Performance Evaluation of Consistency Models Using a New Simulation Environment for SVMS

Salvador Petit, Julio Sahuquillo, and Ana Pont
Departamento de Informática de Sistemas y Computadores*
Universidad Politécnica de Valencia
Cno. de Vera s/n, 46071, Valencia (Spain)
{spetit, jsahuqui, apont}@disca.upv.es

Abstract

Shared virtual memory (SVM) systems are organizations of distributed shared memory systems (DSM). These systems offer a shared memory programming model that is more intuitive than the message passing paradigm. Other advantages include low hardware and maintenance costs. But the software consistency models implemented on these systems imply high communication latency due to the messages sent for maintaining memory coherence. To our knowledge, current performance evaluation studies are made using real systems.

The simulation tool used is an execution-driven simulator aimed at studying the behavior of memory consistency models, with the exception of those needing compiler modifications. It presents a cheap and flexible way to do performance studies and design efficient consistency approaches for SVM systems. This paper presents a performance evaluation comparison of two well-known consistency models using a new simulation environment for such systems, presenting results difficult to gather in a real system. These results show how the write operations.

Keywords: shared virtual memory, consistency models, networks of workstations, execution-driven simulators.

1. Introduction

Programming models traditionally used in Networks of Workstations (NOW) [21] usually focus on the message passing paradigm. This is because these models need a minimum physical level of integration that – in the most basic case – can be limited to a simple LAN. The main advantage offered by NOWs is easy maintenance as their nodes and network can be easily upgraded.

However, the traditional message passing programming model does not allow an appropriate use of current parallel code. In addition, loosely coupled architectures do not allow an easy implementation of more intuitive programming models, such as the shared memory that is commonly used in uniprocessor systems (multithreading). Symmetric Multiprocessors (SMP) [20] and Distributed Shared Memory (DSM) [19] are common architectures which implement physical parallelization in these kind of programs. The former offers low cost but reduced scalability and system maintenance facilities; the latter offers the mentioned benefits but at a high hardware and design cost.

There is no current low-cost alternative to cover the parallel architecture niche over 16 processors in SMP systems. A possible solution introduced by Li and Hudak in [1], is a Shared Virtual Memory system (SVM). As consequence of their software implementation, SVM systems offer a cheap way to implement DSM systems, by supporting the shared memory programming model on loosely coupled architectures.

Current results presented in the open literature regarding SVM systems are obtained from real systems. This is an inflexible way to design the whole system. Using simulators is a cheaper and more flexible way to handle the design. This paper uses a new simulation environment for DSM systems [23], specifically aimed at shared virtual memory systems (SVM) in NOWs. The tool simulates the detailed behavior of these systems, varying memory consistency models and local area networks (LAN). It is an execution-driven simulator [17, 18], that can make use of real loads such as the SPLASH-2 benchmark suite [9]. Using this new tool a performance evaluation study of Sequential Consistency (SC) and Lazy Release Consistency (LRC) models has been done. The

---

*This work has been partially supported by Generalitat Valenciana under Grant GV98-14-47
study compares their communication time, the number of messages transmitted and memory latencies.

The remainder of this paper is organized as follows: section 2 discusses the SVM systems to be simulated and the details of the memory consistency models found in the open literature; section 3 introduces the developed simulation environment; section 4 presents the performance evaluation study; and finally, section 5 presents the most interesting conclusions.

## 2. Shared Virtual Memory

These systems are implemented by the virtual memory management, which forms part of the operating system. They benefit from a more intuitive programming model (the shared memory model), and from loosely coupled hardware systems such as NOWs. Fault tolerance, maintenance and upgrading facilities can be achieved because nodes and networks are physically independent.

Nevertheless, the software implementation of these systems implies high communication latency for maintaining memory coherence. In addition, the large coherence unit (the page) increases the potential for false sharing. Some solutions have been proposed in the open literature [12,10,3,7,4,8,16] to reduce both latency and false sharing. We discuss some of the solutions and focus on relaxed memory consistency models to make a performance evaluation study.

