CSE 6306 Advance Operating System
Spring 2002

Consistency Issues in Distributed Shared Memory Systems

Chingwen Chai
University of Texas at Arlington
cxc9696@omega.uta.edu

Abstract

In the field of parallel computing, the popularity of Distributed Shared Memory (DSM) systems is believed to be increasing. The idea of distributed shared memory is to provide an environment where computers support a shared address space that is made by physically dispersed memories. Distributed shared memory received much attention because it offers the power of parallel computing using multiple processors as well as a single system memory view which makes the programming task easy.

Consistency in a distributed shared memory system is an important issue because there might be some potential consistency problems when different processors access, cache and update the shared single memory space. In order to improve performance and get correct result of computation, distributed shared memory systems designers should choose the proper paradigm of memory coherence semantics and consistency protocols.

The purpose of this paper is to provide an introductory overview of distributed shared memory systems and point out the consistency problems and the possible solutions. We will also study the cases of several state-of-the-art implementations and their contribution in maintaining system memory consistency.

1. Introduction

1.1. Overview

In 1986, Kai Li published his PhD dissertation entitled, "Shared Virtual Memory on Loosely Coupled Microprocessors," thus opening up the field of research that is now known as Distributed Shared Memory (DSM) systems. [1] Since then, lots of researches in distributed shared memory systems have been proposed. In distributed shared memory systems, processes share data across node boundaries transparently. All nodes in the distributed shared memory system perceive the same illusion of a single address space (Figure 1). Any processor can access any memory location in the address space directly. Memory mapping managers is responsible for mapping between local memories and the shared memory address space. Other than mapping, their chief responsibility is to keep the address space coherent at tall times; that is, the value returned by a read operation is always the same as the value written by the most recent write operation to the same address. [2] There advantages of distributed shared memory systems including:

- Processes can run on different processors in parallel
- Memory mapping, page faulting, data movement are managed by distributed shared memory without user intervention
- Single address space simplifies programming tasks

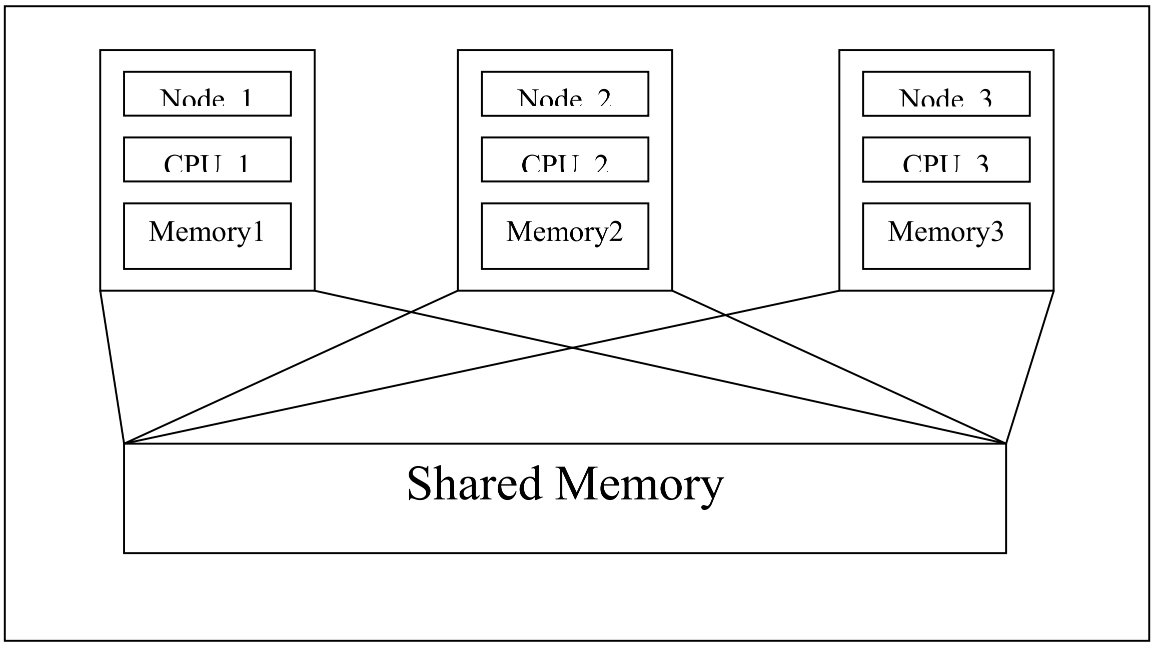

Figure 1 A single image illusion of distributed shared memory systems

1.2. Design Issues

Several design issues need to be addressed before we go further into this survey. Each of these factors significantly affects the performance of the system.

