Performance of Software Distributed Shared Memory Systems

Dimitra Koletsou, Tycho Nightingale, Suleyman Sair, Ozgur Sinanoglu
Department of Computer Science and Engineering
University of California, San Diego
San Diego, CA
{koletsou,tycho,ssair,ozgur}@cs.ucsd.edu

Abstract

A distributed shared memory (DSM) system implements a global shared address space on physically distributed memory over several uni-processor or multiprocessor machines. DSM systems yield the high performance, scalability, and cost-effectiveness of message passing based systems, while providing the simple parallel programming model of shared memory systems. While a DSM system hides the communication burden from the application programmer, it must reduce the communication latency, both software overhead and transfer time, in order to achieve true performance transparency with respect to common memory multiprocessors. The consistency model and page size have an impact on the performance of the system by determining the amount of data and number of coherency operations, i.e. synchronization messages, used during system operation. Achieving optimal DSM performance heavily relies on interaction between program structure and consistency model. Additional techniques such as dynamic page sizing and a speculative DSM can further enhance performance.

1 Introduction

In order to satisfy the increasing demand for high computing power of recent applications, different configurations of multiple processors are frequently used. For example, multiprocessor machines running parallel applications offer increased performance. Additionally multiprocessor machines with a common physical memory can deliver this high computing power while offering a simple parallel programming model using read and write sharing. A message passing parallel application running on several distributed uni-processors can also offer increased computing power, but application programming is difficult, especially when passing complicated data structures.

A distributed shared memory (DSM) system implements a global shared address space on physically distributed memory over several uni-processor or multiprocessor machines. DSM systems yield the high performance, scalability, and cost-effectiveness of message passing based systems, while providing the simple parallel programming model of shared memory systems.

Since DSM systems offer a transparent interface, convenient programming model and compatibility with existing parallel software, they have gained wide acceptance in both technical and commercial computing environments. This paper surveys the issues related to performance in software DSMs.

1.1 DSM Design

According to the seminal paper by Li and Hudak [1] there are two major choices when designing a DSM system. First is the choice of granularity of the shared memory objects, i.e. the page size, and second, the consistency model.

The choice of page size presents a trade-off between the overhead of transferring data and increased possibility of contention. Smaller page sizes increase the network overhead while minimizing false sharing. False sharing is the effect that occurs when multiple processors are using non-overlapping portions of the same memory page. Li and Hudak suggest that a good compromise for page size granularity is selecting the shared memory page size equal to the virtual memory page size. This equality has the advantage of using the existing page-fault schemes, including the use of the hardware memory management unit (MMU) and modularity for page-fault handlers.

The consistency model is responsible for maintaining the coherence, ensuring that the value returned by a read operation is always the same as the value written by the most recent write operation, and the distribution of the shared data. The choice of a consistency model not only specifies how the memory system will appear to the programmer, but also influences how parallel programs are written and impacts the overall design of the DSM system. For example, the consistency model specifies which processors may have a copy of a shared page and what will happen on a write to it.

The consistency model and page size have an impact on the performance of the system by determining the amount of data and number of coherency operations, i.e. synchronization messages, used during system operation.

1.2 DSM Performance

While a DSM system hides the communication burden from the application programmer, it must reduce the communication latency, both software overhead and transfer time, in order to achieve true performance transparency with respect to common memory multiprocessors.

Software overhead represents the operating system cost of calling a message-handler for incoming messages, creating and reading messages and the cost of the DSM protocol implementation. Faster processors and hardware support for messaging reduce the effect of software overhead. Moreover, protocols that reduce the number of messages also exhibit lower software overhead and increased performance.

The network transfer time causes remote shared-memory accesses to take up to 10–100 times longer than local memory accesses. Reducing the remote memory latency is critical to achieving high performance.

The rest of this paper is organized as follows: Section 2 discusses consistency protocols, Section 3 is concerned with page size issues, Section 4 introduces a speculative DSM. A discussion on future directions is the subject of Section 5. Finally, we conclude and summarize in Section 6.

2 Consistency Protocols

The first half of this section focuses on implementation choices. Determining the right implementation of consistency protocols for software DSMs is critical to efficiency. Protocols vary in the implementation of *write trapping* and *write collection* methods. Compiler instrumentation and twinning are two write trapping mechanisms that could be employed. On the other hand, scanning timestamps and building diffs are two widely used methods for write collection. Moreover, a protocol can allow single or multiple writers to a shared memory object.