### 2.1. Memory Consistency Models

The Sequential Consistency (SC) model [12] was the original consistency approach used in a SVM system [14]. This model ended up showing the problems mentioned above (high latency and false sharing); and consequent research has focused on new relaxed consistency models to reduce the number of coherence messages.

The Release Consistency (RC) model [10], was developed on the Munin system [2], and is a viable alternative for those parallel programs that are free of race conditions and have sufficient synchronization. The model only writes from one processor to another when the issuing processor reaches a release or an unlock instruction. This is achieved by making a copy or twin of the target page when the first write on the page is detected; then, the writing processor generates a table of differences, or diff, of each page modified at release time. Finally, the diff is sent to each node in the copyset.

It can be seen in figure 1 that a large number of messages is unnecessary to maintain the correct execution of the RC programs. For example, if the variables x and y are in the same page and shared by three processors, but only two of which update the x variable, then there is no need to send a diff to the third processor. Notice that false sharing can be avoided because it is supposed that the accesses are free of race conditions [15].

![Figure 1 – Communication in RC consistency model.](image-url)
The Lazy Release Consistency model (LRC) [3] relaxes RC by delaying coherence maintenance until the next acquire or lock. This model was used for the first time in the TreadMarks system [5]. This system uses an invalidation policy that even further reduces the presence of diffs in the network. For consistency maintenance, processor A maintains a list of the write notices it produces as well as those produced by other processors.
(transmitted through a release/acquire pair). These write notices are transmitted to the following processor B, that carries out an acquire on the semaphore liberated by A. Then B invalidates the corresponding pages. When B fails on some of the invalidated pages, it requests the diff(s) to update the page. Figure 2 shows that this method produces fewer transactions than under RC.

LRC needs to maintain a distributed garbage collection algorithm of diffs and write notices, since a node itself cannot be sure when to get rid of them. This normally coincides with the moment of arrival at a barrier point. This system necessarily reduces the available memory in the nodes (if the period between barriers is long) and increases the contention at the barrier times.

Home-based protocols such as HLRC (Home Lazy Release Consistency) [7] and AURC (Automatic Update Release Consistency) [4] do not need to store diffs because they select a home node for each page. This node picks up the diffs generated by the different nodes for the page. If a node fails in a page because it received a write notice, it then retrieves the whole page from the home. Special hardware is used under AURC to capture the individual writings on the shared pages and send them to the home. As Figure 3 shows, AURC avoids software overhead in the generation and application of diffs [11].

There are even more relaxed consistency models [8, 16]. They reduce software latency and reduce the number of messages on the network by changing the compiler (Entry Consistency Model) or program code (Scope Consistency Model). These last two models are currently outside of the scope of the existing simulator version.

3. **LIDE**

This section discusses the LIDE simulation environment that was developed from the two simulators, LIMES [18] and SIDE (initially called SMURPH) [22]. Figure 4 shows the block diagram of the proposed environment, and how the simulators connect to each other. LIMES allows parallel program execution and we use it to collect the memory references generated by the workloads. SIDE allows memory consistency model design, and injects the transactions needed for coherence maintenance.

LIMES is a cache memory simulator of SMP systems running on i486 compatible processors. It can be concluded from [6] that the number and composition of the memory references generated by these systems is similar to the data obtained from a RISC multiprocessor machine. Therefore, the memory references obtained from simulations of i486 multiprocessor systems can be extended for simulating other kinds of multiprocessor systems.

![Figure 4 – Block diagram of the LIDE simulation environment.](image-url)
A SMP memory model is very different from a DSM model – and even more different from a SVM model – although the differences are only expressed from a physical point of view (architectural design and temporary behavior). The references generated by distributed programs executed on these systems are equivalent. In other words, the logical behavior of a DSM system does not really differ from that of a SMP system.

