International Journal of Enhanced Research in Management & Computer Applications, ISSN: 2319-7471
Vol. 2 Issue 8, Sept.-Oct., 2013, pp: (5-11), Available online at: **www.erpublications.com**

Memory Consistency Models

Akshay Sharma[1], Deepak Basora[2], Ankur Sharma[3]
Computer Science department, Dronacharya College of Engineering, Gurgaon, India

Abstract: The memory consistency model for a shared-memory multiprocessor specifies the behaviour of memory with respect to read and write operations from multiple processors. *Relaxed models* that impose fewer memory ordering constraints offer the potential for higher performance by allowing hardware and software to overlap and reorder memory operations. Many of the previously proposed models either fail to provide reasonable programming semantics or are biased toward programming ease at the cost of sacrificing performance. The optimizations enabled by relaxed models are extremely effective in hiding virtually the full latency of writes in architectures with blocking reads. We evaluate all the consistency models and the comparison for the weak consistency model and release consistency model, the performance benefits of exploiting relaxed models based on detailed simulations of realistic parallel applications. We believe that the combined benefits in hardware and software will make relaxed models universal in future multiprocessors, as is already evidenced by their adoption in several commercial systems.

Keywords: Multiprocessors, Optimizations, Ordering Constraints, Semantics, Simulations.

Introduction

Parallel architectures provide the potential for achieving substantially higher performance than traditional uniprocessor architectures. The key differentiating feature among multiprocessors is the mechanisms used to support communication among different processors. By utilizing the fastest available microprocessors, multiprocessors are increasingly becoming a feasible and cost-effective technology even at small numbers of processing nodes.

Multiprocessors with a single address space, such as shared-memory architectures, make the entire memory accessible to all processors and allow processors to communicate directly through read and write operations to memory. The single address space abstraction greatly enhances the programmability of a multiprocessor. In comparison to a message-passing architecture, the ability of each processor to access the entire memory simplifies programming by reducing the need for explicit data partitioning and data movement. The single address space also provides better support for parallelizing compilers and standard operating systems. These factors make it substantially easier to develop and incrementally tune parallel applications. Since shared-memory systems allow multiple processors to simultaneously read and write the same memory locations, programmers require a conceptual model for the semantics of memory operations to allow them to correctly use the shared memory. This model is typically referred to as a memory consistency model or memory model. To maintain the programmability of shared-memory systems, such a model should be intuitive and simple to use [1]. Consistency models place specific requirements on the order that shared memory accesses (events) from one process may be observed by other processes in the machine. More generally, the consistency model specifies what event orderings are legal when several processes are accessing a common set of locations [2].

The memory system model is more complex because the definitions of “last value written”, “subsequent loads” and “next store” become unclear when there are multiple processors reading and writing a location. Furthermore, the order in which shared memory operations are done by one process may be used by other processes to achieve implicit synchronization. Unfortunately, architecture and compiler optimizations that are required for efficiently supporting a single address space often complicate the memory behavior by causing different processors to observe distinct views of the shared memory. It is one of the challenging problems in designing a shared-memory system is to present the programmer with a view of the memory system that is easy to use and yet allows the compiler optimizations that are necessary for efficiently supporting a single address space.

There have been numerous attempts at defining an appropriate memory model for shared memory systems. Several memory consistency models have been proposed in the literature: examples include sequential consistency, processor consistency, and weak consistency. This thesis focuses on providing a balanced solution that directly addresses the trade-off between programming ease and performance, providing automatic portability across a wide range of implementations.

Memory consistency models

In computer science, **consistency models** are used in distributed systems like distributed shared memory systems or distributed data stores (such as a file systems, databases, optimistic replication systems or Web caching). The system supports a given model if operations on memory follow specific rules. The data consistency model specifies a contract between programmer and system, wherein the system guarantees that if the programmer follows the rules, memory will be consistent and the results of memory operations will be predictable. Verifying sequential consistency is not decidable in general, even for finite-state cache-coherence protocols [3]. Consistency models define rules for the apparent order and visibility of updates, and it is a continuum with tradeoffs [4].

Sequential consistency

The most commonly assumed memory consistency model for shared memory multiprocessors is *sequential consistency*, formally defined by Lamport [5]:

> ***Definition:*** *[A multiprocessor system is 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.*

Sequential consistency is one of the consistency models used in the domain of concurrent programming (e.g. in distributed shared memory, distributed transactions, etc.).