2.1 Write Trapping and Write Collection

Write trapping is used for detecting which shared data are changed during a particular execution interval, as only the modified data need to be transmitted at synchronization points [2]. Compiler instrumentation requires the compiler to emit extra code to set a dirty bit on writes to shared memory. A dirty bit that is set indicates that the corresponding shared data have been modified. As the overhead to set the dirty bit is incurred on every write to shared data, larger sized regions will result in fewer dirty bits and minimal overhead. Twinning, on the other hand, makes a copy of the shared data for the system, called the *twin*, and later compares the user copy of the shared data to the twin to discover which elements were modified. The granularity of the shared object determines the preferred method for write trapping.

Write collection involves determining which data need to be sent when a synchronization request arrives [2]. In timestamping, a timestamp is associated with each unit of shared data. After discovering that a block has changed using write trapping, every block that has a larger timestamp than the timestamp contained in the request message, is piggybacked on the lock grant message. When the diffing method is used however, records called *diffs* are employed. A diff is a run-length encoding of changes to the shared object, in other words a log of changes to it. *Lazy diffing* can be used to reduce the overhead of creating diffs. In this scheme, diffs are only made at the time of lock request instead of at every lock release. Timestamping incurs more computation overhead as all timestamps must be scanned each time a request comes in. Diffs on the other hand, are created once and can be reused until a subsequent write invalidates the current diff.

2.2 Single vs. Multiple Writers

Sophisticated DSM systems achieve high performance through a combination of relaxed consistency models and multiple-writer protocols [3]. Single-writer protocols allow multiple readers to access a given datum simultaneously, but require a writer to gain ownership and exclusive access to a page before modifying it. Such protocols are relatively straightforward to implement because all copies of a given datum are identical. Page faults can be satisfied by retrieving a new copy of the data from any other processor that has a copy. Unfortunately, simplicity comes at the expense of message traffic. The effects of false sharing are alleviated by supporting multiple-writer protocols. Multiple-writer protocols allow multiple nodes to simultaneously modify different sections of the same page. Modifications are later reconciled to create diffs minimizing the effects of false sharing. Diffs allow each processor to make only the necessary updates to shared pages. The disadvantages are that these protocols are more complex, a diffing mechanism must be used to merge multiple modifications to the same page, and the memory overhead is high.

The rest of this section describes different protocols that make use of the above mentioned implementations and provides a comparison of performance.

2.3 Sequential Consistency

Single writer sequential consistency (SW-SC) is an implementation of a page-based, single-writer protocol [3]. Either a single writer or multiple readers may have a copy of a given page at any one time. Page ownership is migrated to each processor as a copy of the page is requested, regardless of whether the request was for a write or a read copy. Using sequential consistency, the result of any execution is the same as if the operations of all 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. The ping-pong problem is addressed by guaranteeing a processor access for a minimum quantum of time to any newly retrieved page before it can be invalidated by another processor. The ping-pong problem occurs when multiple processors simultaneously attempt to write to the same page. A processor might request and gain ownership of a page, but receive an ownership request for the page from another processor before the fault handler exits. In this case, the modification is not completed before ownership is lost, and ownership must be re-requested. Since modifications are often minor, even a small quantum completely hides the problem in most cases.

2.4 Release Consistency

Release consistency requires that synchronization operations for any access to shared variables employ lock acquire and release messages, which force the coherence of shared data observed by the processor sending the message. It does not guarantee that shared memory is consistent all the time however, it rather makes sure that consistency is achieved only after synchronization operations. This naturally results in lower synchronization traffic than a more restrictive memory consistency model. There are variations of this model which depend on when the modified data are propagated.

Eager Release Consistency In an *Eager Release Consistency* (ERC) model [4], a processor delays propagating its modifications to shared data until it comes to a release. At that time, it propagates the modifications as diffs to all other processors that have, i.e. cache, the modified pages. For an invalidate protocol, the only thing done is sending invalidations for all modified pages to other processors that cache these pages (Figure 1). To limit the amount of data exchanged, an update protocol sends a diff of each modified page to other cachers, and then these diffs are merged in the other cached copies (Figure 2). In either case, the release blocks until the acknowledgments have been received from all other cachers. On an acquire, the only operation performed is that the lock release is sent from the last releaser to the current acquirer. On an access miss, the page is obtained from the current owner.

