

Notes on Memory Consistency and Cache Coherence

Where the rubber meets the road in a computer system is, in some sense, how instructions *are* handled by the hardware and how instructions are *expected to be* handled by the software. This is the point at which hardware and software have differing goals and satisfying both without breaking the system is the Computer Architect's challenge. Software wants to keep things simple. It wants a consistent, never-changing view of the computer it is programming. Software does not want to be modified with every generation of hardware changes. Hardware, on the other hands, wants to runs programs as fast as possible and underlying technology might change the challenges on a generation by generation basis. Making a program run fast is, to a large extent, dependent on good software and algorithmic efficiency. But once that has been achieved, any further performance improvements, without overly complicated software maneuvers, is in the realm of hardware (and compilers, which know the hardware more intimately than higher levels of software). And in order to make programs runs fast, hardware has to cut corners and do things differently than what the simple sequential software program dictates.

A couple of examples of how hardware speeds up execution without doing exactly what the software expects it to be doing, are:

1. Out of order execution: Hardware might execute independent, future, instructions while waiting for a current instruction, which is taking long, to finish.
2. Branch prediction and speculative execution: Hardware may predict the direction execution would likely follow at a decision point and continue to execute down that path even if the decision has not yet been confirmed by the hardware.

To maintain correct execution, hardware has to, still, make sure that it can undo any state changes it has made in its attempt to complete the program fast. It helps to recognize that the hardware is, quite often, able to speed up programs significantly by doing such tricks; however, these tricks are kept invisible to the software - these tricks do not change any state visible to the software. Software's simple view of the computer it is programming must not be polluted by the hardware trying to do things out of order or speculatively because the state change can unintentionally change the program execution. The simple view of the computer visible to software is called the architecture of the computer and the state of the machine visible to the software is called the architected state. The architected state consists of architected registers (fast storage, where software can perform operations on operands), memory (slow storage, where software can maintain longer term data, the program code being run on the machine, intermediate results, operands not being actively operated upon, final results etc.) and the instruction set (which defines how you can write programs - what operations that may be performed on operands, the operand sizes, the size of the memory, number of registers etc.).

In a uniprocessor, there is only one program running at any time. The illusion of maintaining the architected state, while running ahead in terms of actual execution, is a little bit easier in case of a uniprocessor compared to a multiprocessor. The notion of "time" needs to be maintained across instructions in that single thread of execution to conform to the notion of time expected by the software. For example, if the software executes

```
inst 1:   $regA = regB + regC$   
inst 2:   $regD = regA + regE$ 
```

that is, an instruction $regA = regB + regC$ followed by $regD = regA + regE$. The second instruction should pick up the value of $regA$ that the first instruction generated. If the second instruction is executed before the first instruction, then the program execution will be incorrect. If however, the second instruction were $regD = regE + regF$, it might have been possible for the hardware to execute it without waiting for the first instruction to complete. The hardware takes care of such situations (called register dependencies) relatively easily because the dependences between instructions are identifiable by the register names. One may argue that if the *value* in the register does not change due to the first instruction, then the second instruction could have gone ahead and executed without waiting for the first one to complete. Though true, such determination adds more complexity. It is, however, possible to go ahead and execute the second

instruction ahead of the first instruction, but not update the architected state until the hardware makes sure that the execution was, indeed, correct.

The situation gets a little bit more complicated when the dependence is a memory dependence. That is, the dependence cannot be readily identified by looking at the register names because the register names may be different, indicating no register dependence, but there might still be a dependence. For example, if the software executes

```
inst 1:  regA = regB + regC
inst 2:  store MemoryAtAddress(regA), 10
inst 3:  load regE, MemoryAtAddress(regZ)
inst 4:  regD = regE + regF
```

In this case, using only register dependences, the hardware could identify the dependence between inst 1 and inst 2. It could also identify a dependence between inst 3 and inst 4. However, seeing no dependence between the first pair and second pair of instructions it might go ahead and execute the pairs independently. That would be wrong if the *value* of *regZ* happened to match the value of *regA*, because then the memory address inst 2 is storing to would match the memory address inst 3 is loading from. Therefore, for correct execution, inst 2 would have to complete the store before inst 3 could start its load. This is the memory dependence which hardware has to also identify.