The system provides sequential consistency if every node of the system sees the (write) operations on the same memory part (page, virtual object, cell, etc.) in the same order, although the order may be different from the order as defined by real time (as observed by a hypothetical external observer or global clock) of issuing the operations.

There are two aspects to sequential consistency:

- Maintaining program order among operations from individual processors, and
- Maintaining a single sequential order among operations from all processors.

The latter aspect makes it appear as if a memory operation executes *atomically* or *instantaneously* with respect to other memory operations. Sequential consistency provides a simple view of the system to programmers as illustrated in Fig 1, where P1…Pn shows the processors.

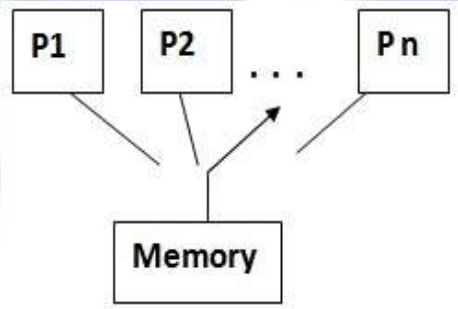

Figure 1: Conceptual representation of sequential consistency

Conceptually, there is a single global memory and a switch that connects an arbitrary processor to memory at any time step. Each processor issues memory operations in program order and the switch provides the global serialization among all memory operations. The sequential consistency is weaker than strict consistency (which would demand that operations are seen in order in which they were actually issued, which is essentially impossible to secure in distributed system as deciding global time is impossible) and is the easiest consistency model to understand, since a system preserving that model is behaving in a way expected by an instantaneous system.

Relaxed Memory Consistency Models

The original specifications of these models emphasized system optimizations enabled by the models; we retain the system-centric emphasis in our descriptions of this section. We focus on models proposed for hardware shared-memory systems; relaxed models proposed for software-supported shared memory systems are more complex to describe. The

basic idea behind relaxed memory models is to enable the use of more optimizations by eliminating some of the constraints that sequential consistency places on the overlap and reordering of memory operations. While sequential consistency requires the illusion of program order and atomicity to be maintained for all operations, relaxed models typically allow certain memory operations to execute out of program order or non-atomically. The degree to which the program order and atomicity constraints are relaxed varies among the different models.

Fig 2 shows the relaxations allowed by the memory models. We categorize relaxed memory consistency models based on two key characteristics: (1) how they relax the program order requirement, and (2) how they relax the write atomicity requirement.

With respect to program order relaxations, we distinguish models based on whether they relax the order from a write to a following read, between two writes, and finally from a read to a following read or write. In all cases, the relaxation only applies to operation pairs with different addresses.

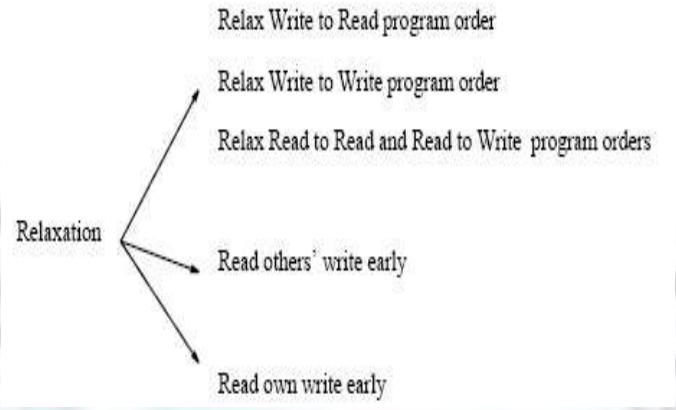

Fig 2: Relaxations allowed by memory models.

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

Figure 3: Simple categorization of relaxed models.

Fig 3 provides an overview of the models. A tick indicates that the corresponding relaxation is allowed by straightforward implementations of the corresponding model. It also indicates that the relaxation can be detected by the programmer (by affecting the results of the program) except for the following cases. The "Read Own Write Early" relaxation is not detectable with the SC, WO, Alpha, and PowerPC models. The "Read Others' Write Early" relaxation is possible and detectable with complex implementations of RCsc.