Lazy Release Consistency It suffices to update the copy of each processor only when it acquires the lock. Therefore, *Lazy Release Consistency* (LRC) model [4] does not make modifications globally visible at the time of a release. Instead, it only guarantees that a processor acquiring the

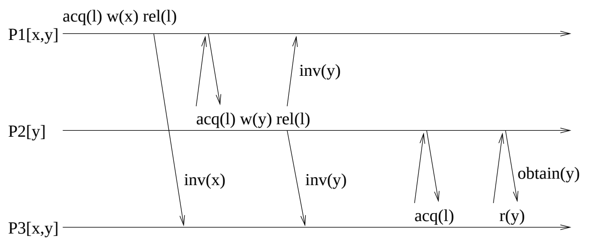

Figure 1: Eager Invalidate. x and y are shared objects residing in the same page guarded by lock l.

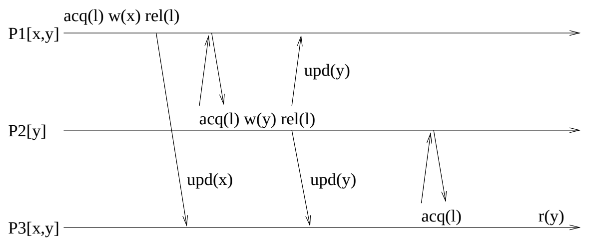

Figure 2: Eager Update. x and y are shared objects residing in the same page guarded by lock l.

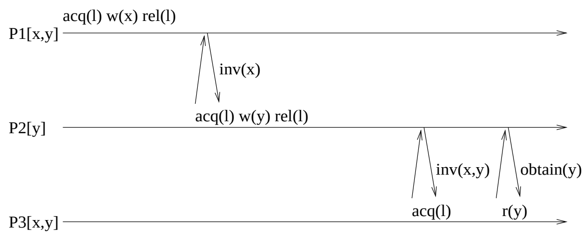

Figure 3: Lazy Invalidate. x and y are shared objects residing in the same page guarded by lock l.

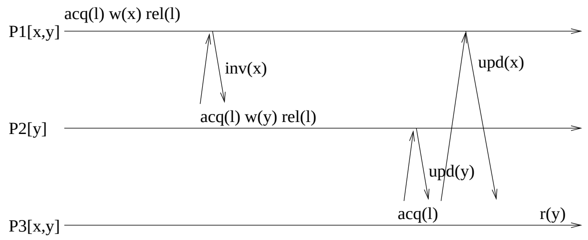

Figure 4: Lazy Update. x and y are shared objects residing in the same page guarded by lock l.

lock will see all modifications that precede the acquire. A modification precedes a lock acquire if it occurs before any release such that there is a chain of release-acquire operations on the same lock, ending with the current acquire. Only the modifications that precede the acquire are sent to the acquiring processor. The modifications can be piggybacked on the message that grants the lock for improved performance. On an acquire, the acquiring processor sends its current timestamp to the previous releaser. The releaser uses the above mentioned timestamping method to send back the write notices for all modified pages, where a write notice is an indication that a page has been modified. Depending on whether a lazy invalidate (LI), or a lazy update (LU), protocol is used, the acquiring processor either invalidates all pages in its cache for which it received write notices or updates those pages according to the diffs (see Figure 3 and Figure 4 respectively). If the processor never had a copy of the page, an access miss, a copy of the page may have to be retrieved as well as a number of diffs. These diffs are then reflected in the page before it is accessed.

