Memory Consistency Models
for Shared Memory Multiprocessors and DSM Systems

Jelica Protić, Igor Tartalja, and Milo Tomasević
Department of Computer Engineering, School of Electrical Engineering
University of Belgrade
POB 816, 11000 Belgrade, Yugoslavia
E-Mail: {eproticj, etartalj, etomasev}@ubbg.etf.bg.ac.yu

Abstract - The use of systems with multiple processors that support shared memory programming paradigm is rapidly increasing nowadays. Possible buffering, pipelining, and optimization of shared memory accesses, as well as the existence of multiple copies of shared variables in these systems, may cause specific implications that can not be understood just as an intuitive extension of an uniprocessor memory model. Therefore, the memory consistency model formally specifies the memory system behavior to be expected by the programmer. This paper reveals the essence of several memory consistency models: sequential, processor, weak, release (with eager and lazy implementation), and entry. It also provides definitions and a set of examples that underline differences between particular models. Results of several performance evaluation studies are also discussed.

I. INTRODUCTION

In multiple-processor systems with shared address space, various orderings of memory accesses issued by individual processors and various ways of interleaving memory accesses of different processors are possible. Memory consistency model defines a set of allowable access orderings to the shared locations. It also imposes specific requirements on the order in which shared memory accesses from one processor may be observed by other processors in the system. The most restrictive memory consistency models (sequential and processor consistency) treat accesses to all shared memory locations equally. More relaxed models (weak, release, lazy release and entry consistency) make difference between ordinary and synchronization accesses, imposing specific restrictions on synchronization accesses.

Weakening of consistency semantics represents one of the key sources for performance improvements of shared memory multiprocessors [16], [15] and DSM systems [13], since it allows overlapping and reordering of memory accesses. However, it puts an additional burden on the programmer to be aware of the restrictions that implemented model imposes. Anyway, different applications have different requirements on data consistency, and it is not easy to satisfy all of them on the system level. Therefore, involvement of the programmer is sometimes inevitable in resolving those problems on the application level.

Several efforts in surveying the field under consideration have been already made (e.g. [12], [7]). Some of those papers contain only a brief overview of memory consistency models, while the others do not cover more recent models. This survey is intended to be unique in its exhaustiveness and to provide simple illustrations in order to help in understanding differences between appropriate models.

II. MEMORY CONSISTENCY MODELS: AN OVERVIEW

A. Sequential Consistency vs. Processor Consistency

Sequential and processor consistency models do not make any difference between ordinary accesses and synchronization accesses. They represent very restrictive, but intuitively understandable abstraction of memory system behavior.

Basic condition and requirements for sequential consistency are defined in [11] as follows:

Condition: A multiprocessor 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.

Requirement 1: Each processor issues memory requests in the order specified by its program.

Requirement 2: Memory requests from all processors issued to an individual memory module are serviced from a single FIFO queue. Issuing a memory request consists of entering the request on this queue.

Implementation of sequential consistency suggested in [8] implies that all writes are sent to the write buffer (processor stalls only when the buffer is full). However, all pending writes must be performed before read, and then, processor stalls for read to perform. The implication is that all nodes view the same sequential interleaving of reads and writes [12].

Conditions for sequential consistency hold in the majority of bus-based shared memory multiprocessors, as well as in early distributed shared memory (DSM) systems, such as IVY and Mirage.

Processor consistency was firstly introduced in [9] by the following definition:

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.

The order in which memory operations can be seen by different processors need not be identical, but the
sequence of writes issued by each processor must be observed by all other processors in the same order. Unlike sequential consistency, processor consistency implementations allow reads to bypass pending writes in queues from which memory requests are serviced.

Precise definition of conditions for processor consistency is given in [7]. Examples of systems that guarantee processor consistency are Digital VAX 8800 and DSM system PLUS.

To illustrate the difference between sequential and processor consistency, we will analyze the example in Figure 1.