In a multiprocessor, multiple programs (or pieces of programs) may be running simultaneously. The notion of "time" across instructions within a given thread of execution *and* the notion of time across instructions in separate threads, both, need to be maintained by the hardware in accordance with the expectations of the software. It may seem like the hardware required to maintain the notion of time across instructions within a given thread of execution would be similar to how it is done in a uniprocessor. It is similar, but there is a significant difference. In a uniprocessor the hardware *knows* that there is no other thread of execution running and it takes advantage of that knowledge to pull off a few more tricks which might not work in a multiprocessor system (a shared-memory multiprocessor system, to be precise). The notion of time across threads is comparatively simpler to think about because any synchronization at that level has to be requested explicitly by the software, and that allows the hardware to execute each individual thread at it pleases until it detects a synchronization command in the instruction stream of a thread.

Memory Consistency is basically the guarantees that the hardware provides to software with respect to the ordering between memory related instructions (loads and stores) within each thread and across multiple threads. The ordering clearly defines what the software can and cannot expect in terms of hardware's ordering behavior for all combinations of load and store instruction to any address. A way to break this space down is to think of memory operations a pair at a time, Load-Load, Load-Store, Store-Load and Store-Store. Another dimension to think about these pairings is the behavior when two memory operations are to the same address versus when they are to different addresses.

Cache Coherence, which is often mentioned in the same breath as Memory Consistency, does not really deal with the ordering between memory ops, and is not a part of the contract between hardware and software. It is purely a hardware optimization to help hardware function correctly, and thus provide the Memory Consistency abstraction correctly. Cache Coherence deals with keeping multiple copies of a piece of memory, which may be cached in several caches across a multiprocessor, in sync. If one of the copies of a block of memory, sitting in one of the caches, is modified by a processor, all the other copies have to be immediately informed so that the processors that are using those other copies do not read state versions of the data. In addition, if there are two writes to the same part of memory, maybe sitting in two different caches, the first write has to be made visible to *all* the other caches in the shared-memory multiprocessor system before the second write is made visible to *any* of the other caches. These two properties are called "write propagation" and "write serialization".

Let's look at the issues of coherence, consistency etc. on uniprocessors first, and then on multiprocessors. Often a single processor is designed and then many of them are put together to behave like a multiprocessor (Symmetric Multi-Processing). However, it is important to realize that a uniprocessor

intended to be used in a multiprocessor system may have to be designed differently than one not intended for multiprocessor use, depending on the consistency model to be supported. We'll see why.

Uniprocessors

In a uniprocessor reordering between memory ops to *different* addresses is fine

(load A, load B) to (load B, load A) is OK

(store A, store B) to (store B, store A) is OK

(load A, store B) to (store B, load A) is OK

(store A, load B) to (load B, store A) is OK

The reason this is fine is because there is no other thread in the system which could observe this order and infer anything from the order. In a multiprocessor, there are other threads and often the *order* of the memory operations carries information. This will become clearer in the multiprocessors section below.

Reordering between memory ops to the *same* address needs a little more attention

(load A, store A) to (store A, load A) is NOT OK

(store A, load A) to (load A, store A) is NOT OK

Although reordering (store A, store A) or (load A, load A) does not seem to make much sense, remember that the amount or location of data being read or written might be different between the two ops. Even so, (load A, load A) does seem totally harmless. And it is! (store A, store A) ordering might have to be maintained because the bytes being updated at a given address (A), by the two stores might be overlapping, causing the older value in that overlapping section to overwrite the newer value if the order is reversed.

In general, the uniprocessor hardware makes sure that for a *given* address, (store A, load A), (load A, store A) and (store A, store A) order is maintained. But they do like to send things out of order, even

speculatively out of order, if possible. So to continue to execute out of order, while making sure that the above mentioned orders are maintained, the speculative out-of-order processors use a load store queue.

All instructions enter the ReOrder Buffer in program order and retire in program order. In order to save time hunting for address collisions with older loads and stores, load and store operations are also put into hardware structures called load queue and store queue respectively. The entries in the load and store queue contain the address that the load or store is for and the pointer to the ROB entry containing the corresponding load or store. In other words, the load and store queues have no extra information that the ROB does not have; it is a condensed version of the ROB holding only the memory ops from the ROB.

