Performance Analysis of Shared Memory Model in a Multiprocessor Environment

Pankaj Gupta*, Garima Verma**
**Computer Science Department, BIT's Mesra. pgupta@bitmesra.ac.in*
***Computer Science Department, MTU Noida. garima.verma@globalinst.in*

ABSTRACT

As we know that transition from single processor to multi-processors presents some challenges for shared memory design and implementation these are: keeping caches coherent and maintaining memory consistency.

The problem of cache coherency occurs when any one processor tries to access a region of shared memory which has been modiûed in another processor's private cache, but not yet written back to main memory, which results in a read to memory which is out of date.

Memory Consistency Model is needed for a shared memory system to define the ordering in which reads and writes must be performed, both relative to each other in the same program and to other reads and writes by another program on another processor. The programmer must know the memory consistency model used by the hardware in order to ensure program correctness.

Though there are many solutions to the problem, each memory consistency model has its own advantages and disadvantages and it is not always clear which protocol works the best for a particular system and application. Here we presented the unifying framework by which we describe, understand and compare the sequential and total store order memory model using MATLAB. Performance of the models here is based on the memory cycle count.

Keywords **– Shared Memory, Multiprocessor, Memory Consistency Models, SC, TSO, Machine cycle.**

I. INTRODUCTION

Although the performance of microprocessor is growing at an exponential rate, but as suggested by Moore's law, there is always demand for large computing capacity beyond what can readily be provided by a single processor, hence we need a multiprocessor system.

II. SHARED MEMORY MULTIPROCESSOR

A popular architecture for multiprocessors is the shared-memory architecture where all processors share the same memory space. By sharing the same memory, processors can communicate to coordinate their execution. In Shared-Memory architecture processors can coordinate their execution by basic memory read and writes operations.

In a programmer's point of view, a simple shared-memory multiprocessor could be modelled as shown in Figure 1. Here all the processors are connected to the main memory through a switch which grants the access to only one arbitrarily selected processor at a time. The connected processor will then perform one memory operation, according to the order specified by its program, and disconnect itself from the memory.

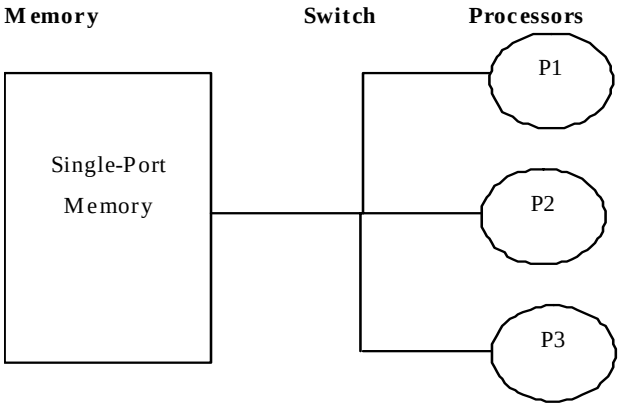


Fig. 1 A Programmers Model of a simple Shared Memory Multiprocessor

III. MEMORY CONSISTENCY MODELS

One of the major attributes that describe a shared-memory multiprocessor is its memory consistency model, which is basically a contract between hardware and software regarding the semantics of memory operations. The simple abstract model shown in Figure 1 is an example of a memory consistency model, called *Sequential Consistency* (*SC*), which was first defined by Lamport in 1979.

The SC model is already restrictive enough that it violates many optimizations used by most microprocessors today. Therefore, some relaxations have been made to the SC model to allow for better performance. This attempt results in relaxed memory consistency models [11, 9, 14, 12, 10, 5, and 7] which become less intuitive, however, making them difficult for both hardware designers and software writers to understand.

IV. RELAXED CONSISTENCY MODELS

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.