Lazy Hybrid Protocol The *lazy hybrid protocol* [5] (LH) is a variant of LRC (Figure 5). In the LH protocol, as in LU, the releaser piggybacks write notices along with the modifications to those pages that it believes the acquirer has a copy of in its' memory. The acquirer does not attempt to request any other modifications. While updating the pages that it received diffs for, it invalidates those for which it received a write notice but no modifications were included in the lock grant message. In general, the acquirer must communicate with other processors in order to pick up all the required diffs. However, as all prior modifications to a page can be obtained from the processor that last modified it, it suffices to communicate with the lock releaser. The LH protocol combines the advantages of LU and LI protocols. First, like the LI protocol, the hybrid only exchanges a single pair of messages between the acquiring and the releasing processors. As a result, lock acquisition latency for the LH protocol is close to that of the LI protocol. The only additional overhead comes from the need to send and process the modifications piggybacked on the lock grant message. Second, the amount of data exchanged is smaller than that of the LU protocol. Finally, the hybrid sends updates for recently modified pages cached by the acquirer. It is likely that these pages will be accessed, therefore reducing the number of access misses, and as a result reducing the latency and the number of miss messages.

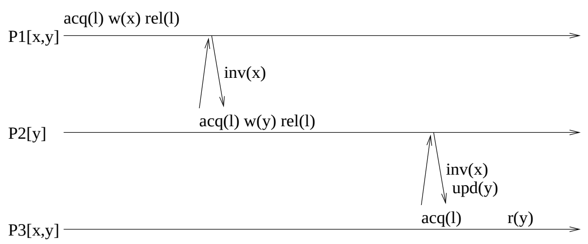

Figure 5: Lazy Hybrid. x and y are shared objects residing in the same page guarded by lock l.

Aggressive Release Consistency Protocol Instead of at the time of the acquire like the LRC model, *aggressive release consistency* (ARC) model [6] postpones the enforcement of data coherence to the time of the first shared memory access after an acquire (Figure 6b). Since memory accesses, if any, between the acquire instruction and the first shared data access instruction involve only local variables, the execution of these local memory accesses never compromises the integrity of shared memory consistency. Therefore ARC guarantees the same data consistency as LRC. Every processor in LRC, before fetching shared data, must first issue a lock request message to acquire the lock. Then it sends a data request message to get the updated data where distinct lock and data request messages are sent respectively. Any processor receiving a lock request (data request) releases the lock (sends the data), in each case two messages are needed. To satisfy the access of one shared data after the lock acquire instruction, four messages are required in total. In contrast, ARC model enforces data coherence at the time when a processor accesses the shared data for the first time after the lock acquire instruction, making it possible to merge a lock request message and a data request message into one message; correspondingly a lock release message and a data reply message can be merged into one message, so the total number of messages is two. ARC enforces data consistency at the page level and employs a multiple writer protocol with the propagation of modifications to overcome coarse granularity and false sharing. Like LRC, write notices are piggybacked in lock release messages and the data reply messages are transferred only on the demand. According to ARC, any access miss on a shared page brings the required modifications from all the processors which updated the shared data to be fetched.

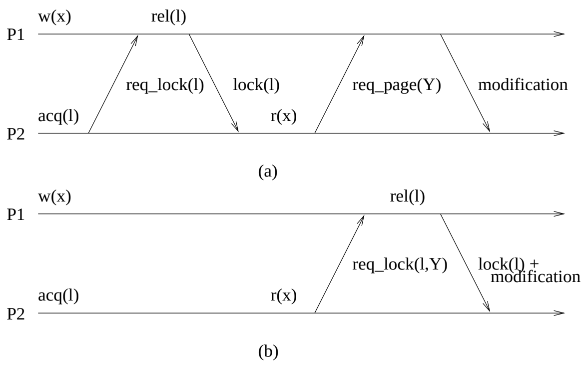

Figure 6: LRC vs. ARC. x is the shared object that resides in page Y.

Entry Consistency In *Entry consistency* (EC) [2], all shared data must be explicitly declared as such in program text, and be associated with a lock or barrier. When an acquire is done on a synchronization variable, only shared data guarded by that synchronization variable are made consistent. Therefore, EC requires detailed data access behavior to be known by the programmer and thus, programming is harder and more error prone. For example, in the EC model, if it is desired that elements of an array be accessed independently in parallel, then different array elements must be associated with different locks [7]. On the other hand, it allows multiple critical sections involving disjoint shared variables to execute simultaneously, increasing the amount of parallelism.

2.5 Comparisons

Different protocols have their merits under certain conditions. In this section, we will analyze and compare these merits of the above mentioned protocols.

Single-writer protocols require ownership. They are simple to implement but have high message traffic. On the other hand, multiple-writer protocols minimize the effects of false sharing, but use many small messages in doing so, and therefore have lower performance. To reduce associated software overhead, higher speed CPUs or dedicated hardware support for diffing are required. Finally, while more flexibility can be obtained by multiple-writer protocols, they are more complex to implement.