SIDE is a heterogeneous environment for the simulation of networks and processes (state machines). It has two main advantages. Firstly, it offers an object-oriented programming model with defined classes for the management of several types of synchronization between processes (tails, signals, shared memory, messages, etc.). Secondly, it provides several examples of high performance networks (such as ATM). We have introduced in SIDE the code to simulate the SVM systems handled by the operating system. The code includes the consistency model (duly isolated), and the simulator of the physical network interconnecting the nodes. So, it is easy to change the physical network module without any, or just minimal, modification of the memory system code.

SIDE controls the simulation on a scheme of events and processes. The execution of different parts of the code for each process depends on the arrival of the expected events. Events are message arrivals, signals from other processes, arrival of a process to a particular state, timeouts generated by processes, etc.

LIDE uses LIMES to obtain the processor references, and it injects these in SIDE, which simulates the behavior of the memory and network systems. SIDE processes several references at the same time, and replies to LIMES when any reference is satisfied. Communication between both simulators is made by means of two UNIX named pipelines.

Because simulators uses different time scales, one of the main problems is to establish at run-time the convenient flow of events between LIMES and SIDE to assure the correct execution of the parallel benchmarks. The obvious solution to this problem is to perform the synchronization cycle-by-cycle. However, because of the independence of SIDE processes, it is impossible to know the global system state at any given moment. Thus, it is impossible to carry out the optimization used in LIMES to reduce the synchronization overhead each cycle. This solution introduces a huge overhead, which grows because the synchronization is performed by UNIX pipes. This can be tackled in a fast DSM system simulation because the memory service takes just a few cycles; but this time is much longer in a SVM system, and consequently, this method is not feasible.

A new synchronization scheme is used that does not need cycle-by-cycle synchronization. In the scheme, SIDE counts the real simulation time and LIMES is only aware of the correct order in which the references are satisfied. The working scheme is organized in the five steps discussed below:

1. The memory simulator (LIMES) waits for every processor to issue a memory reference.
2. The memory simulator sends to SIDE the memory reference and the number of processing cycles to be simulated. These cycles are independently counted in each processor as the amount of processing time elapsed since the previous memory reference.
3. With the information from 2 SIDE then simulates in a parallel way for each processor:
   a) The corresponding processing cycles.
   b) The service time for current memory reference.
4. SIDE replies to LIMES when one of the memory references is satisfied.
5. Return to the step 1.

Two different simulators are used by LIDE, so two different time-scales are available. LIDE takes its time-scale from SIDE, because as mentioned above, LIMES is used as a colleague generating references.

Figure 5 shows how the proposed scheme works. Only when the LIMES simulator arrives at T1 do we know all the references from the processors involved (step 1). Then, LIMES sends to SIDE the corresponding information (step 2) by means of the pipe. Next, SIDE simulates the number of cycles corresponding to the number of instructions from the previous reference to the actual reference in each processor (step 3). When the timeline arrives to T2, processor 0 reference is satisfied (perhaps because the reference was in local memory) and LIMES is restarted (step 4). Nine cycles later, processor 0 issues another reference and SIDE can continue simulating the new 9 cycles from processor 0 and the work left in memory system 1 (step 5).

In this way, we achieve the correct execution because the order in which the events are generated is known by LIMES, and we obtain the correct time because SIDE knows this.

More details about LIDE and its block structure can be found in [23].
4. Performance Evaluation Study

We are interested to know how two well-known consistency models, SC and LRC, perform. For this purpose we simulate their behavior using LIDE as simulation tool.

Three benchmarks from the SPLASH-2 workloads were selected: FFT, Barnes, and Radix. The default was used as the problem size except for the FFT kernel and Barnes application, where results were obtained for $2^{18}$ particles and $2^{12}$ particles, respectively.

The modeled architecture consisted of a single cluster varying from 2 to 32 processors, with a 1024-byte page size. The memory assumed in the nodes was large enough to avoid disk pagination. We assumed an ideal network to perform the initial tests to check the behavior of the selected consistency models among the parallel benchmarks.