Relaxed consistencies have broadly categorized the various models based on how they relax the program order constraint. The first category of models includes the IBM-370 [3], Sun SPARC V8 total store ordering (TSO) [2, 4], and processor consistency (PC) [12, 16] models, all of which allow a write followed by a read to execute out of program order. The second category includes the Sun SPARC V8 partial store ordering (PSO) [2, 4] model, which also allows two writes to execute out of program order. Finally, the models in the third and last category extend this relaxation by allowing reads to execute out of program order with respect to their following reads and writes. These include the weak ordering (WO) [15], release consistency (RC) [12, 16], Digital Equipment Alpha (DEC Alpha) [5, 6], Sun SPARC V9 relaxed memory order (RMO) [7], and IBM PowerPC (PowerPC) [8, 9] models.

In this Paper we are just focusing on Sequential Consistency and Total store ordering Memory model. Hence in the rest of the paper we will talk only about SC and TSO memory models.

V. SEQUENTIAL CONSISTENCY

A natural way to define a memory model for multiprocessors is to base it on the sequential semantics of memory operations in uniprocessors. An intuitive definition would require executions of a parallel program on a multiprocessor to behave the same as some interleaved execution of the parallel processes on a uniprocessors.

Such a model was formally defined by Lamport as sequential consistency [13] (abbreviated as SC). The definition below assumes a multiprocessor consists of several correctly-functioning uniprocessors, referred to as sequential processors [Lam79] that access a common memory. Another implicit assumption is that a read returns the value of the last write to the same location that is before it in the sequential order described above.

Definition: Sequential Consistency

[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.]

A conceptual SC system can be modelled as shown in Figure1 where processors are connected to the shared memory through a switch. The following point describes the behaviour that appears to programmers:

1. The switch connects the memory to only one processor at a time, and the memory services only one operation at a time, thus, making each

memory operation to appear to execute atomically with respect to other memory operations. The order in which memory operations are serviced at the memory is called the **memory order**. All processors observe the same view of this memory order.

2. When granted the access to the memory, a processor executes its memory operations in the order specified by its program, called the **program order**.
3. A read operation returns the value from the most recent write operation (according to the memory order) to the same memory location.
4. Switch arbitration is fair so that memory operations from all processors are eventually serviced.

Here we emphasize on the appearance of the system because the actual systems need not strictly implement what is illustrated in the conceptual model nor do they have to maintain the stated ordering requirements at all times, as long as any execution result produced by these systems can be explained as if it were produced by a hypothetical, strict SC implementation. (An execution result refers to the values returned by the read operations in the execution.) In other words, a system may aggressively perform beyond what is allowed by the memory model provided that programmers will not notice it doing so. For example consider Figure 2(a) where

P1	**P2**	**P1**	**P2**
S[A]=1		S[A]=0	S[B]=0
S[B]=2		S[A]=1	S[B]=1
S[C]=3		L[B]=0	L[A]=0
	L[C]=3	L[B]=1	L[A]=1
	L[A]=1		
	L[B]=2		
(a) SC		(b) Not SC	

Fig. 2 Examples of execution results.

- A strict SC implementation can produce the shown execution result if all memory operations from processor P1 are performed before those from P2.
- However, this same execution result can still be produced even if S[A] = 1 and S[B] = 2 (as well as L [A] and L [B]) are executed out of order.
- Although such reordering would indeed violate the second condition of the SC model above regarding maintaining the program order, it does not lead to an execution result that is not producible by the strict SC implementation.
- Therefore, programmers can choose to believe that such reordering did not occur. Hence, the memory order that appears to programmers needs not correspond to the actual order.

Unfortunately, determining (both statically and dynamically) whether or when it is safe to deviate from the strict definition is difficult. This effectively rules out several hardware optimizations which are otherwise applicable to sequential uniprocessors such as overlapping accesses to different memory locations and using write buffers to hide write latency. Compiler optimizations affecting memory operations also become severely restricted; complex code analysis has to be performed to determine when such optimizations would be safe (e.g., other processors must not be able to observe the reordering of memory operations, if any). Oftentimes, such analysis has to remain conservative and produce code which is less efficient but it is guaranteed to be correct in all possibilities.

