# CACHE CONSISTENCY AND SEQUENTIAL CONSISTENCY

**James R. Goodman**
**goodman@cs.wisc.edu**
**Computer Sciences Department**
**University of Wisconsin-Madison**
**1210 W. Dayton St.**
**Madison, WI 53705**

*This Technical Report was originally prepared in February 1989 for a tutorial for the IEEE Futurebus Working Group. It was circulated with the Futurebus minutes and later circulated to the IEEE Scalable Coherent Interface (SCI) Working Group, where it has been referenced as Technical Report Number 61, March 1989. It appears here unchanged except for typographical corrections and minor clarifications.*

## Memory Consistency and Multiprocessor Synchronization

A von Neumann processor has an implied order in which memory accesses are logically executed. For a single processor, *program order* implies that for any memory location, a read to that location will return the value most recently written to it. For a multiprocessor, however, an additional requirement may be placed on the memory system: the order in which memory operations occur may be observed by other processors to achieve implicit synchronization. Consider the following example of such use of memory ordering:

```
Process 1:
     a = f();
     barrier = 0;

Process 2
     while (barrier)
          ;    /* wait */
     b = g(a);
```

Initially, `barrier` is non-zero. Process 2 is using `barrier` to guarantee that `a` has been set to the value returned by `f()`. If Process 2 reads `barrier` as non-zero it assumes that Process 1 has not yet completed the computation of `f()`. Thus memory is used implicitly to synchronize a producer and a consumer.

While the synchronization function is essential for a shared-memory system, it is only occasionally that the memory is used in this way. For example, all references during the computation of `f()` are probably not even shared. Even if they are, it is unlikely that a program would want to use the values while Process 1 is computing `f()`.[1] So what really matters, in general, is that at a certain point in the execution of a program, a new value (or set of values) has been computed and after that point other processors will access the new values.

Such a point is usually synchronized through explicit synchronization mechanisms, such as a *semaphore* or a *barrier*[Jord78]. It is generally regarded as good programming technique to synchronize explicitly, rather than depending on the memory system to provide implicit synchronization. While there are well-known methods for implementing semaphores from a memory in which all reads and writes are observed to occur in a globally consistent way [Pete81], it is customary to provide atomic hardware primitives, such as Test-and-Set, Enqueue, Dequeue, or Compare-and-Swap, to simplify the synchronization operations. The use of such primitives, with the added guarantee that all pending memory operations are completed before such an atomic operation, provides several advantages. Among them are that code is easier to understand and, perhaps more importantly, easier to debug. More efficient hardware implementations are also possible.

## Strong Consistency

We have identified three levels of consistency that might be guaranteed by a parallel computer. The strongest and most restrictive is called *sequential consistency* by Lamport,[2] with the

[1]There are asynchronous SOR algorithms, for example, that converge faster by working with the “latest value” of a variable.

following definition:

> A multiprocessor is said to be *sequentially consistent* if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

A system that adheres to this level of consistency is said to be a *strongly ordered* system [DuSB88]. This definition implicitly excludes parallel write operations. While conceptually clean, its cost in performance is likely to be high for a large-scale parallel machine.

## Weak Consistency

The weakest form of consistency only guarantees that accesses to a given memory location are strongly ordered. Thus writes to different memory locations may be observed to occur in different order by different processors. This is in fact sometime called simply *cache consistency*: each read of a memory location is guaranteed to obtain the most recently written value. In practice, this is of course inadequate, since no guarantee of synchronization is possible with such a system. Thus some stronger form of ordering is provided conditionally, only for certain variables. These typically are explicit synchronization variables, such as that specified in a Test-and-Set instruction. An implementation technique for guaranteeing consistency at a specified point is the *fence* primitive [BrMW85]. When a fence operation is initiated by a processor, execution is blocked until all pending write (and possibly, read) operations have completed. If this operation is applied at the time of a synchronization operation only it is know as a *weakly ordered* system [DuSB88]. The use of a fence can potentially result in higher performance because write operations emanating from a single processor can be overlapped with reads and ensuing writes, except when a fence is encountered. Also, truly parallel writes can occur. Unfortunately, it is necessary for the programmer to state explicitly whenever a fence must be inserted, though such a requirement can be inferred, for example, whenever a Test-and-Set instruction is encountered. Problems arise, of course, when such points are not explicitly identified. For example, the IBM System/370 architecture includes a Test-and-Set instruction, but no Unset instruction, whose function can nominally be achieved with a simple write operation. The result is that code written for the 370, including MVS, will not execute correctly on a weakly ordered system [Brya89].

