Memory Consistency in Multiprocessors

Atticus Deutsch

Introduction

This article presents a overview of what memory consistency means within the context of a multiprocessor and discusses different models of memory consistency and their implementations. This article is meant for people who have some knowledge of computer architecture such a a college level course.

Within the last decade we have seen architects push computers to the limits in terms of power consumption. While increasing clock frequency linearly scales performance, it also linearly scales power consumption. Thus, clock frequency has begun to level off [1]. Performance has been saved ,however, by a shift to multi core or (multiprocessor) machines. Multi

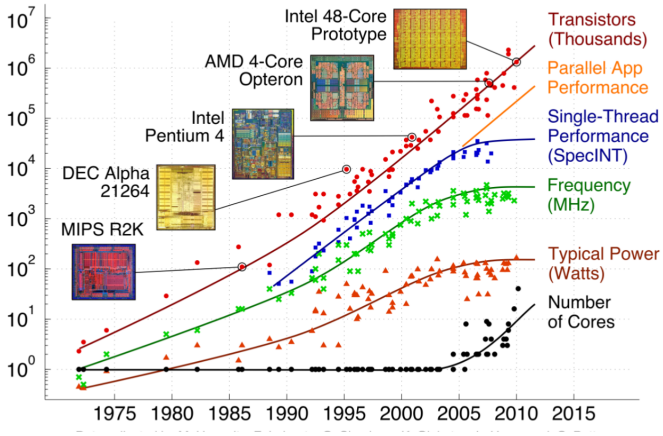

Fig. 1. The power problem in computing [1]

processors increase performance by exploiting parallelism. Each processor in a multiprocessor could be working on their own problem - as if being multiple separate uni-processor machines - thus theoretically being able to scale performance linearly with respect to the processor count. The second scenario is multiple processors working on a *single* problem, or equivalently, doing computation that has shared data. When a programmer writes the stream of instructions for each of the processors, it's possible that the some of the streams refer to and manipulate the same data. The programmer thus needs a way of understanding in what overall order the memory operations of any of the streams of instructions will be executed, so they can write a program that does what they want. This problem is solved by a computer's memory consistency model.

What is memory consistency?

When we program computers in a language like C, how do we reason about making the code do what we want? How do we define a correct execution? For the most part, without thinking about it, we write our lines of code from top to bottom in the order that we want the commands to be executed. Thus, on a uniprocessor, we can define a correct execution in terms of the order of instructions, namely as being **equal to that of the instructions being executed in the order they are written.** The machine, under the hood, might not actually execute instructions in the order they are written, but it will generate the same result.

The rule in bold above can be thought of as a **memory consistency model**, and the implementation of that rule a **memory consistency implementation.**

In the case of the uniprocessor, it's trivially easy to define a correct execution in terms of the

order of instructions, but what about in the case of a multiprocessor with multiple streams of instructions? This is how all modern computers operate and each one has a memory consistency model.

Whats the difference between memory consistency and memory coherency

Many students (and even professors who are removed from the subject of computer architecture) conflate memory consistency with memory coherency. Memory coherency is the concept of synchronizing copies of the same data. The simplest definition of memory coherency is **a processor always reads the latest write to a memory location**. In a multiprocessor, **cashe coherency** is making sure that cashe blocks across all caches which refer to the same memory location are synchronized. A processor's cashe holds copies of data from main memory closer to the processor, and also is built with faster memory technology, thus it improves performance. Cashe coherency makes it so that each processor's cashe is identical with respect to its processor making a memory request. Cashe coherency is important because without it, we could not have multiprocessors with caches. This is because any changes to shared memory by a processor would not be seen by the other processes even in a write-through cashe scenario. Without cashe coherency, by definition, you wont be able to do anything you want with a multiprocessor. When defining a memory consistency model, if the machine is to have a cashe, cashe coherency is assumed.

CONSISTENCY MODELS

There is a diverse taxonomy of consistency models in the wild each on defining what a *correct* execution of a program is.

Sequential Consistency

What is Sequential Consistency?