Finally, note that caches are not included in the SC model since they should be transparent to software. A cache coherence protocol governs the propagation of each newly written value among caches such that write operations to the same memory location are visible to all processors in the same order. Cache coherence is necessary but not sufficient for Sequential Consistency because the SC model further requires that operations to all memory locations must appear to all processors in the same order. As an

example, the execution result shown in Figure 2(b) is cache coherent but not sequentially consistent.

- It is cache coherent because load operations for each memory location observe the values changing in the same order as they are written by the store operations.
- However, there is no memory order that can produce such a result while remaining sequentially consistent.
- Consider, for example, L [B] = 0 performed by processor P1. The fact that it reads value 0 means that it must be performed after P2's S [B] = 0 has written the value 0 to the location, but before P2's S [B] = 2 overwrites that with a new value.
- Because operations from P1 (as well as P2) must appear to happen in the program order, both S [A]'s preceding L [B] = 0 must have already been performed before L [B] = 0 and memory location A will hold the most recent value, 1.
- This disallows the first L [A] performed by P2, which happens later, to see the already overwritten value, 0.

An execution (or the result of an execution) of a program is sequentially consistent if there exists at least one execution on a conceptual sequentially consistent system that provides the same result (given the same input and same initial state in memory). Otherwise, the execution violates sequential consistency.

VI. TOTAL STORE ORDERING (TSO)

The total store ordering (TSO) [2, 4] model is one of the models proposed for the SPARC V8 architecture. Figure 3 shows the representation for this model. The TSO model always allows a write followed by a read to complete out of program order. All other program orders are maintained. If a read matches (i.e., is to the same location as) a write in the write buffer, the value of the last such write in the buffer that is before it in program order is forwarded to the read. Otherwise, the read returns the value in memory, as in the SC.

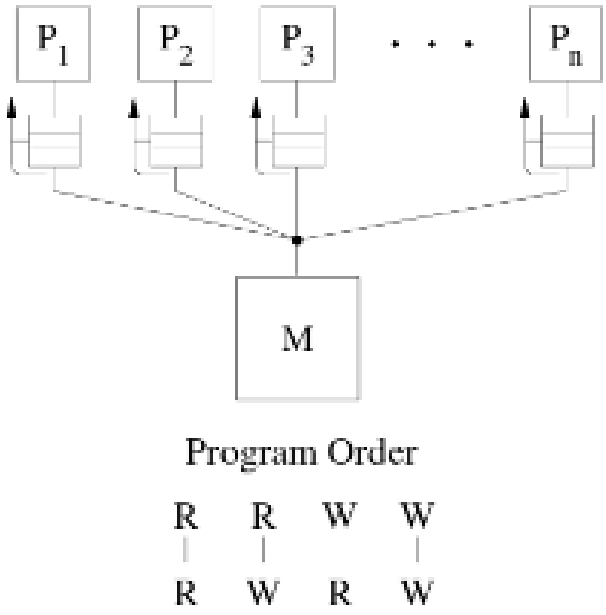


Fig. 3 The Total Store Order (TSO) memory model [2]

Because the value of a write in the buffer is allowed to be forwarded to a read, the value requirement for TSO is different from the simple memory value requirement for SC. If we consider operations as executing in some sequential order, the buffer-and-memory value requirement requires the read to return the value of either the last write to the same location that appears before the read in this sequence or the last write to the same location that is before the read in program order, whichever occurs later in the sequence. This requirement captures the effect of forwarding the value of a write in the buffer in case of a match.

P1	**P2**	**P1**	**P2**
a1: S[A]=1	a2: S[B]=1	a1: S[A]=1	a2: S[B]=1
b1: u=L[A]	b2: v=L[B]	b1: S[C]=1	b2: S[C]=2
c1: w=L[B]	c2: x=L[A]	c1: u=L[C]	c2: v=L[C]
d1: w=L[B]	d2: x=L[A]		
	(a)		**(b)**

Fig. 4 Example program segments for the TSO model

Figure 4 presents a couple of program segments that illustrate the differences between the TSO and SC models. First consider the program segment in Figure 4(a).