## Processor Consistency

There is an intermediate level of consistency, stronger than weak ordering, but weaker than strong ordering, that guarantees correct behavior in all but pathological cases, and permits substantially higher performance than strong ordering. We call it *processor ordering*, or *processor consistency*.

> A multiprocessor is said to be *processor consistent* if the result of any execution is the same as if the operations of each individual processor appear in the sequential order specified by its program.

Thus the order in which writes from two processors occur, as observed by themselves or a third processor need not be identical, but writes issuing from any processor may not be observed in

[2]L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs", IEEETC-28, 9, September 1979.

any order other than that in which they are issued.

It is possible to construct a situation in which processor ordering fails, but to date we have been unable to identify a single application for such code. An example is the following:

```
Process 1:
     a = 1;
     if (b == 0) kill Process 2;

Process 2:
     b = 1;
     if (a == 0) kill Process 1;
```

Initially, `a` and `b` are zero. If the memory system guarantees strong ordering, both processes can never be killed. But with processor ordering it is possible.

A system with processor ordering may also implement fence operations in conjunction with synchronization operations, though it is our belief that such operations are generally unnecessary. We note that many processors that appear be strongly ordered in fact are only processor ordered. For example, any processor that prefetches operands *or instructions* is not strongly ordered. For processors that are capable of prefetching operands, the compiler may actually introduce incorrect behavior by scheduling the code if a program depends on something more than weak ordering. This is very difficult to detect for any program that requires strong ordering (not just processor ordering). The Digital VAX 8800, in particular, guarantees processor consistency, and includes warnings that strong ordering is not guaranteed[FuKH87].

## Choosing and Guaranteeing the Appropriate Level of Consistency

In a system where a single ordering of write operations is universally observed, processor consistency is guaranteed. This is true for any system where there is a single path connecting any two processors (*e.g.*, a bus), if write notification signals (including cache invalidations) are propagated in order, *i.e.*, writes may never pass other writes (they may pass read requests). However, if there is more than a single path connecting two processors (*e.g.*, parallel buses), *even if the path for a given memory location is unique*, processor consistency is not easily guaranteed. A straightforward way to guarantee processor consistency is to limit the number of outstanding writes to one, thereby guaranteeing that all write operations emanating from the processor will be observed in program order.[3] This limit implies some kind of completion signal for every write operation. Such a mechanism is also required for a system supporting only weak ordering for determining when the fence operation may unblock the processor.

Given the requirement of an acknowledgement for every write issuing from any processor, we now have the framework for defining the levels of consistency in a uniform way. Each processor keeps a count of the number of write operations that have been initiated but not yet

[3]This method is actually slightly stronger than processor consistency. Notice, for example, that a single write before the if statements for both processors in the above example will now guarantee correct behavior. With processor consistency, this would only be true if both processors wrote to the same location.

acknowledged. Then weak ordering can be guaranteed by implementing a fence operation, which blocks the processor from reading or writing anything until the count is zero. Processor ordering can be guaranteed by blocking the processor from writing whenever the count is non-zero. Strong ordering can be guaranteed by blocking the processor from *reading or writing* whenever the count is non-zero.

Such policies could be changed under processor control, allowing the programmer to choose the appropriate level of consistency. This approach allows a nice separation of mechanism (the counter) from policy (the event ordering).

| | Weak | Processor | Strong |
|---|---|---|---|
| Read | ∞ | ∞ | 0 |
| Write | ∞ | 1 | 0 |
| Fence | 0 | 1 | (0) |

Count for which, if exceeded, processor must be blocked.

## References

[BrMW85] W.C. Brantley, K.P. McAuliffe, and J. Weiss, “RP3 Processor-Memory Element,” *1985 International Conference on Parallel Processing*, (August 1985), pp. 782-789.

[Brya89] Ray Bryant, *Private Communication*, January 1989.

[DuSB88] M. Dubois, C. Scheurich, and F. A. Briggs, “Memory access buffering in multiprocessor,” *Proceedings of the 13th International Symposium on Computer Archtiecture*, June 1986, pp. 434-442.

[FuKH87] J. Fu, J.B. Keller, and K.J. Haduch, “Aspects of the VAX 8800 C box design,” *Digital Technical Journal*, No. 4, (February 1987), pp. 41-51.

[Jord78] H. F. Jordan, “A special purpose architecture for finite element analysis,” *Proceedings of the 1978 International Conferecne on Parallel Processing*, pp. 263-266, 1978.

[Pete81] G. L. Peterson, “Myths about the mutual exclusion problem,” *Information Processing Letters*, Vol. 12, No. 3, (June 1981), pp. 115-116.