(store A, load A)

A load, before it is marked ready to go with all its operands ready, should make sure that there is not an older store in the system (and, therefore, in the store queue) which is going to write to the same address. If there is, and the store data is available, the load should make sure it uses that data (it might have to still do a load and before sending the data to the processor register merge the store data). If the load finds out for sure that there is no store to the same address that is older than itself, it can issue. If there is some older store in the store queue, whose address is not known yet, then the load cannot be sure if it can go. What if the store's address matches its address when it is eventually calculated? Some microarchitectures allow a load to still go ahead speculatively. The load is betting that most likely the address of the unknown older store will be different and therefore there is no need to wait for it. However, to guarantee correctness, before the load retires, it must make sure that the address, indeed, did not match. How does it make sure of that? The store might have retired by the time the load is ready to make that check (remember, the store is older; it has no dependency on the load). Hence, the *store* takes up such responsibility. The store knows that once its address is computed any loads that come in to the load queue from that point on will be able to check its address. Only younger loads which checked its address *before* its address was ready could have speculatively gone ahead. So as soon as the store's address is generated it searches the load queue to see if a younger load with a matching address went ahead. If it finds a match it intimates the hardware to squash the load and all following instructions. Some optimizations are possible here. All stores need not search the load queue upon address generation. Only those stores which were bypassed by eager loads which did not wait for the store address generation need to check the load queue. A bit in the store queue could be set against such stores which need to run the check. Symmetrically, all loads that went ahead speculatively could be marked, so that the store knows which younger loads to compare its newly generated address

against. Another optimization could be to supply the correct store data to the load that went ahead speculatively (since it would not have retired anyway, it being younger than the store). Any instructions which used the speculative load data might have to be squashed and re-executed, but only if the data supplied by the store was different than what the load speculatively loaded (lot of times the value stored might not change the value loaded)!

(load A, store A)

A store needs to make sure it does not modify the architected state out of program order compared to an older load which goes to the same address range. The older load should not be allowed to see the store data. A store may complete out of order but it should not make that store data be visible to any older loads. The store does this by writing its store data to the store queue (or an associated, ordered, structure) until it is at the head of the ROB, at which point it can retire the store data to the cache. In the previous section we saw how a store catches a younger load to the same address range that has gone ahead of it. It does not bother looking for older loads. The older loads do not bother looking for younger stores. And the younger stores do not update the cache out of order. So the older load does not get the younger store's data.

(store A, store A)

A store needs to make sure it does not modify the architected state out of program order compared to an older store which goes to the same address range. The older store should not be allowed to overwrite the younger store's data. This is achieved by having separate entries for each store in the store queue (even if they are to the same address) and by updating the cache in program order, even though the stores are considered complete as soon as they have written their data to the store queue. That is the store is completed out-of-order as soon as its data is written to the store queue, but committed in-order at which point that data is moved from the store queue to the cache.

Multiprocessors

In a multiprocessor system, there are two programming models - shared memory model and message passing model. In message passing two parallel tasks see separate virtual address spaces, do not share any data, and communicate using explicitly constructed messages which are delivered via the I/O sub-system. The shared memory model allows threads/processes running on two separate processors (more accurately, two separate processing contexts) to communicate via memory, that appears shared, using simple load and store instructions! If the two parallel contexts are *threads* (kernel-level threads or user-level threads like pthreads) the virtual address space is also shared between the contexts. A thread can store to an address and a different thread can get that value simply by loading from that address after the store is done! Simple! Well, simple to program at least. There is some complexity that arises because of the notion of "after" - more generally, the notion of global order. We'll get into that in a bit. Continuing the discussion, if *processes* implement the parallel contexts, the contexts have to communicate via inter-process communication (IPC), such as pipes, sockets etc. The two contexts do not share the same virtual address space, but the OS could let them share the same physical frame of memory. I am not sure if this continues to be called shared memory programming. In other words, I am not sure if the "memory" in shared memory programming refers to virtual address space or the physical memory. At any rate, in shared memory programming, the physical memory is, typically, also shared by contexts. Whether there are one or more physical DIMMs is irrelevant. If the physical memory is distributed (across many nodes, for example) the physical memory is referred to be distributed shared memory. The main idea of shared physical memory is that data at a given address is located at only one place across the entire shared physical memory. This is, as is probably obvious, not very efficient. If a processor has to make a write to a memory location visible to all other processors which might be interested in using the data, it must update the memory each time it is written. What's more, the physical memory may be quite far away. If during the time the write makes it to that distant memory, some other processor wants to read the location, it must not be allowed to read it, for it might get the stale value. So there are performance issues (memory far away) and, more importantly, correctness issues (writes have to be made visible to other processors). But wait. It gets even more complicated. When some work gets distributed across processing contexts the tasks (pieces of work) need to synchronize - they need to have a notion of time - a notion of "before" and "after". A task may one proceed "after" a different task has reached a certain point in *its* code. A task may have to make sure no