- **Virtual memory and distributed shared memory**:

  Modern computer systems employ the concept of virtual memory to achieve better performance. The virtual memory management mechanism is responsible for page replacement, swapping and flushing. Similarly, in satisfying a remote memory request, the distributed shared memory would have to consult the virtual memory manager to get a page frame, etc [3]. The effectiveness of the distributed shared memory paradigm depends crucially on how quickly a remote memory access request is serviced and the computation is allowed to continue.

- **Granularity**:

  Computation granularity refers to the size of the sharing unit. It can be a byte, a word, a page or other type of unit. Choosing the right granularity is a major issue in distributed shared memory because it deals with the amount of computation done between synchronization or communication points. Moving around code and data in the networks involves latency and overhead from network protocols. Therefore, such remote memory accesses need to be integrated somehow with the memory management at each node. This often forces the granularity of access to be an integral multiple of the fundamental unit of memory management (usually a page) or simply transfer part of the page to reduce the latency [3].

- **Memory Model and Coherence Protocols**:

  To ensure correct multiprocessor execution, memory models should be employed with care. Two conventional memory models are utilized in many distributed shared memory systems. Sequential Consistency memory model ensures that the view of the memory is consistent at all times from all the processors. The other is Release Consistency, which distinguishes between kinds of synchronization accesses, namely, acquire and release, establishing a consistent view of shared memory at the release point [3]. Several coherence protocols are used to maintain memory consistency and will be identified in detail in later sections.

1.3. Consistency Problems in Distributed Shared Memory

To get acceptable performance from a Distributed Shared Memory System, data have to be placed near the processors who are using it. This is done by replicating and replacing data for read and write operations at a number of processors. Since several copies of data are stored in the local cache, read and write access can be performed efficiently. The caching technique increases the efficiency of Distributed Shared Memory Systems, but it also raises the consistency problems, which happens when a processor writes (modifies) the replicated shared data. How and when this change is visible by other processors who also have a copy of the shared data becomes an important issue.

A memory is consistent if the value returned by a read operation is always the same as the value written by the most recent write operation to the same address [2]. In a distributed shared memory system, a processor has to access the shared virtual memory when page faults happen. To reduce the communication cost initiated by this reason, it seems naturally to increase the page size. However, large page size produces the contention problem when a number of processes try to access the same page and it also triggers the false sharing problem, which, in turn, may increase the number of messages because of aggregation. [4].

False sharing is caused by the large size of the memory page and considered to be a performance bottleneck to distributed shared memory systems. False sharing occurs when two unrelated variables (each used by different processes) are placed in the same page. The page appears shared, even though the original variables were not [5]. Conventional programming usually requires processes to gain exclusive access to a page before it starts modification. Therefore, false sharing leads to a race condition where multiple processors compete for ownership of a page while actually they are modifying totally different sets of data.

Several techniques are introduced to reduce the effect of false sharing including: Relaxed memory consistency model and write-shared protocols. We will investigate these solutions and the implementations in later sections.

A cache coherence problem can be illustrated as follows. It occurs when processors get different view of memory when accessing and updating at different time. For example, if two processors, X and Y, cache two different variables, A and B, locally, the value in the cache may not be coherent when one of them modified the value of the variable and the other processor is not notified (Figure 2). This memory inconsistency may leads to serious computation problems.

Time	Processor X	Processor Y
↓	A = 0	B = 0
	A = 1	B = 1
	A = A + B	
		B = A + B

(Figure 2)

1) Variables A and B are initialized to 0.
2) A and B are updated to the value 1
3) Because of the lack of memory coherency mechanism, X still thinks that B is 0 and Y still thinks that A is 0
4) Finally, X and Y will both think A and B to be 1, which is the wrong answer. The correct answer for A and B should be 2

Many solutions are proposed to reduce or even eliminate these consistency problems. We will investigate some of them in later sections

2. Memory Coherence Models

2.1. Sequential Consistency

Lamport defined the system to be sequentially (strictly) consistent if [7]:

> *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.*

The system ensures that all accesses of the shared memory from different processors interleave in a certain manner so that the consequential execution is the same as if these accesses are executed in some sequential order. While this model guarantees that every write is immediately seen by all processors in the system, it also generates more messages for maintaining this kind of consistency and, thus, higher latency. Moreover, determining sequential consistency is an NP-complete problem [7], which may leads to serious system slowdown in large-scale distributed shared memory systems.

2.2. Processor Consistency

Processor consistency allows writes from different processors to be seen in different orders, although writes from a single processor must be executed in the order that they occurred. Explicit synchronization operations must be used for accesses that should be globally ordered. The main advantage of processor consistency is that it allows a processor's reads to bypass its writes and hence increase the system performance.

2.3. Relaxed Consistency

Relaxed (weak) consistency does not require changes to be visible to other processors immediately. When certain synchronization accesses occur, all the previous writes have to be seen in the program order. Two processes are said to be competing if at least one of them is a write. Shared memory accesses are categorized either as ordinary or synchronization accesses, with the latter category further divide into acquire and release accesses [8].