Recall the definition of a correct execution of a single stream of instructions from the introduction as being: **equal to that of the instructions being executed in the order they are written.** The most intuitive abstraction from single stream (processor) to multiple streams (multiprocessors) of this correctness definition was first mentioned by mathematician Dr. Edsger W. Dijkstra in his 1971 paper, *Hierarchical Ordering of Sequential Processes*[2], but later formalized by and coined **sequential consistency** by Leslie Lamport in his 1979 paper, *How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs* [4]. An N-process program is sequentially consistent if:

> "the result of any execution is the same as if the operations of all processors (cores) were executed in some sequential order, and the operations of each individual processor (core) appear in this sequence in the order specified by its program."

Let's be clear, however. Just because the memory consistency model says the result of the execution must be the same as if the operations from each individual processor (core) appear in this sequence in the order specified by its program, doesn't mean that the operations from each individual processor *actually* need to be executed in the order specified by its program. However, when trying to build a sequentially consistent machine we see that we see that instructions can rarely be executed out of program order. Consider the following scenario:

```
//initially A=0, B=0
```

Processor 0	**Processor 1**
`A=++;`	`while(flag==0){} //spin`
`B=++;`	`print A;`
`flag=1;`	`print B;`

In the above example, from `processor 0`'s perspective, it doesn't matter what order the three operations are done in. It could execute them in any order and get the same result. This is what a uniprocessor would do in a technique OOO (out of order execution) because it might be fastest to execute the reads to `A` & `B` first then execute the write to `flag` then lastly execute the writes to `A` & `B`. However, if `processor 1` did all of its operations right after `processor 0`'s write to `flag`, 0 would be printed as being the value of `A` & `B`.

We clearly see however that any sequentially consistent execution would always print `A` prints as 1 and print `B` as 1, thus that reordering is not aloud. It turns out any speculative reordering on a sequentially consistent machine requires a system of verification that consistency wasn't broken. [8].

```
//initially A=0
```

Processor 0	**Processor 1**
A=1;	B=1;
	print A;

In the above example, A could print as 1 or 0 because processor 0's instruction could occur before or after or in between those of processor 1. This demonstrates that Sequentially consistent systems are not deterministic.

How is sequential consistency implemented?

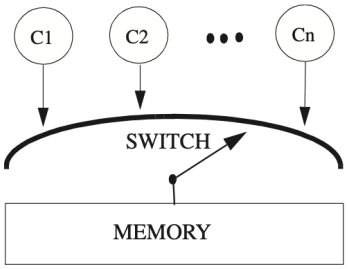

Fig. 2. Sequentially consistent machine with a switch [8]

A naive implementation of a sequential consistent machine is one where each processor executes memory operations in program order, and the single memory module serves the different processors one at a time using a switch. See 3.

In his paper, Lamport provided a more efficient implementation that achieves sequential consistency. Namely, a requirement that all memory operations be *issued* in program order with respect to a single processor, and for each memory module, there must also be a FIFO (first in first out) queue, which executes memory operations in the order that they were issued. This is a less strict model then using a single queue and memory module like in Fig. 3; however, both imply sequential consistency. This FIFO protocol introduced by Lamport was pioneering in the sense that is was an architectural change that didn't change the memory consistency model, however, was able to improve performance.

In 2006 it was Miexner and Sorin who first proved the intuitive notion that cashe coherent systems which receive requests in program order (with respect to individual processors) are sequentially consistent [6]. This way, each processor can have its own cashe, and as long as the cashes are coherent, a multiprocessor can benefit from the low latency and high bandwidth of capabilities cashes.

Branch speculation is also very important part of processor efficiency. Instead of having to wait for the result of the branch instruction to finish which slows down the processor by only allowing for a single in flight instruction, modern processors speculate the next instruction after a branch. If they guessed correctly, everything is fine. If the guess incorrectly, the incorrect computation is flushed. Today, processors predict branches with around 99% accuracy. Correct branch speculation is naturally sequentially consistent. Executing loads in an incorrect speculation will also maintain sequential consistency when flushed; however, stores cannot.

Over the years, the performance of systems which adhere to sequential consistency failed to compete with that of less strict memory consistency models. Today, sequentially consistent systems are no longer existent on the commercial market.