other task is using a resource while it uses it. Such communication of "order" often requires memory accesses to *different* addresses to be retained in program order! An example scenario is - a task writes a value (say, at address A) only *after* it makes sure that a different variable (say, at address B) is set to 1. Maybe the latter value is the indication from a different task that the first task may now proceed. So the sequence is - read a location, compare with a certain value and conditionally write to a different address. If the compiler or the hardware reorder the read and write (which a uniprocessor would be happy to do in order to get out of order execution going, especially given that these are to different addresses) the "synchronization" between the two tasks could be screwed up. The write to Address A would happen even as the read of Address B, which happens later when the order is reversed, indicates that the write should not have happened! Therefore, in short, a multiprocessor system is a whole another ball game.

Shared memory makes writing parallel programs easier because the parallel threads do not have to worry about data layout, data affinity, messages etc. But to keep that shared memory illusion going, the hardware has to do something beyond providing the usual uniprocessor promises of keeping the order between (load A, store A), (store A, load A) and (store A, store A) intact. It has to provide guarantees that (load A, load A) is in order! It has to also provide guarantees that (load A, store B), (store A, load B), (load A, load B) and (store A, store B) are in order! Basically, unaware of what the programmer of the parallel tasks is using to synchronize between the two, the hardware has to pretty much say "I promise to maintain program order between memory operations in each thread, irrespective of the addresses and irrespective of the memory operation (load or store). Further, I also promise to maintain a consistent global view of these memory operations ". In other words, the "program order" promises to maintain the order of memory ops within a thread; the "global order" promises to maintain *some* consistent global view of all memory operations. The "program order" is predictable and repeatable; "global order" is not predictable or repeatable, but for a given run it is guaranteed to be consistent, that is, the order in which one processor sees the memory operations on the bus will be the order in which all other processors see memory operations on the bus. This "promise" is also called the "memory consistency model" supported by the architecture. The architecture making this particular type of promise is called a "sequentially consistent architecture".

Sequential Consistency is a bit much in most cases. It seems to seriously restrict the amount of instruction reordering and out of order execution the compiler or hardware can take advantage of. (Or does it? We'll get to that in a bit). Typically, not all of the addresses written to and read from are used for synchronization; in fact, only a small percentage of the addresses are. Added to that, most uniprocessor hardware already provides caches and load store queue support. It would be great if a write could be stored in the cache rather than be sent all the way to the physical memory (potentially a remote memory). That way, multiple writes would not cause multiple writes on the memory bus. Also, it would be great if we could use the load store queue and maybe extend it a little bit to provide the ordering where necessary, while allowing the hardware to be speculative and out-of-order. This is where the microarchitecture comes into the picture. Caches and load store queues are all microarchitectural structures. The programmer does not know about them. The programmer only knows that the architecture has promised a sequentially consistent model. The microarchitecture can do whatever it wants under the covers, for performance's sake, as long as it maintains the illusion that memory accesses are sequentially consistent! This is the approach adopted by the MIPS R10k processor.

The writes can go to the caches instead of having to go all the way to the memory for performance reasons. For correctness, the writes that make it to the caches have to be made visible to all other "sharers", that is all other processing contexts that might be using the data. In order to make sure that the other processing contexts do not read or write the *same* data before the write by the first processor is made visible to all the other caches, there needs to be some centralized structure that orders memory accesses to the *same* address. This brings in the notion of coherence. Coherence means that memory access to a given address are visible to all sharers in a coherent way. Notice that there is no strict "global" ordering between memory accesses to a given address by one processor and another. The various parallel tasks are all running independently, and they all obey some synchronization rules that have been coded into the program. But they do not rely on a central global timekeeper. It is not required that memory accesses by two parallel tasks are to be ordered in some fixed global order does. There is no one who keeps global time. The programs are developed with that knowledge. The software is written so that it works correctly no matter which order ends up actually happening. Only requirement is that such order be made honestly and consistently visible to all threads.