- Under the SC model, the outcome (u, v, w, x) = (1, 1, 0, 0) is disallowed.
- However, this outcome is possible under TSO because reads are allowed to bypass all previous writes, even if they are to the same location.
- Therefore the sequence (b1, b2, c1, c2, a1, a2) is a valid total order for TSO.
- Of course, the value requirement still requires b1 and b2 to return the values of a1 and a2, respectively, even though the reads occur earlier in the sequence than the writes.
- This maintains the intuition that a read observes all the writes issued from the same processor as the read.

Figure 4(b) shows a slightly different program segment. In this case, the outcome (u, v, w, x) = (1, 2, 0, 0) is not allowed under SC, but is possible under TSO.

VII. COMPARING SC AND TSO MEMORY MODEL

With the help of above discussion we can compare both the SC & TSO memory model on the following basis:

- **Execution:** SC executions are proper subset of TSO executions; all SC executions are TSO execution, while some TSO executions are SC execution, some are not.
- **Implementation:** Implementations follow the same rule, i.e. SC implementation is a proper subset of TSO implementation.

More generally a memory consistency model Y is strictly more relaxed (weaker) than a memory consistency model X if all X execution are also Y execution, but not vice-versa. If Y is more relaxed than X, then it follows that all X implementation are also Y implementation. It is also possible that two memory consistency models are incomparable because both allow execution precluded by the other.

As we know that performance for any memory model depends on the 3P's concept given by Sarita.V.Adve [1], for the case of SC & TSO model the 3P works as follows:

- **Programmability:** SC is the most intuitive. TSO is close because it acts like SC for common programming idioms.
- **Performance:** For simple cores TSO can offer better performance than SC, but difference can be made small with speculation.
- **Portability:** SC is widely understood, while TSO is widely adopted.
- Hence from the above discussion we came to know that relaxed models (TSO) are most commonly used than the strict sequential consistency model. Although the programming is complex for relaxed models but the performance of relaxed models (TSO) are better than the sequential consistency model. We came to this conclusion after evaluating the performance of both the SC model and the TSO model with the module developed using MATLAB for performance evaluation of shared memory model.
- We originally developed this module for performance analysis of shared memory model in a multiprocessor environment.

VIII. SIMULATION RESULTS

Figure 5 shows our GUI for performance analysis of shared memory model in a multiprocessor environment. Here we had taken two pop-up menus one is for processor 1 and the other is for processor 2. This means that here the main memory is shared only by two processors.

For simplicity of discussion we assumed that all the operations defined in the each processor are single machine cycle operation.

We considered that each processor having sufficient private memory for their own operations. Also we considered that both share only 16 bytes of the shared memory.

Here we used 8 Programs for simulation and analysis and they are numbered as 1 to 8. For each processor there are four programs for simulation and analysis. In our module, for processor 1 program 1 to 4 is fixed and for processor 2 programs 4 to 8 are fixed. Also we assumed that the each processor is same in the architecture.

- ALU1 ALU Operations 1
- ALU2 ALU Operations 2
- COMPR Compare or decision Operation
- RDPRV Read from Private Memory
- WRPRV Write to Private Memory
- RDSHR Read from shared Memory
- WRSHR Write to Shared Memory