<table>
<thead>
<tr>
<th>Processor 1</th>
<th>Processor 2</th>
<th>Processor 3</th>
<th>Processor 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>a: A=1</td>
<td>c: C=1</td>
<td>d: PRINT A,B,C</td>
<td>e: PRINT A,B,C</td>
</tr>
<tr>
<td>b: B=1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Possible interleavings of write operations are: abc, acb, and cab. Order of values for variables according to those interleavings are:

<table>
<thead>
<tr>
<th>for abc</th>
<th>for acb</th>
<th>for cab</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABC</td>
<td>ABC</td>
<td>ABC</td>
</tr>
<tr>
<td>000</td>
<td>000</td>
<td>000</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>101</td>
</tr>
<tr>
<td>110</td>
<td>101</td>
<td>111</td>
</tr>
<tr>
<td>111</td>
<td>111</td>
<td></td>
</tr>
</tbody>
</table>

Assuming that PRINT is an atomic operation, only values 010 and 011 can not be printed at all for both sequential and processor consistency, since this would imply that sequential operations of processor 1 A=1 and B=1 are swapped. If processors 3 and 4 print values that do not exist in the same column of the table, it means that they view different sequential orders of writes in the system, so the system can be processor consistent, and it is not sequentially consistent. An example for this situation is if one of processors 3 and 4 prints 110, and the other prints 101 or 001. Another example is if one prints 001, and the other 110 or 100. If processors 3 and 4 print any combination of values from the set {000,100,101,111} the system can be sequentially consistent.

Figure 1: Comparison of sequential and processor consistency

B. Introducing Synchronization Accesses:

Weak and Release Consistency

Weak consistency distinguishes between ordinary accesses and synchronization accesses. It requires that memory becomes consistent only on synchronization accesses. This approach was introduced in [4], and more precise conditions to ensure the weak consistency are enlisted in [7].

1. before an ordinary LOAD or STORE access is allowed to perform with respect to any other processor, all previous synchronization accesses must be performed, and
2. 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
3. synchronization accesses are sequentially consistent with respect to one another

Therefore, in this model a synchronization access must wait for all previous accesses to complete, while ordinary reads and writes must wait only for previous synchronization accesses. Many variants of weak consistency were proposed [5],[1] but few were implemented, such as one described in [14] and used in SPARC architecture by Sun Microsystems.

Release consistency further differentiates synchronization accesses to acquire and release, in order to enable the protection of ordinary shared accesses between an acquire-release pair. Acquire synchronization access represents an attempt to gain an access to the shared resource. Release sets the shared resource free and it is associated with a write synchronization access.

According to the definition stated in [12], system is release consistent if:
1. Before an ordinary access on a given processor is allowed to perform, all previous acquires on that processor must be performed, and
2. Before a release on a given processor is allowed to perform, all previous ordinary reads and writes on that processor must be performed, and
3. Synchronization accesses are processor consistent

<table>
<thead>
<tr>
<th>Processor 1</th>
<th>Processor 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>acquire(s1)</td>
<td>acquire(s2)</td>
</tr>
<tr>
<td>A=1</td>
<td>E=1</td>
</tr>
<tr>
<td>B=1</td>
<td>F=1</td>
</tr>
<tr>
<td>release(s1)</td>
<td>release(s2)</td>
</tr>
<tr>
<td>C=1</td>
<td>G=1</td>
</tr>
<tr>
<td>acquire(s2)</td>
<td>acquire(s1)</td>
</tr>
<tr>
<td>D=1</td>
<td>H=1</td>
</tr>
<tr>
<td>a: PRINT E,F,G,H</td>
<td>b: PRINT A,B,C,D</td>
</tr>
<tr>
<td>release(s2)</td>
<td>release(s1)</td>
</tr>
</tbody>
</table>

Acquire(s1)-release(s1) pair is guarding writes to A and B, and b PRINT operation is also guarded by s1. Therefore, it is not possible, both for weak and release consistency, that values of A and B printed by b are different. The same applies to E and F, and a PRINT with respect to s2.