The only requirement is that there be *some* central ordering unit that provides *some* order visible to everyone. The order could be the reverse of what "really" happened. It does not matter because the only "reality" visible to all and abided by all is the reality the central ordering unit makes visible. So thread A may have done its memory operation "before" thread B in real time, but if thread B's change reaches the central ordering unit first, thread B's memory op is considered to have happened first. Thread A knows nothing about what actually happened in real time. So it believes the central ordering unit when it advertises the fact that the memory op from thread B reached it first, and backs off (makes appropriate state modifications in the cache).

With that understanding, let us revisit the issue of coherence. The only way a processing context can change the state of the shared memory is by stores. Therefore, stores have to be made visible to all the other sharers. Since all processors check caches first before going to the memory, the store to a memory location by a processing context has to be able to do two things to make sure the latest store is visible to all other processing contexts. It must either update other caches with the stored data, or invalidate the cacheline corresponding to the store address in all the other caches. In the latter case, it must also supply the data to the other processing context when it needs it. The central ordering structure must also provide guarantees that while a store is being thus propagated to other caches, no other cache can access that address for a memory operation. These two properties are called write propagation and write serialization, and together provide coherence. The coherence is at the cache level since it is the caches that are maintaining multiple copies of the same data. In a traditional shared memory system, the physical memory does not keep shared copies of data; it does not need memory coherence. There is also the issue of register coherence. Two processing contexts might have read a value to a register. So when one of them writes to the address, the other one may not know about it if the write propagation does not reach the registers; and it typically does not reach the registers. This is prevented by making sure that the programmer (or the compiler) marks shared variables as not register allocatable. So these memory locations are not loaded into a register before being used in any operation, but are, rather, used straight from the cache or memory in an operation. That is, one of the operands in an instruction is a memory operand.

Cache coherence is a hardware trick to continue to take advantage of caches. Caches allow data to be closer to the processor and also allow the system to keep multiple copies of the data if there are multiple sharers. It is just an illusion that the processors share a single memory where there is only one copy of each data. So, as soon as someone updates the memory location, the hardware makes it look like the updated value is visible to all threads and is ready to be used by the other threads. It does not have anything to do with the hardware providing ordering between memory accesses to *different* addresses. However, sequential consistency requires that all memory accesses within a program retain their original program order. Therefore, this order has to be maintained at the individual processor level. Once beyond the processor boundary, that is, once retired, the order is maintained by the rest of the memory hierarchy. The processor really feels restricted if it truly executes memory operations in program order. It instead uses the load store queue and speculatively executes instructions out of order, just like it did in the uniprocessor context.

(store A, store B)

The order between two stores, (store A, store B), is maintained automatically using the load store queue by simply retiring them in-order and, therefore, making their store data visible to the memory hierarchy in-order. This is exactly the same as described in the uniprocessor case even though the addresses of the two stores are different.

(load A, store B)

The order between a load and a store, (load A, store B), is similarly maintained by retiring the instructions in order and having the load ignore a younger store in the store queue.

(store A, load B)

The order between a store and a load, (store A, load B), is a little tricky. The load should not go ahead of the older store even though the address of the store is different. This implies an extension to the uniprocessor load store queue functionality where a load does not issue if there are any older stores, irrespective of their address being ready or not.

(load A, load B)

Similarly, the order between a load and a load, (load A, load B), is also tricky. A younger load should not go ahead of an older load even though the addresses of the loads are different. Does this imply that a load cannot issue if there is another load in the load queue? It seems very restrictive. There is an optimization possible here. The order of the loads, (load A, load B), being correct only matters if there was a (store B, store A) sequence from some other processor context in between the two loads. If that happened, then reordering load A and load B could lead to an inconsistent global order - load B, store B, store A, load A. So, load A sees the value stored by store A, but a younger load in program order (load B) sees a value of B older than what the older load could have seen. Figure 1 explains this scenario in greater detail.

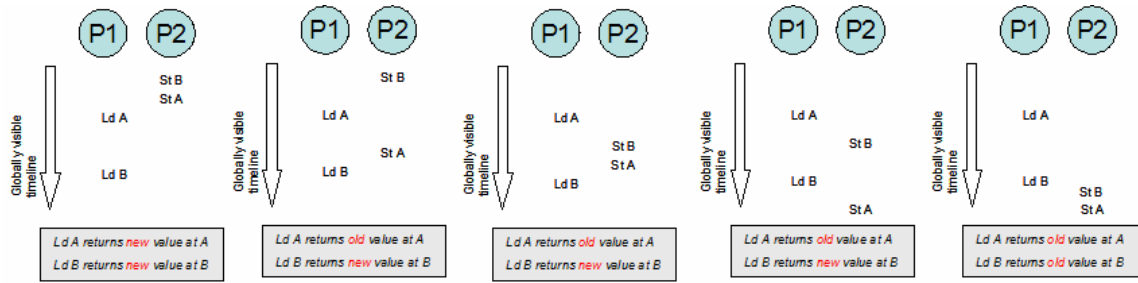


Figure 1(a) : Various valid global orders between the statements on the two processors when per-thread program order is maintained. Note that the new value of A and old value of B can never be seen by P1