Strict Consistency

In strict consistency each operation in each stream of instructions is executed in program order. In addition, only one operation may be in flight at a time, and the order in which operations from different streams are interleaved is deterministic. Consider this example: In the above figure, time is the x-axis, and W(x)1 means a write to the variable x returns 1. This example is sequentially consistent, just consider the sequence R(x)0, W(x)1, R(x)1. It's not strictly, consistent, however because W(x)1 happens before R(x)0 in time thus a request to

```
P0:    W(x)1
P1:                R(x)0          R(x)1
```

Fig. 3. Execution which is not allowed under strict consistency [5]

the variable x must return a 1. In practice, strict consistency is only used in uniprocessors.

TSO/x86 Consistency

Research in the field of memory consistency, like other fields, is driven by what has applications in the real world. In the real word of computing, performance is key, and this has lead to many architects, like those who designed x86, to chose the memory consistency model called TSO/x86. As aforementioned, sequential constituency is slow. Sequential consistency can make it very difficult to do branch speculations as well implement store-buffers. Many believe that x86's choice of processor consistency was largely based on keeping the store buffers, which relieve the processor from having to stall when issuing a store. In order to make a store in a multi-processor the processor who wants to make the store must achieve write privileges of the necessary cashe block. For instance, if the cashe coherency is following the MSI protocol, the processor making the store must be granted M state for the necessary cashe block[1]. It can take some time to get the necessary cashe block in a writing state, and that's why modern processors have store buffers. A store buffer is essentially a queue of pending stores. Reads don't have to wait in the queue, and can fly past the store that were issued before them. If the read is for a address that is to be updated by a store in the buffer, the read can simply read from the buffer. Having these buffers is fast and all, but it breaks sequential consistency. Consider the following scenario:
T1 writes HI to 0xA, it waits in T1s buffer
T2 writes BYE to 0xB, it waits in T2s buffer
T1 reads 0xB and sees 0
T2 reads 0xA and sees 0

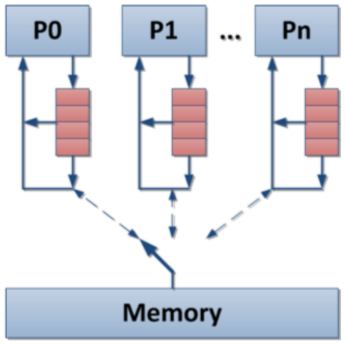

Fig. 4. Multiprocessor with store buffers [5]

The problem with this is that a processor can't see the write that was made by the other processors, making the result non-sequentially consistent. A consistency model that allows store buffers would have to relax the *write followed by read* ordering that sequential consistency enforces. This is exactly what TSO/x86 does.

There is a little more to the TSO/x86 memory consistency model than simply relaxing write followed by read ordering. TSO/x86 was first defined formally in the paper *x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors* [7] where Sewell et al. explain companies such as Intel and AMD's neglect of defining the memory consistency model of x86. Sewell provides a definition of **x86-TSO** which has so far been consistent with Intel and AMD processors. This model includes a FENCE() operation which can by used to situationally enforce write followed by read ordering, thus enforcing sequential consistency. This way, using FENCE(), any algorithm that requires sequential consistency - such as Dekker's Algorithm[2] - can be implemented on an Intel/AMD processor.

Processor consistency is a generalization of x86-TSO that was first defined by Gharachorloo et al.[3]. In processor consistency, *write followed by read* ordering is relaxed, but the memory system isn't necessarily coherent. Processor consistency is usually used however to refer to a coherent system. In that same paper, Gharachorloo et al. defined an even weaker

1. https://en.wikipedia.org/wiki/MSI_protocol

2. https://en.wikipedia.org/wiki/Dekker%27s_algorithm

(more relaxed consistency model called release consistency. (See next page).

Relaxed Consistency

A relaxed consistency model is one that doesn't require an memory operation orderings in order to consider a computation valid.