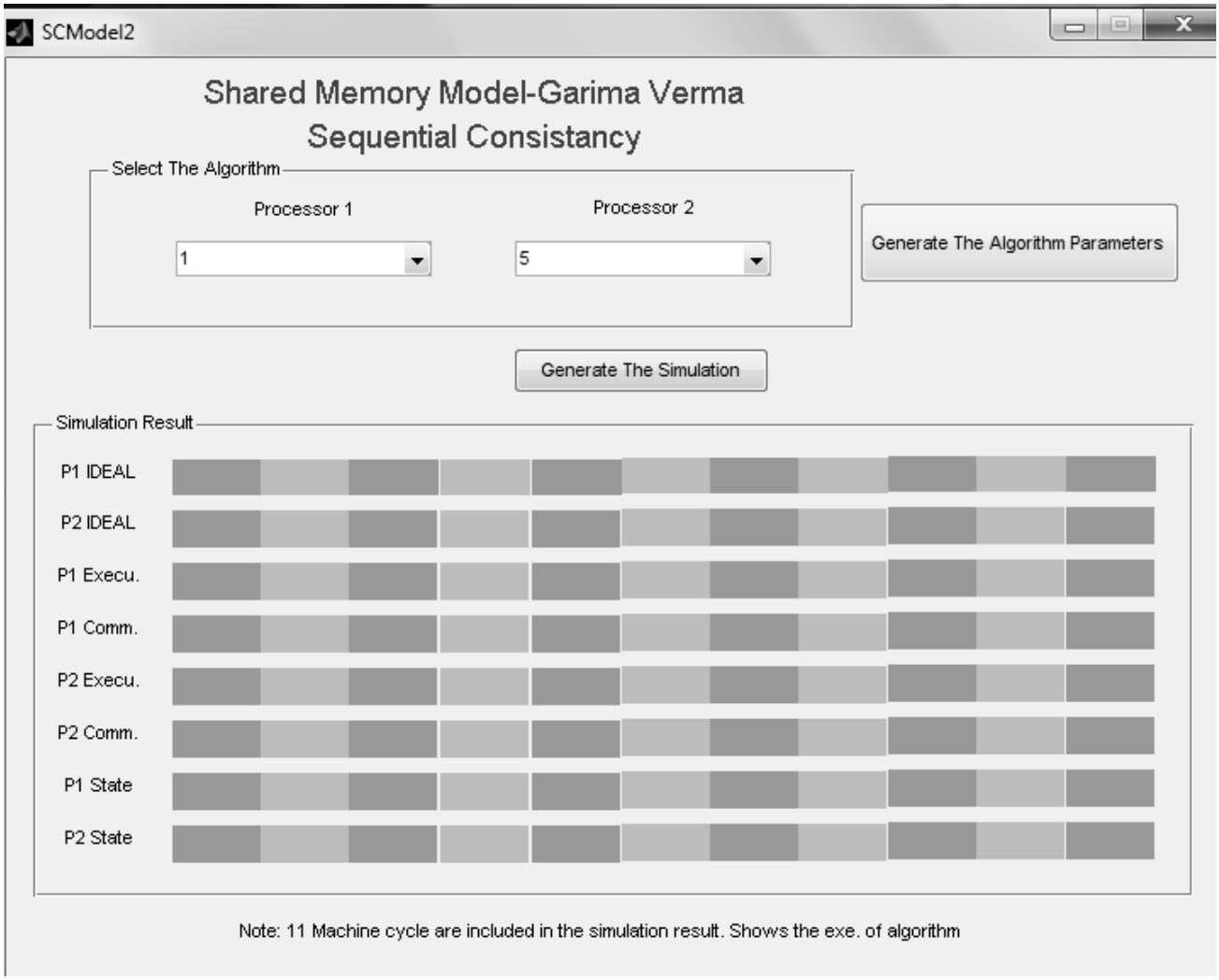


Fig. 5 GUI for performance analysis of Sequential Consistency Memory ModelFor simulation we assumed that processors are having limited operations like-

Operations involved in each Test Program are given below:

Program 1: ALU1;
RDSHR;
RDPRV;
WRSHR;

Program 2: ALU2;
WRSHR;
COMPR;
WRPRV;

Program 3: ALU2;
WRSHR;
RDSHR;
ALU2

Program 4: COMPR;
RDSHR;
WRSHR;
RDPRV;

Program 5: RDPRV;
WRSHR;
ALU1;
WRPRV;

Program 6: ALU1;
WRPRV;
RDPRV;
COMPR;

Program 7: ALU1;
COMPR;
WRSHR;
RDSHR;

Program 8: RDSHR;
WRSHR;
WRPRV;
RDPRV;