When it comes to comparing ERC to LRC, instead of sending updated shared data to all other processors at a release as in ERC, LRC enables indications of updated shared data (instead of the data itself) to be piggybacked on the lock release message sent to the processor. This subsequently avoids sending messages unnecessarily to those processors which do not acquire the shared data. In other words, the updated shared data are only sent on the demand of the processor which acquires the shared data. LU sends fewer messages than EU because updates are only sent to the next processor to acquire the lock. For a lock operation, in LU the new lock holder collects all the diffs necessary to bring its cached pages up-to-date, communicating with the other concurrent last modifiers for any local page. All the other messages – which serve the purpose of finding the owner of and transferring the lock – are the same for lazy and eager protocols. On unlocks, the eager protocols send write-notices to all cachers of locally modified pages. The lazy protocols do not communicate on unlocks. In the light of these comparisons, it can be stated that when the data is heavily shared the locks requests dominate the synchronization. Because DSMs consume network bandwidth, applications with greater communications needs tend to favor lazy protocols.

The latency of synchronization is the dominant factor in the efficiency of software DSMs. Experiments reveal that with fast processors, the performance of DSM systems that can be achieved on an Ethernet network is limited. Serialization of messages, collisions, and low bandwidth severely constrain speedups, even for coarse-grained programs. Higher bandwidth point-to-point networks, such as ATM LANs, allow much better performance, with good speedups even for medium-grained applications. Fine-grained applications still perform poorly even on such networks, due to the frequency and cost of synchronization operations. Thus, employing lazy protocols that reduce the amount synchronization is essential. Moreover, using hybrid implementations of LRC would further reduce communication overhead, yielding higher performance when compared to other LRC schemes. If using lazy protocols does not attain the desired performance, programs can be restructured to reduce the amount of synchronization.

Delaying the propagation of the modifications until the first shared memory access makes it possible to merge two messages into one (See Figure 6a and Figure 6b). While the total number of messages is reduced, merging these messages yields more data per message. However, one message header is saved and the part of the message that includes the information about the diffs is shortened as the page number that the diffs refer to is already available in the lock part of the message. It is advantageous to deliver larger amounts of data through fewer messages since data size is less relevant than the software overhead associated with sending messages. If the application program uses more locks ARC outperforms LRC, otherwise the performance of the two models is approximately the same.

When comparing the performance of different implementations of EC and LRC, the differences are mainly due to the granularity (i.e. page or block size). In general, the coherence unit for LRC is the virtual memory page size. Thus, when the coherence unit is larger than a page, EC performs better as it sends fewer messages. The downside is, with larger sizes, false sharing starts deteriorating the performance. Although it is harder to implement, the extra programming effort put into EC attempts to bring in exactly the right data to a processor.

3 Page Size

Increasing the page size of a DSM can enhance performance by reducing the number of page faults, coherence operations and associated messages, but an increased page size yields a) greater false sharing and b) unnecessary coherence messages and second results in unnecessary data being transfered across the network.

To take advantage of the increased page size, Amza *et al.* propose an algorithm in [8] that dynamically aggregates pages into larger *page-groups*, which need not be contiguous. The faulting behavior of the pages in a group is monitored and the algorithm decides whether to aggregate more pages into a group or whether to split-up page-groups into pages. This aggregation increases performance by fetching multiple pages that are accessed by the processor together while avoiding an increase in false-sharing which could otherwise be caused by a static increase in page size.

4 Speculative Coherency

The overhead associated with the coherence operations inhibits DSMs from attaining performance transparency with respect to common memory multiprocessor machines because a remote memory access takes significantly longer than a local memory access. One method to improve performance is to hide the remote memory access latency. By predicting and speculatively performing the necessary coherence operations in advance, the remote memory access overhead can be potentially hidden.