For the weak consistency model, it is not possible that a prints G=0 and H=1, or that b prints C=0 and D=1. This is because of the fact that all previous ordinary accesses must become visible before the next acquire is allowed to perform (e.g. C=1 and G=1), and it is not possible that ordinary accesses after acquire are performed before acquire (e.g. D=1 and H=1). Anyway, this result is possible for release consistency, since ordinary accesses before an acquire do not have to become visible on acquire, but only on the following release.

For weak consistency, it is not possible that both processors print 1100, because synchronization accesses have to be sequentially consistent, so both processors must observe the same interleaving of synchronization accesses. However, this situation is possible for release consistency, where synchronization accesses only have to be processor consistent.

Figure 2: Comparison of weak and release consistency
Theoretical background concerning release consistency is explained in details in [7], as well as the advantages of this model over the weak consistency. Unlike in the case of weak consistency, ordinary accesses following a release do not have to wait for release to complete, and an acquire synchronization access need not wait for previous ordinary accesses to be performed. Therefore, optimizations are possible where acquire is overlapped with the previous ordinary accesses, and release is overlapped with the following ordinary accesses.

Different implementations of release consistency can be found in Dash and Munin DSM systems. Dash implementation (RC) propagates remote ordinary accesses as they occur, so the pipelining of those operations is possible. On the other side, in Munin implementation, named Eager Release Consistency (ERC) all changes are delayed till release, when they can be merged in order to minimize the length of messages.

C. Further refinements: Lazy Release Consistency and Entry Consistency

A specific variant of release consistency, lazy release consistency, is presented in [10]. Instead of propagating modifications to the shared address space on each release (like in release consistency), modifications are further postponed until the next relevant acquire. It means that only writes associated to the chain of preceding synchronization operations on the same lock have to be propagated on acquire. Lazy release consistency was implemented in DSM system Treadmarks.

Finally, entry consistency is a further improvement of release consistency. This model requires that each ordinary shared variable has to be associated with a synchronization variable using language level declaration. Modification to an ordinary shared variable is postponed to the next acquire of its associated synchronization variable. Since other non-associated variables do not have to be modified at that moment, number and length of messages are reduced. Entry consistency was introduced, and its implementation in DSM system Midway is described in [2].

---

<table>
<thead>
<tr>
<th>Processor 1</th>
<th>Processor 2</th>
<th>Processor 3</th>
<th>Processor 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>acquire(s1)</td>
<td>acquire(s2)</td>
<td>PRINT A,B,C,D acquire(s1)</td>
<td>PRINT A,B,C,D acquire(s2)</td>
</tr>
<tr>
<td>A:=2</td>
<td>C:=2</td>
<td>PRINT A,B,C,D release(s1)</td>
<td>PRINT A,B,C,D release(s2)</td>
</tr>
<tr>
<td>B:=1</td>
<td>D:=1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A:=A-B</td>
<td>C:=C-D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>release(s1)</td>
<td>release(s2)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Initial values: A=0, B=0, C=0, D=0

In order to compare various implementations of release consistency (Dash-like implementation RC, Munin-like implementation ERC, and TreadMarks-like implementation LRC) let us assume that critical region on the processor 3 is performed after critical region on the processor 1, and critical region on the processor 4 is performed after critical region on the processor 2. For RC, it is possible that the first PRINT on processor 3 writes the value 2 for the variable A, and it is also possible that the first PRINT on processor 4 writes the value 2 for the variable C. Both PRINT operations are not protected, and new values of variables are sent by processors 1 and 2 to other processors as they occur. Unlike this implementation, ERC will never print value 2 for A or C, since all updates are merged on release, and temporary values generated between acquire-release pairs are not forwarded to other processors before the release occurs. It is important to notice that new values are sent to all processors in the system on release, no matter if they compete for the same lock, or if the new value will be needed or not.