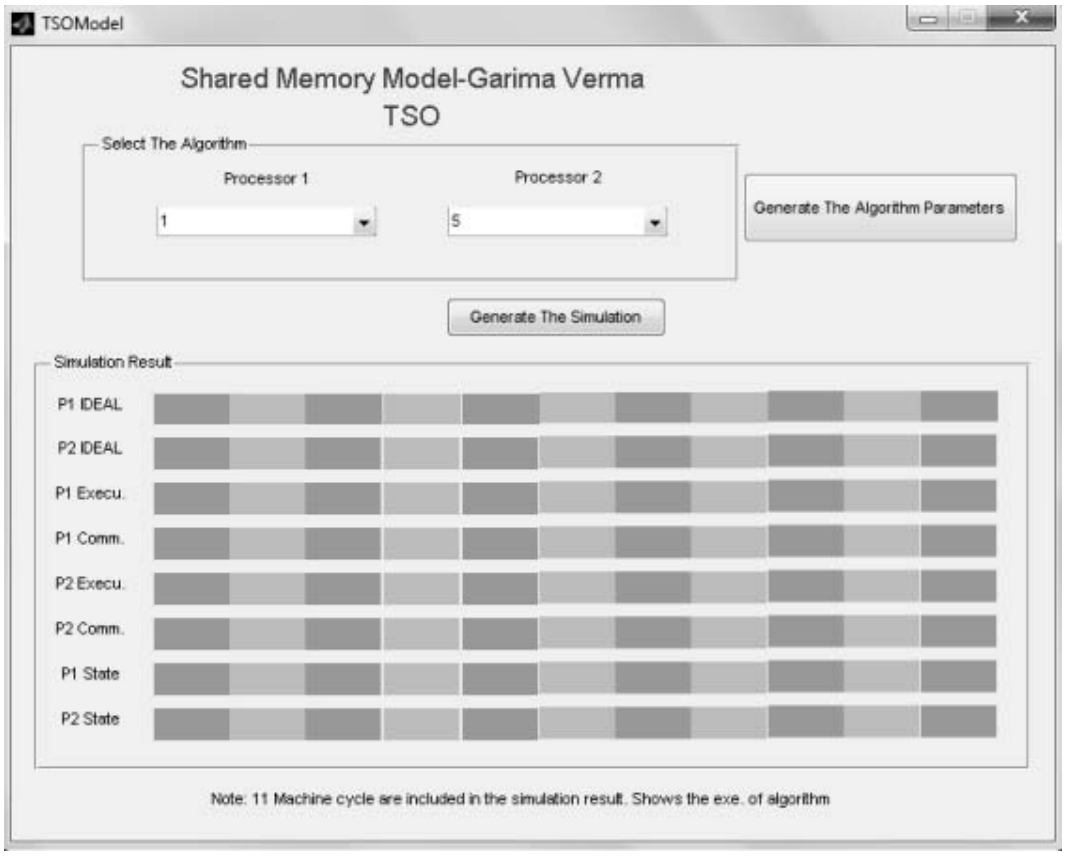
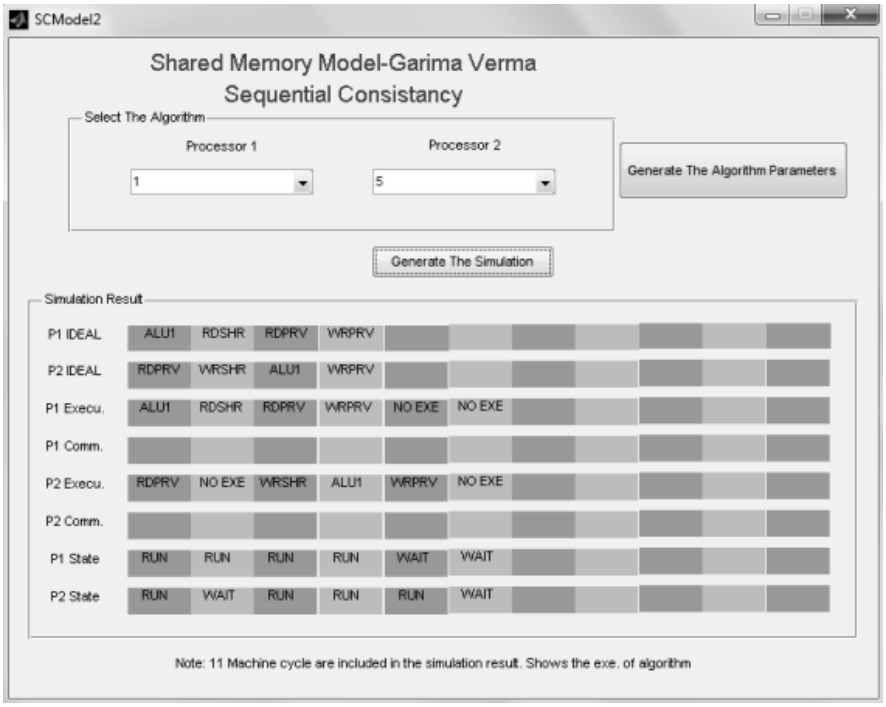