The chosen page size is relatively small, but, under suitable benchmarks, it produces sufficient false sharing effects to compare the behavior of both consistency models. These effects grow as the problem size grows; thus, our results would be magnified with a larger problem size.

The load in each processor consists of the SPLASH benchmark plus the operating system overhead to maintain the consistency model. A memory data access that does not involve a page fault takes just one simulation cycle; on the other hand, a context change (owing to page faults, new network messages, or operating system scheduling) takes 1000 simulation cycles. The time overhead due to \texttt{diff} creation and application is taken into account in the LRC model. This time period is a function of the page size. These overheads are not present when a page is copied because the model assumes that this task is performed by DMA, in parallel with the operating system.

While pending network requests exist the processor is busy serving them. The assumed service time is 100 cycles. When no network request is pending, the processor schedules the SPLASH-2 workload, and this takes 1000 cycles.

Figure 6 shows the total communication time measured in processor cycles for the selected benchmarks. It has been obtained by subtracting the computation time from the total execution time. As expected in Barnes and Radix the LRC model performs better than the sequential. The exception appears in the FFT benchmark, where the differences are only just a bit worse. To explain the large differences in Radix kernel and the strange behavior of the FFT kernel we concentrate into the traffic bus and gathered the number of messages that each model sends to the bus. Figure 7 shows these results. In all three benchmarks LRC model sends less messages than the sequential model. The huge differences in Radix between both models explain the above behavior.
The results in FFT changes respect to the previous ones where results in the LRC model were a bit worse, that is because barriers in the LRC model are slower (send more information) than in the sequential one. In addition, false sharing is negligible [9] independently of the page size; consequently, both models behave similar. In general, false sharing is the catalyst that makes differences in both models become larger and larger.

Figure 6 – Total communication time in the selected benchmarks.

Figure 7 – Number of messages sent using the LRC and SEQ consistency models.

Figure 8 shows the average latencies for both write and read operations. Latencies in write operations are dramatically large in Radix. Even, in some cases this latency is more than two hundred times greater than the obtained in FFT and Barnes.
That is due to the huge false sharing showed in this kernel which carries negative effects on latency time, as the well-known ping-pong of shared data pages between processors issuing writes [24].

In Radix this phenomenon has an adverse impact in writes and not in reads due to the nature of this kernel, where communication all-to-all processors is driven through writes rather than reads [25]. This is also the reason that differences in write latencies in FFT are greater than in read latencies. More details of the nature of these kernels can be found in [26].

In Barnes application LRC offers better results both in writes and reads. These results are coherent with the offered in [3].

5. Conclusions

In a previous paper we introduced LIDE [23] as a simulation environment for distributed shared memory systems to compare performances of consistency models. In this work we have used LIDE to evaluate two consistency models (LRC, SEQ). LIDE is based on two well-known simulators: LIMES and SIDE. LIMES uses real workloads in simulations and collects the memory references generated by them. SIDE simulates the memory architecture and the interconnection network.

Previous studies [3,4,7,10,12] use the real system to obtain results of consistency models. The main contribution of this work is that the evaluation study of the models has been made using a software package and workloads of the SPLASH-2 benchmark suite. This fact has allowed us to study writes and reads in a separate way concluding that follows below.
As expected, the number of messages send to the network is always less in the LRC than in the sequential model; consequently the execution time is better using the LRC consistency model.

SPLASH-2 benchmark suite was designed for shared memory multiprocessor systems, which use the cache block as coherence unit. Pages (coherence unit in SVM) are much greater than cache blocks sizes. If pages are used as unit to share data among processors and consistency software protocols take care of coherence, different latencies between reads and writes can be found. These differences vary with the workload and the consistency protocol. In some cases differences in sequential model between write and read latencies are so large that writes latencies represent a significant percentage of the total program execution time, in spite of the amount of writes is much smaller than the amount of reads.

False sharing makes that differences in write latencies among the protocols grow. Even, in some cases the differences are much (hundreds of times) greater.

6. References