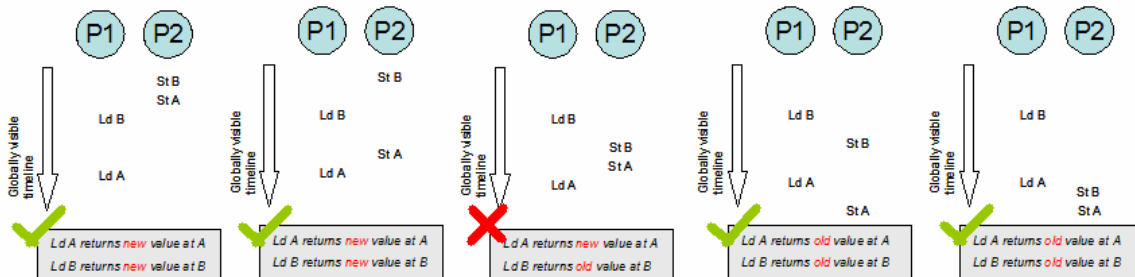


Figure 1(b) : Various global orders between the statements on the two processors when program order is reversed on P1's thread. Note that in the third case shown, an inconsistent global order, not possible by sequential consistency, results

So, the load store queue only needs to detect this case and take corrective action. One option is to not allow loads to go out of order at all, like described above (insulated load queue). The other option is to let the loads go out of order and allow a store on a *different* process context to search this load queue looking for loads that might have gone out of order.

A special case of the above is if the loads are to the same address, (load A, load A), and there is a store A in between from another processing context. In that case the reordering of the loads should not be allowed. This is explained in Figure 2. Otherwise, reordering is fine. Therefore, when the store A happens it can search the load queue of the processor doing the loads and if it finds that there is an outstanding load to the same address it squashes it and all following instructions. That is, when the store reaches a processor and finds that there are no loads to the same address, either both the loads have already committed (so out of order does not matter), or neither of the loads have yet been started (so, out of order does not matter). Only if there is an outstanding load to the same address is there even a chance that there might be a second load to the same address which is actually out of order. Therefore, to be safe it is OK to squash the oldest load to that address and everything younger in the processor. This is conservative since there may not be another load already in or coming in at that processor before the first load is serviced. Instead of this snooping approach, the more localized way to take care of this is to make the load either not issue ahead of an older load's completion (if the address of the older load is known), or, if the address of the older load is not

known by the time the younger one is ready to issue, to issue the younger load and upon address generation (or completion) of a load to check to see if a younger load to the same address was issued, and if so, to squash it. This latter approach is also conservative in that there might not have been any Store to that address in between the loads. So it may make sense to keep track of all the store (invalidate) addresses that showed up on the bus after a load was speculatively issued without knowing the address of an older load. When the address of the older load becomes known, it checks for younger loads with a matching address only if there was an external store (invalidate request, in an invalidate-based coherence protocol) that matched the address of the older load. Basically, however it is implemented, when the older load completes address generation it can make sure a younger load did not get ahead of it.