Processor Consistency

Processor consistency is the first model that we consider where the multiple-copy aspects of the memory are exposed to the programmer. To relax some of the orderings imposed by sequential consistency, Goodman introduces the concept of processor consistency [6]. Processor consistency requires that writes issued from a processor may not be observed in

any order other than that in which they were issued. However, the order in which writes from two processors occur, as observed by themselves or a third processor, need not be identical. Processor consistency is weaker than sequential consistency; therefore, it may not yield 'correct' execution if the programmer assumes sequential consistency. However, Goodman claims that most applications give the same results under the processor and sequential consistency models. Specifically, he relies on programmers to use explicit synchronization rather than depending on the memory system to guarantee strict event ordering. Goodman also points out that many existing multiprocessors satisfy processor consistency, but do not satisfy sequential consistency. The description given in [6] does not specify the ordering of read accesses completely. We have defined the following conditions for processor consistency.

Condition 1: Conditions for Processor Consistency

- *Before a LOAD is allowed to perform with respect to any other processor, all previous LOAD accesses must be performed, and*
- *Before a STORE is allowed to perform with respect t0 My Other processor, all previous accesses (LOADS and STORES) must be performed.*

The above conditions allow reads following a write to bypass the write. To avoid deadlock, the implementation should guarantee that a write that appears previously in program order will eventually perform.

Fig 4 shows the representation for this model where $P_1 \ldots P_n$ are the processors and $M_1 \ldots M_n$ are the memories. The conceptual system consists of several processors each with their own copy of the entire memory. By modeling memory as being replicated at every processing node, we can capture the non-atomic effects that arise due to presence of multiple copies of a single memory location.

Since the memory no longer behaves as a single logical copy, we need to extend the notion of read and write memory operations to deal with the presence of multiple copies. Read operations are quite similar to before and remain atomic. The only difference is that a read is satisfied by the memory copy at the issuing processor's node (i.e., read(R) from Pi is serviced by Mi). Write(W) operations no longer appear atomic, however.

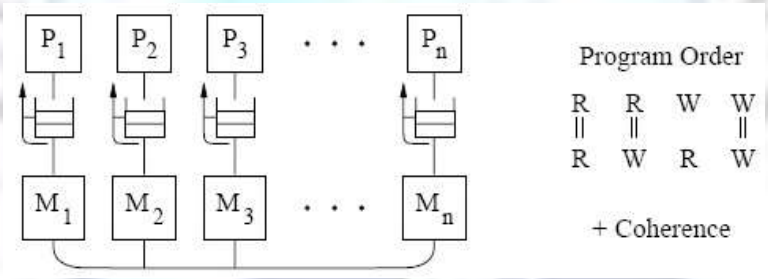

Figure 4: The processor consistency model.

Each write operation conceptually results in all memory copies corresponding to the location to be updated to the new value. Therefore, we model each write as a set of *n* sub-operations, W(1)…: W(n), where *n* is the number of processors, and each sub-operation represents the event of updating one of the memory copies (e.g., W(1) updates the location in M1). Since we need to refer to the sub-operations of a write, we will also refer to a read as a single atomic sub-operation for uniformity (denoted as R(i) for a read from Pi). We use the double lines between a pair of operations to denote the fact that operations may no longer be atomic and that all sub-operations of the first operation must complete before any sub-operations of the second operation.
Since write operations are no longer atomic, processor consistency imposes an additional constraint on the order of write sub-operations to the same location. This requirement is called the coherence requirement.

Weak Consistency

The weak consistency is one of the consistency models used in the domain of the concurrent programming (e.g. in distributed shared memory, distributed transactions etc.). The protocol is said to support weak consistency if:

- All accesses to synchronization variables are seen by all processes (or nodes, processors) in the same order (sequentially) - these are synchronization operations. Accesses to critical sections are seen sequentially.
- All other accesses may be seen in different order on different processes (or nodes, processors).
- The set of both read and write operations in between different synchronization operations is the same in each process.

Therefore, there can be no access to a synchronization variable if there are pending write operations. And there cannot be any new read/write operation started if system is performing any synchronization operation. The opposite of weak consistency is strong consistency, where parallel processes can observe only one consistent state.

A weaker consistency model can be derived by relating memory request ordering to synchronization points in the program. The intuition behind weak ordering is that most programs are written using synchronization operations to coordinate memory operations on different processors and maintaining program order at such synchronization operations typically leads to correct outcomes for the program. Figure 5 shows the representation for weak ordering.