The difference of both RC and ERC to the LRC is that in LRC modifications are not propagated on release, but on the following acquire on the same lock. Therefore, modifications to A and B will be forwarded to processor 3 only, and it will not happen before acquire (so the first PRINT on processor 3 must write zeroes for A and B, and the second PRINT must write ones). For the same reason, modifications to C and D will be forwarded to processor 4 only.

Finally, let us assume that for entry consistency model, ordinary variable A is protected by synchronization variable s1, and C by s2. In this case, if entry consistency is applied, only modification to A will be propagated on the next acquire to s1, and not the modification to B. Therefore, the second PRINT on processor 3 can write different values for A and B and processor 4 for C and D, respectively.

---

Figure 3: Comparison of release and entry consistency

III. PERFORMANCE CONSIDERATIONS

Globally, weakening of memory consistency model increases the performance potential, but also complicates the programming model and, in the case of hardware implementations, also increases hardware complexity. Several comparative performance studies have addressed those issues. In [8] sequential, processor, weak, and release models are compared to the base model, where processor immediately stalls on each read and write and wait for the operation to perform. Simulations were performed assuming DASH architecture with write-through first level caches. Three benchmark applications were used (MP3D, LU, and PTHOR), and processor utilization was considered as the main performance indicator in the analysis. Maximum performance improvements achieved by relaxing consistency model were 41% for MP3D, 29% for PTHOR, and 11% for LU. The highest relative performance gains were obtained when moving from sequential to processor consistency. Applications with lower synchronization rates (MP3D and LU) performed comparably for weak and release consistency. Surprising result is that the PTHOR application with high synchronization rate performed better for processor than for weak model, since the loss caused by stalling for synchronization was greater than gain of pipelining writes between synchronization accesses.

Another comparative analysis of the same consistency models using analytical approach based on stochastic Petri nets is presented in [3]. The results are in agreement with those reported in [8]. Qualitatively similar results were also obtained in another simulation...
analysis presented in [17]. This study is based on multiprocessor with processors and their write-back caches connected by Omega network. It compares two variants of sequential consistency, weak consistency, and release consistency. The results show that the performance improvements of relaxed models are up to 35%, compared to an aggressive implementation of sequential consistency.

Various protocols implementing release consistency (eager update, eager invalidate, lazy update, lazy invalidate, and a combination called lazy hybrid) were compared in [6] for a set of applications from SPLASH benchmark suite using simulation method. This study showed that performance gains highly depend on the application. Lazy protocols were superior for medium-grained application Water, where lazy hybrid and lazy invalidate are 2.5 times faster than eager invalidate and 25% faster than eager update, when software overhead is not taken into account.

Comparative implementation analysis of the Midway system [2] for 4 processors connected by a fast ATM network performing matrix-multiply showed speedup of about 12% when moving from release consistency to entry consistency.

IV. CONCLUSION

In this paper we have described existing memory consistency models in a uniform manner, including formal definitions as well as comprehensive examples that illustrate the essence of the models. We have also emphasized the performance implications of relaxing memory consistency semantics. However, the actual gains reported in several studies do not precisely distinguish the effects of the model itself from the effects of underlying architecture, network characteristics, buffer lengths, cache implementations, software overheads, etc. Application characteristics also highly affect possible performance improvement. Nevertheless, the higher the memory latency and the lower the hit ratio of local accesses, the greater the importance of relaxing memory consistency model in order to achieve better performance of the overall system. Therefore, the interest for those issues will be definitely increasing, especially in attempts to build parallel systems on the top of networks of workstations.

ACKNOWLEDGMENTS

A lot of time is spent in fruitful discussions with Professor Veljko Milutinović on these issues. Authors are very thankful for his constant encouragement and support. This work was partially supported by FNRS.

REFERENCES