Fig. 6 GUI for performance analysis of Total Store Order Memory Model.

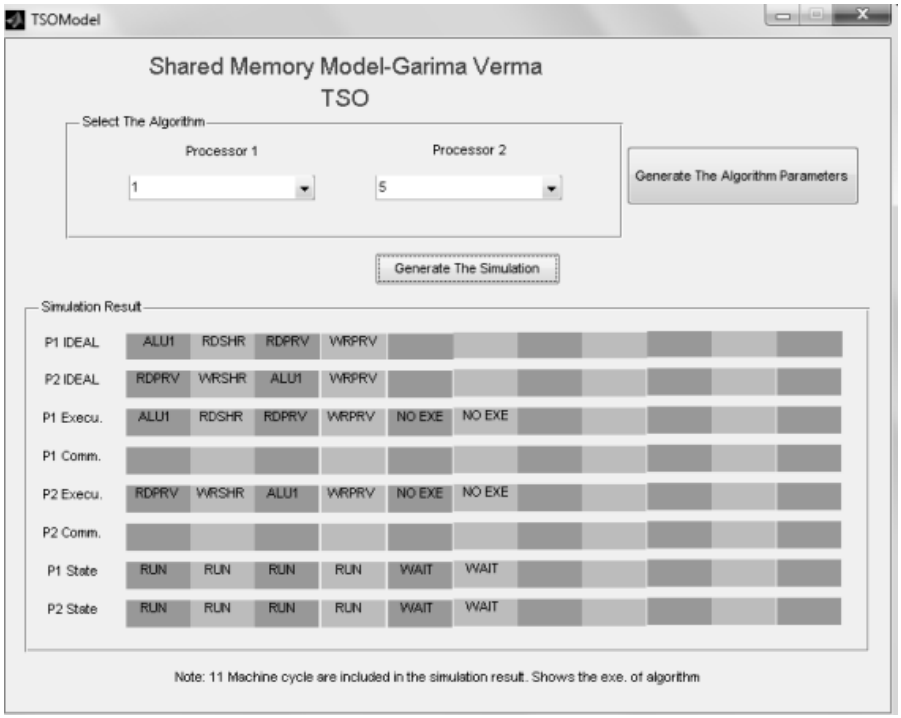
Selecting any pair of test program for processor 1(1, 2, 3, 4) and for processor 2 (5, 6, 7, 8) we can get the simulation results for performance analysis of SC Model and TSO Model based on the count of memory

cycle. If the count of memory cycles for any model is less than the other model for the same set of programs, it is said to have better performance.

Case 1: If we consider test program 1 for processor 1 and test program 5 for processor 2, then the simulation results for both the memory models are shown in the following snapshots:



The above snapshot shows the result for SC model, where both the processors complete their operations in 5 memory cycles by sharing the provided shared memory.



The above snapshot shows the result for TSO model, where both the processors complete their operations in 4 memory cycles by sharing the provided shared memory.

For program set 1 and 5 taken for processor 1 and processor 2 respectively we can say that for this scenario the TSO model is better than SC model.