A speculatively coherent DSM using a *Memory Sharing Predictor* (MSP) [9] predicts remote memory access and allows for the timely execution of the associated coherence operations. Predictions must be made early enough to completely hide remote access latency. For predictions, Lai and Falsafi use a pattern-based predictor tracking only a subset of the message stream, more specifically only request messages. To ensure the timeliness of predictions, they rely on the use of common memory access patterns. For example, to determine the time of a read request in producer/consumer sharing, the DSM should predict when the producer is done writing. Then the DSM can speculatively invalidate the producer's page and forward it to the consumers. If the DSM can not issue a prediction, then upon the first consumer's read, it speculatively sends the page to the remaining consumers.

Achieving high performance in a speculatively coherent DSM requires accurate predictions. Poor predictions cause unnecessary messages, invalidations and data transmissions.

5 Future Directions

We can utilize an integrated compiler and run-time environment to increase performance as well as provide ease of programming.

Compilers provide access pattern information to the run-time system at the points in the code that will be executed in parallel. The run-time uses these points to determine available computational and communication resources. Based on the access patterns, as well as computing power, the run-time can then make intelligent decisions not only to distribute the computational load evenly, but also to minimize communication overhead in the future. The result is a system that adapts both to changes in access patterns as well as to changes in computational power, yielding to reduced execution time. The access pattern information can also be used to prefetch data. Additionally, this system would be able to partition the work so that processes access only their local data, minimizing false sharing.

Note that load balancing can have adversary effects on locality. Distributing data across the network would increase the number of necessary messages that are exchanged to gather data. As communication overhead is a dominant factor in DSMs, this would degrade overall performance. Consequently, there is a need to avoid unnecessary movement of data and at the same time minimize false sharing. The run-time system can employ the access pattern information gathered by the compiler to provide this capability as well.

As pointed out in previous sections, EC minimizes the amount of data that is transferred at lock acquire by associating shared data objects with synchronization variables. However, this comes at the expense of increased programming complexity. During type-checking, the compiler locates the use points of shared variables. Therefore, the association process can be automated by enclosing those use points with a pair of lock acquire and release operations. In doing so, it would provide the advantages of EC while minimizing programming complexity.

6 Conclusions

Achieving optimal DSM performance relies heavily on interaction between program structure and consistency model. This interaction determines the rate of message traffic used in communication. Performance reducing software and transmission overheads are directly proportional to the number of messages. Even if a balance is found between program structure and consistency model, additional techniques such as dynamic page sizing and MSP can further enhance performance.

When the ideal case of program structure and protocol balance has resulted in minimal number of possible messages, additional performance can be obtained through the use of more powerful CPUs and more efficient network protocols such as ATM-LANs.

References

[1] K. Li and P. Hudak, "Memory Coherence In Shared Virtual Memory Systems," *ACM Transactions on Computer Systems*, 1989.

[2] S. Adve and A. Cox and S. Dwarkadas and R. Rajamony and W. Zwaenepoel "A Comparison of Entry Consistency and Lazy Release Consistency Implementations," *Proceedings of the Second IEEE Symposium on High-Performance Computer Architecture*, 1996.

[3] P. Keleher, "The Relative Importance of Concurrent Writers and Weak Consistency Models," *Proceedings of the 16th International Conference on Distributed Computing Systems*, 1996.

[4] P. Keleher and A. Cox, and W. Zwaenepoel, "Lazy Release Consistency for Software Distributed Shared Memory," *Proceedings of the 19th Annual International Symposium on Computer Architecture*, 1992.

[5] S. Dwarkadas and P. Keleher and A. Cox and W. Zwaenepoel, "Evaluation of Release Consistent Software Distributed Shared Memory on Emerging Network Technology,"*Proceedings of the 20th Annual International Symposium on Computer Architecture*, 1993.

[6] S. Fu and N-F. Tzeng "Aggressive Release Consistency for Software Distributed Shared Memory," *Proceedings of the 17th International Conference on Distributed Computing Systems*, 1997.

[7] A. Tanenbaum, "Distributed Operating Systems," *Prentice Hall*, 1995.

[8] C. Amza and A. Cox and K. Rajamani and W. Zwaenepoel, "Tradeoffs Between False Sharing and Aggregation in Software Distributed Shared Memory," *Proceedings of the Sixth ACM SIGPLAN symposium on Principles & practice of parallel programming*, 1997.

[9] A. Lai and B. Falsafi, "Memory Sharing Predictor: The Key to a Speculative Coherent DSM", *Proceedings of the 26th Annual International Symposium on Computer Architecture*, 1999.