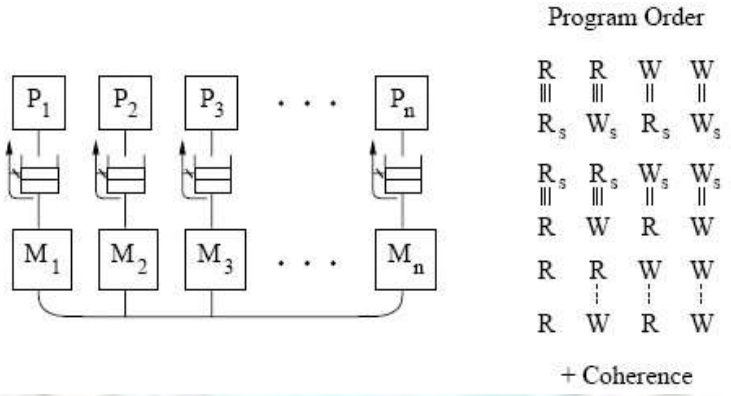

Fig 5: The weak ordering model.

As an example, consider a processor updating a data structure within a critical section. If the computation requires several STORE accesses and the system is sequentially consistent, then each STORE will have to be delayed until the previous STORE is complete. But such delays are unnecessary because the programmer has already made sure that no other process can rely on that data structure being consistent until the critical section is exited, Given that all synchronization points are identified, we need only ensure that the memory is consistent at those points. This scheme has the advantage of providing the user with a reasonable programming model, while permitting multiple memory accesses to be pipelined. The disadvantage is that all synchronization accesses must be identified by the programmer or compiler.

The weak consistency model proposed by Dubois et 01. [7] is based on the above idea. They distinguish between ordinary shared accesses and synchronization accesses, where the latter are used to control concurrency between several processes and to maintain the integrity of ordinary shared data. The conditions to ensure weak consistency are given below.

Condition 2: Conditions for Weak Consistency

- *before an ordinary LOAD or STORE access is allowed to perform with respect to any other processor, all previous synchronization accesses must be performed*
- *before a synchronization access is allowed to perform with respect to any other processor, all previous ordinary LOAD and STORE accesses must be performed, and*
- *Synchronization accesses are sequentially consistent with respect to one mother.*

Release consistency

Release consistency is one of the consistency models used in the domain of the concurrent programming (e.g. in distributed shared memory, distributed transactions etc.). Release consistency extends the ideas in weak ordering by further distinguishing among memory operations. There are two kinds of coherence protocols that implement release consistency:

- *eager*, where all coherence actions are performed on release operations,[8] and
- *lazy*, where all coherence actions are delayed until after a subsequent acquire[9]

Tread Marks is an application of lazy release consistency [10].

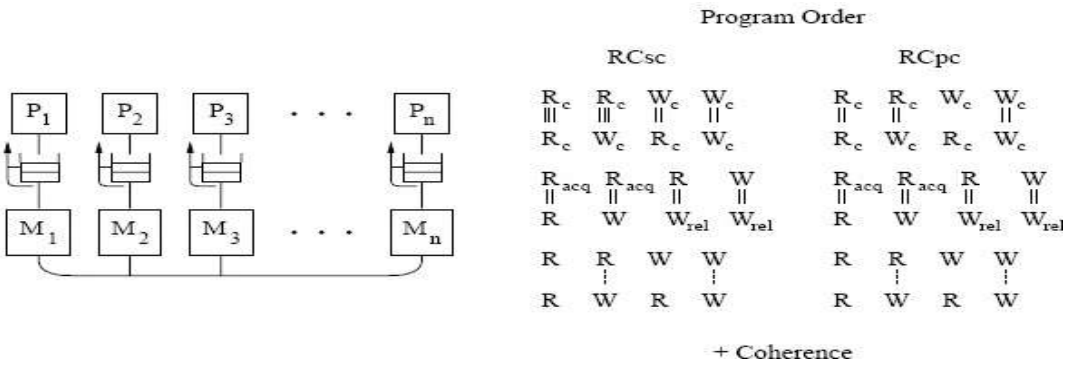

Figure 6: The release consistency (RC) models.

Fig 6 provides the representation for release consistency. There are two flavours of release consistency that differ in the order that is maintained among synchronization operations. The first flavour maintains sequential consistency among synchronization operations and is referred to RCsc, while the second flavour maintains processor consistency among such operations and is called RCpc. Except for the program order requirements, these two models are identical. Considering the conceptual system, release consistency is similar to weak ordering except a read is allowed to return the value of a write (to the same location) that is in the buffer. The value requirement for both RCsc and RCpc is the same as that for PC. Finally, both models obey the coherence requirement.