For all other set of test programs the simulation results for both the SC and TSO model is shown in the following table:

Table 1: Simulation Results in tabular form

Test Program number for Processor 1	Test Program number for Processor 2	Memory Cycle Count by simulation result for SC Model	Memory Cycle Count by simulation result for TSO Model
1	5	5	4
1	6	4	4
1	7	4	4
1	8	5	4
2	5	5	5
2	6	4	4
2	7	4	4
2	8	5	5
3	5	6	5
3	6	4	4
3	7	5	4
3	8	6	5
4	5	6	4
4	6	4	4
4	7	5	5
4	8	6	4

IX. CONCLUSION

From the above table of simulation results it is clear by the count of memory cycles that in all scenarios the TSO memory model gives better performance over the SC memory model.

Hence it is the most adopted memory model we have.

X. FUTURE SCOPE

There are several directions for future research which may be classified into following categories:

1. Further research on algorithms, as I had taken small algorithms.
2. Further research on the model, as extension of this is possible.
3. The Model I developed can be applied to other memory models also.
4. Further research on related concepts.

REFERENCES

[1] Sarita V. Adve. *Designing memory consistency models for shared – memory multiprocessors*. PhD thesis, University of Wisconsin, 1993.

[2] Pradeep S. Sindhu, Jean-Marc Frailong, and Michel Cekleov. Formal specification of memory models. Technical Report CSL-91-11, Xerox Palo Alto Research Centre, December 1991.

[3] *IBM System/370 Principles of Operation*. IBM, May 1983. Publication Number GA22-7000-9, File Number S370-01.

[4] *The SPARC Architecture Manual*. Sun Microsystems Inc., January 1991. No. 800-199-12, Version 8.

[5] Richard L. Sites, editor. *Alpha Architecture Reference Manual*. Digital Press, 1992

[6] Richard L. Sites and Richard T. Witek, editors. *Alpha AXP Architecture Reference Manual*. Digital Press, 1995. Second Edition.

[7] David L. Weaver and Tom Garamond, editors. *The SPARC Architecture Manual*. Prentice Hall, 1994. SPARC International, Version 9.

[8] Cathy May, Ed Silha, Rick Simpson, and Hank Warren, editors. *The PowerPC Architecture: A Specification for a New Family of RISC Processors*. Morgan Kaufmann Publishers, Inc., 1994.

[9] Francisco Corella, Janice M. Stone, and Charles M. Barton. A formal specification of the PowerPC shared memory architecture. Technical Report Computer Science Technical Report RC 18638(81566), IBM Research Division, T.J. Watson Research Centre, January 1993.

[10] James R. Goodman. Cache consistency and sequential consistency. Technical Report 61, IEEE Scalable Coherent Interface Working Group, March 1989.

[11] Kourosh Gharachorloo. *MEMORY CONSISTENCY MODELS FOR SHARED-MEMORY MULTIPROCESSORS*. PhD thesis, Stanford University, 1995.

[12] K. GHARACHORLOO, D. LENOSKI, J. LAUDON, P. GIBBONS, A. GUPTA and J. HENNESSY, Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors, *Proc. 17th Annual Intl. Symp. on Computer Architecture*, May 1990, 15-26.

[13] L. LAMPORT, How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs, *IEEE Trans. on Computers C-28*, 9 (September 1979), 690-691.

[14] M. DUBOIS, C. SCHEURICH and F. A. BRIGGS, Synchronization, Coherence, and Event Ordering in Multiprocessors, *IEEE Computer 21*, 2 (February 1988), 9-21.

[15] C. SCHEURICH and M. DUBOIS, Correct Memory Operation of Cache-Based Multiprocessors, *Proc. Fourteenth Annual Intl. Symp. on Computer Architecture*, Pittsburgh, PA, June 1987, 234-243.

[16] K. GHARACHORLOO, A. GUPTA and J. HENNESSY, Two Techniques to Enhance the Performance of Memory Consistency Models, *Proc. Intl. Conf. on Parallel Processing*, 1991, I355-I364.