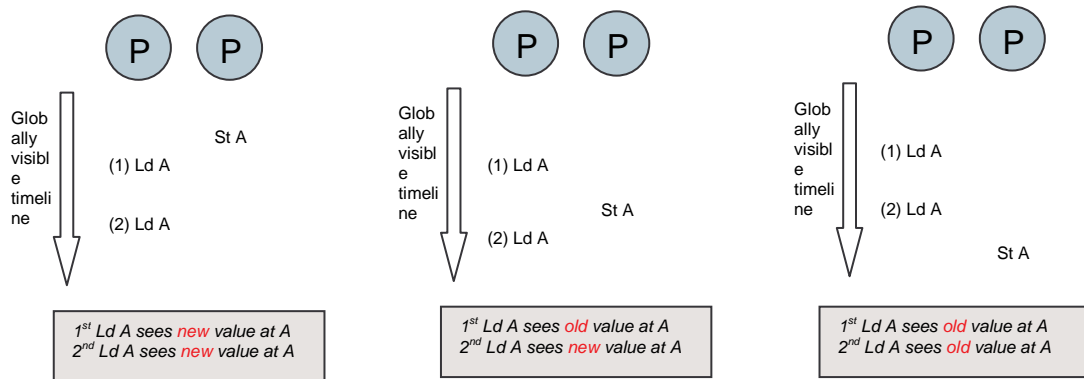


Figure 2(a) : Various valid global orders between the statements on the two processors when per-thread program order is maintained. Note that the first load never sees a new value of A if the second load sees an old value.

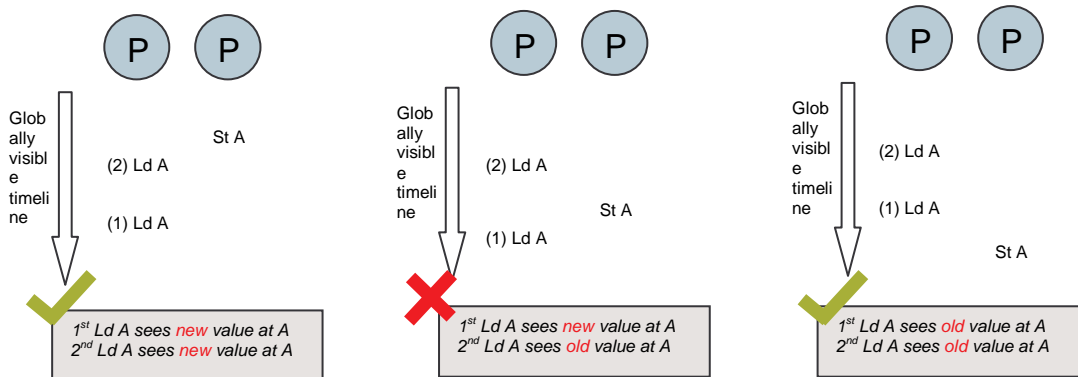


Figure 2(b) : Various global orders between the statements on the two processors when program order is reversed on P1's thread. Note that in the second case shown, an inconsistent global order, not possible by sequential consistency, results.

A different approach to memory consistency, which is more common nowadays, is to use "weaker" consistency models. That is, the architecture makes feebler promises. It instructs the programmer that it will only guarantee ordering between operations of a certain type - for example, "processor consistency" relaxes the ordering between (store A, load B) to allow the load to get started faster. This works quite well because loads are usually more critical and, secondly, such constructs are pretty rare in code written by programmers to synchronize parallel tasks. Note that "pretty rare" does not mean it does not happen. If it does happen "processor consistency" makes no guarantees of the order between such a store and load. So, because the memory consistency model has made its promises clear to the programmer, the onus of writing code free of such unsupported ordering falls squarely on the programmers' shoulders. Weaker consistency models promise even less. PowerPC's memory consistency model says, "My architecture makes no guarantees about ordering between loads and stores to different addresses.". Of course, for the same address it continues to provide the ordering between (load A, store A), (store A, load A) and (store A, store A)

within a single context. This is the typical uniprocessor promise. However, not making any guarantees is no good. How then does a programmer make sure that a certain set of memory operations, that are synchronization critical, happen in a certain order? So, as a way of enforcing order, the PowerPC architecture provides a "sync" instruction. This instruction makes sure that all memory operations before the sync are made visible globally on the system before any of the memory operations after the sync are made visible globally. So if the programmer wants ordering between (load A, load A) then he has to execute (load A, sync, load A). As an aside, the reason ordering between two loads to the same address (say, A) could be important is because there might be a store A from a different parallel task in between the two loads. If that is the case, the first load should see the stale value of A and the second load should see the new value. If loads are reordered, the opposite happens; the program behavior changes. So, how does a sync do this? How does it make sure that none of the memory operations after it are made visible until all the ones before it have been? It is usually pretty simple. Since instructions are fetched and put into the issue queue and ROB in order, when a sync is fetched, nothing is put into the issue queue and fetch is stopped. Then when the queues have been drained, the fetching and dispatching beyond the sync continues. Even for weaker consistency models, the ordering between (load A, load A) is supported by hardware as described earlier. This is because programmers typically forget to insert syncs between loads to the same address.