Release consistency is an example of a weak consistency model that relaxes all orderings, but introduces two synchronization primitives called **acquire()** and **release()** to replace the FENCE primitive from TSO/x86. These new

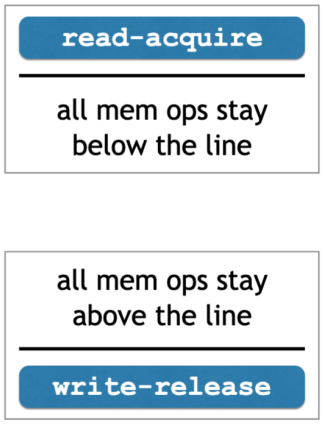

Fig. 5. M*ostly true* graphic of release acquire semantics [5]

primitives work intuitively with a programmer's notion of synchronizing a program. Anyone who has programmed a multiprocessor has probably used locks or mutexes. In release consistency, The acquire() primitive can be though of as a lock() operation, and release() as unlock(). If we acquire() some variable, then any operation *after* that acquire in program order cannot execute until the acquire() finishes. This can be thought of as waiting at a lock(). If we release() some variable, it insures that all operations that happened *before* it in program order are finished before the release is executed. This can be though of as releasing a lock when finished with a critical section. In Fig. 6 you see that *aquire*(), which is a read operation (R), cannot have any memory operations move ahead of it, while *release()* which is a write operation (W) can not have any memory operations move behind it. The bottom left box explains that on

Fig. 6. Multiprocessor with store buffers [5]

a single stream of instructions, an *aquire()* may jump ahead of a *release*().

Notice that to *aquire()* a variable, a processor doesn't need to have completed everything before the *aquire()* yet, and to release() a variable, the processor could have already tarted working on operations after the release().

We saw earlier that Dekker's Algoithm for mutual exclusion can be properly programmed on a x86 machine on by using FENCE()s. Similarly, any algorithm can be made sequential consistent with *aquire()* and *release()* primitives.

Conclusion

	Programmer	Compiler	Hardware
Strict Consistency	What most/novice programmer expects	No optimizations possible	Global ordering / clock required! No OOO, latency hiding difficult!
Sequential Consistency	Least astonishing Typically assumed for cache coherence	Disaster! Almost all optimizations are illegal, no reordering!	Only one outstanding request! No OOO!
Processor Consistency	Sometimes unexpected behavior (membar); however, locks (RMWs) work	May now reorder loads across stores, potential left	Allows for FIFO store buffers & multiple outstanding requests
Relaxed Consistency (WC,RC,EC)	Very hard (membars where needed)	Sweet, sweet freedom!	Allows for unordered, coalescing SBs & OOO CPUs

Fig. 7. Multiprocessor with store buffers [5]

In conclusion, there are dozens of different consistency models that have been used in practice over the years. Sequential consistency is the most intuitive and easiest to reason with, as well as prove properties about. On the other hand, with sequential consistency, performance is dramatically slowed, and its restrictions don't even protect the most primitive read-modify-write instructions. We see that in practice consistency is becoming more and more relaxed

to make for fast programs. If the programmer doesn't want some crazy result, they have the option to use primitives such as FENCE() acquire() and release() which can give you sequential consistency or even (in the case of release consistency) mutually exclusive critical sections. Although programming using a relaxed consistency means the programmer needs to know how to place primitives, programming with multiprocessors almost always requires critical sections (such as two processors trying to execute a read-modify-write) where even sequential consistency cannot achieve a desired output without using higher level primitives.

ACKNOWLEDGEMENTS

The author would like to thank Dr. Gerald Moulds, as well as Aaron Hunter and Adam Ames, for all the support in making this paper possible.

REFERENCES

[1] Krste Asanović, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[2] Edsger W. Dijkstra. *Hierarchical Ordering of Sequential Processes*, page 198–227. Springer-Verlag, Berlin, Heidelberg, 2002.

[3] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. *SIGARCH Comput. Archit. News*, 18(2SI):15–26, May 1990.

[4] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers C-28*, 9:690–691, September 1979.

[5] Heiner Litz. Lecture slides. *CSE226 W20*.

[6] A. Meixner and D. J. Sorin. Dynamic verification of memory consistency in cache-coherent multithreaded computer architectures. In *International Conference on Dependable Systems and Networks (DSN'06)*, pages 73–82, 2006.

[7] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. X86-tso: A rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53(7):89–97, July 2010.

[8] Daniel J. Sorin, Mark D. Hill, and David A. Wood. A primer on memory consistency and cache coherence. *Synthesis Lectures on Computer Architecture*, 6(3):1–212, 2011.