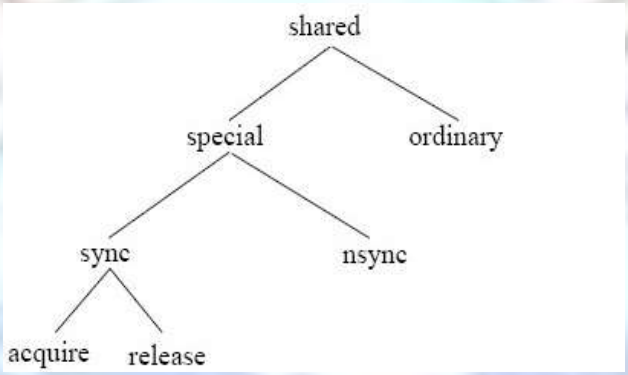

Figure 7: Distinguishing operations for release consistency.

Compared to weak ordering, release consistency provides further distinctions among memory operations. Fig 7 pictorially depicts this classification of memory operations. Operations are first distinguished as *ordinary* or *special*. These two categories loosely correspond to the data and synchronization categories in WO. Special operations are further distinguished as *sync* or *nsync* operations. Syncs intuitively correspond to synchronization operations, whereas nsyncs correspond to asynchronous data operations or special operations that are not used for synchronization. Finally, sync operations are further distinguished as *acquire* or *release* operations. Intuitively, an acquire is a read memory operation that is performed to gain access to a set of shared locations (e.g., a lock operation or spinning for a flag to be set). A release is a write operation that is performed to grant permission for accessing a set of shared locations (e.g., an unlock operation or setting of a flag).

Before issuing a write to a memory object a node must acquire the object via a special operation, and later release it. Therefore the application that runs within the operation acquire and release constitutes the critical region. The system is said to provide release consistency, if all write operations by a certain node are seen by the other nodes after the former releases the object and before the latter acquire it.

The main idea behind release consistency is that read and write synchronization operations have different ordering requirements. The purpose of a write synchronization used as a release is to signal that previous accesses are complete and it does not have anything to say about ordering of accesses that follow it. Therefore, while the completion of the release is delayed until previous memory operations in program order complete, memory operations after a release are not delayed for the release to complete. Similarly, the completion of a read synchronization used as an acquire need not be delayed for previous memory operations to complete. This is because acquire is not giving permission to any other process to read or write the previous pending locations. This allows for extra reordering and overlap of memory operations across acquires and releases.

Conclusion

This paper presented the background information for the memory consistency. We motivated the need for a memory consistency model for the purpose of specifying the behaviour of memory operations, and introduced the notion of sequential consistency as an intuitive model for shared-memory multiprocessors. Next considered were some of the architecture and compiler optimizations that are desirable in multiprocessors, and showed that the majority of these optimizations violate the semantics of sequential consistency. This led to the discussion of alternative memory models that enable a higher degree of performance by relaxing some of the constraints imposed by sequential consistency. Choosing among these models requires considering fundamental trade-offs between programmability, portability, implementation complexity, and performance.

References

[1]. http://www.hpl.hp.com/techreports/Compaq-DEC/WRL-95-9.pdf.

[2]. http://eecs.vanderbilt.edu/courses/eece343/papers/p15-gharachorloo.pdf.

[3]. Shaz Qadeer (August 2003), "Verifying Sequential Consistency on Shared-Memory Multiprocessors by Model Checking". IEEE Transactions on Parallel and Distributed Systems.

[4]. Todd Lipcon (2009-06-11). "Design Patterns for Distributed Non-Relational Databases". http://static.last.fm/: last.fm. Retrieved 2011-03-24. "A consistency model determines rules for visibility and apparent order of updates.

[5]. Leslie Lamport. How to make a multi processor computer that correctly executes multiprocess programs. IEEE Transactions on Computers, C-28(9):690–691, September 1979.

[6]. James R. Goodman. Cache consistency and sequential consistency. Technical Report no. 61, SC1 Committee, March 1989.

[7]. Michel Dubois, Christoph Scheurich, and Fayb Briggs. Memory access buffering in multiprocessors. In Proceedings of the 13th Annual International Symposium on Computer Architecture, pages 434-442, June 1986.

[8]. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors by Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy published in ISCA '90 Proceedings of the 17th annual international symposium on Computer Architecture.

[9]. Lazy release consistency for software distributed shared memory by Pete Keleher, Alan L. Cox, and Willy Zwaenepoel published in Proceeding ISCA '92 Proceedings of the 19th annual international symposium on Computer architecture.

[10]. Tread Marks: distributed shared memory on standard workstations and operating systems by Pete Keleher, Alan L. Cox, Sandhya Dwarkadas and Willy Zwaenepoel published in WTEC'94 Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference.