes to B are local. Naturally, this algorithm involves a large number of remote reads in matrix A. Assume that the algorithms are properly tiled so that the cache utilization is good. In both the cases, before doing the transpose, every node reads and writes to its local segment in A and after doing the transpose every node reads and writes to its local segment in B. Assuming an invalidation-based cache coherence protocol, briefly but clearly explain which algorithm is expected to deliver better performance. How much synchronization does each algorithm require (in terms of the number of critical sections and barriers)? Assume that the caches are of infinite capacity and that a remote write is equally expensive in all respects as a remote read because in both cases the retirement is held up for a sequentially consistent implementation.

Solution: Analysis of local read algorithm: Before the algorithm starts, the band of rows to be transposed by a processor is already in its cache. Each remote write to a cache block of B involves a 2-hop transaction (requester to home and back) without involving any invalidation. After transpose each processor has a large number of remote cache blocks in its cache. Here a barrier is needed before a processor is allowed to work on its local rows of B. After the barrier each cache read or write miss to B involves a 2-hop intervention (local home to owner and back). So in this algorithm, each processor suffers from local misses before transpose, 2-hop misses during transpose, and 2-hop misses after transpose.

Analysis of local write algorithm: Before the algorithm starts, the band of columns to be transposed by a processor is quite distributed over the system. In fact, 1/P portion of the column would be in the local cache. Before the transpose starts, a barrier is needed. In the transpose phase, each cache miss of A involves a 2-hop transaction (requester to home's cache and back). Each cache miss to B is a local miss. After the transpose the local band of rows of B is already in the cache of each processor. So they enjoy hits in this phase. So in this algorithm, each processor suffers from local misses before transpose, 2-hop misses and local misses during transpose, and hits after transpose.

Both the algorithms have the same number of misses, but the local write algorithm has more local misses and less remote misses. Both have the same synchronization requirement. Local write algorithm is better and that's what SPLASH-2 FFT implements. Cheers!

5. [5+5] If a compiler reorders accesses according to WO and the underlying processor is SC, what is the consistency model observed by the programmer? What if the compiler produces SC code, but the processor is RC?

Solution: WO compiler and SC hardware: programmer sees WO because an SC processor will execute whatever is presented to it in the intuitive order. But since the instruction stream presented to it is already WO, it cannot do any better.

SC compiler and RC hardware: programmer sees RC.

6. [5+5] Consider the following piece of code running on a faulty microprocessor system which does not preserve any program order other than true data and control dependence order.

```
LOCK (L1)
load A
store A
UNLOCK (L1)
load B
store B
LOCK (L1)
load C
store C
UNLOCK (L1)
```

(a) Insert appropriate WMB and MB instructions to enforce SC. Do not over-insert
anything.
(b) Repeat part (a) to enforce RC.

Solution:
(a)

```
MB
LOCK(L1)
MB
load A
store A
WMB                    // Need to hold the unlock back before the store to A is done
UNLOCK (L1)
MB
load B
store B
MB
LOCK (L1)
MB
```

```
load C
store C
WMB                     // Need to hold the unlock back before the store to C is done
UNLOCK (L1)
MB
```

(b)

```
LOCK (L1)
MB                              // This MB wouldn't be needed if the processor was not faulty; you lose some RC advantage
load A
store A
WMB
UNLOCK (L1)
load B
store B
LOCK (L1)
MB                              // This MB wouldn't be needed if the processor was not faulty; you lose some RC advantage
load C
store C
WMB
UNLOCK (L1)
```

7. [5+10] Consider implementing a directory-based protocol to keep the private L1 caches of the cores in a chip-multiprocessor (CMP) coherent. Assume that the CMP has a shared L2 cache. The most natural way of doing it is to attach a directory to the tag of each L2 cache block. An L1 cache miss gets forwarded to the L2 cache. The directory entry is looked up in parallel with the L2 cache block. Does this implementation suit an inclusive hierarchy or exclusive hierarchy? Explain. If the cache block sizes of the L1 cache and the L2 cache are different, explain briefly how you will manage the state of the L2 cache block on receiving a writeback from the L1 cache. Do not assume per sector directory entry in this design.

Solution: This is typical design for an inclusive hierarchy. For exclusive hierarchy, you cannot keep a directory entry per L2 cache block. Instead, you must have a separate large directory store holding the directory states of all the blocks in both L2 and L1 caches. If the L1 and L2 block sizes are different, you need to keep a count of dirty L1 sectors with the owner; otherwise you won't be able to turn off the L1M state even after the last dirty sector is written back by the L1 cache. This has no correctness problem, but would involve unnecessary interventions to L1 caches. Fortunately, you don't need any extra bits for doing this for a medium to large scale CMP if the ratio of L2 to L1 block sizes is not too large. Assume a p-core CMP with L2 to L1 block size ratio of k. So you need log(p) bits to store the owner when L1M state is set,
log(k) bits to store the dirty sector count, and p bits to store the sharer vector if the block is only shared. So as long as p >= log(p)+log(k), we can pack the owner and dirty sector count into the sharer vector. This condition corresponds to k <= 2^{p-log(p)} or k <= (2^p)/p. The number on the right-hand side grows very fast with p. For eight cores onward we are in the safe zone.

Previous Next