Two well-know approaches implementing the relaxed consistency are:

- **Release Consistency (RC):**

  Release consistency is a form of relaxed memory consistency. A system is release consistent if:

  - Before an ordinary access is allowed to perform with respect to any other processor, all previous acquires must be performed
  - Before a release is allowed to perform with respect to any other processor, all

previous ordinary reads and writes must be performed

- Special accesses are sequentially consistent with respect to one another.

The advantage of this form of consistency is that it delays the consistency update with synchronization events. Therefore, updates occur only when needed by application and unnecessary messages will be reduced. However, most release consistent systems require the programmer to make explicit use of acquire and release operations.

- **Lazy-Release Consistency (LRC):**

In Lazy-Release Consistency, the propagation of modifications is further postponed until the time of the acquire [10]. A system in LRC has to satisfy the following conditions [11]:

- Before an ordinary read or write access is allowed to perform with respect to another process, all previous acquire accesses must be performed with respect to that other process

- Before a release access is allowed to perform with respect to any other process, all previous ordinary read and store accesses must be performed with respect to that other process, and

- Sync are sequentially consistent with respect to one another

2.4. Entry Consistency

In entry consistency, data must be explicitly declared as such in the program text, and associated with a synchronization object that protects access to that shared data. Entry consistency takes advantage of the relationship between specific synchronization variables which protect critical sections and the shared data accessed within those critical sections. Processes must synchronize via system-supplied primitives. Synchronization operations are divided into acquires and releases. After completing an acquire, entry consistency ensures that a process sees the most recent version of the data associated with the acquired synchronization variable. [16]

The above consistency models can be summarized and illustrated in the following table.

Strictness	Consistency Models

More Strict	Sequential consistency
	Processor consistency
	Weak consistency
	Release consistency
Less Strict	Entry consistency

(Table 1) Summery of consistency

3. Consistency Protocols

Caching shared data introduces increases the system performance in distributed shared memory systems. However, to maintain memory consistency, special designed protocols needed to be implemented to 1) propagate a newly written value to all cached copies of the modified location, 2) detect when a write is completed and 3) preserve the atomicity for writes with respect to other operations [12].

3.1. Write-Shared Protocol

The write-shared protocol buffers the write accesses thus allows multiple writers update concurrently. Two or more writers can modify their local copies of the same shared data at the same time and the modified copies are merged in the next synchronization event.

The distributed shared memory software initially write-protect the memory page containing the write-shared data. When some processor wants to modify this page, distributed shared memory software makes a copy of the page containing the write-shared data and take off the write protection so that further update operations can be done without distributed shared memory software intervention.

The original data page is put in a delayed-update queue. At release time, the system performs a comparison of the original page and its copy and run-length encodes the results of this difference into the space allocated to the copy. Each encoded update consists of a count of identical words, the number of differing words that follow, and the data associated with those differing words. Then each node that has a copy of a shared object that has been modified is sent a list of the available updates. The receiving nodes will then decode the updates and merge the changes into their version of the shared data. This protocol eliminates the ill effects of false-sharing and hence lowers the communication associated with it [14].

3.2. Lazy Diff Creation Protocol

Basically, LDC is identical to write-shared protocol is the sense of create *diffs* for merging further update. The time of creating *diff* in LDC is postponed until the modifications are requested, which differs from that of write-shared protocol. This significantly reduced the number of *diffs* created and improved performance.

3.3. Eager Invalidate Protocol:

Eager protocols push modifications to all nodes that cache the data at synchronization variable releases. If remote copy is read-only, it is simply invalidated; if the copy is marked as read-write, the remote node appends the diff to the reply and then invalidates the page. When the locking processor releases its writes, all other caching nodes are notified that they must invalidate their copies. The acquisition latency is long when lock request pending at release, short otherwise.

3.4. Lazy Invalidate Protocol

In lazy invalidate, the propagation of modifications is delayed until the time of the acquire. The releaser notifies the acquirer, of which pages have been modified, causing the acquirer to invalidate its local copies of these pages. A processor incurs a page fault on the first access to an invalidated page, and gets *diffs* for that page from previous releasers. The execution of each process is divided into partially ordered intervals, which is usually represented by timestamps. Every time a process performs a release or an acquire, a new interval begins. Local copies of pages for which a write notice with a larger timestamp is received are invalidated. This protocol has shortest lock acquisition latency (single message) when request pending, also good when not pending.

3.5. Lazy Hybrid Protocol:

This protocol is similar to lazy invalidate protocol except that lazy hybrid updates some of the pages at the time of an acquire instead of invalidating the modified page. The releaser sends to the acquirer all the modifications that it thinks that the acquirer is interested in. The acquirer invalidates pages for which write notices were received but no modifications were included in the lock grant message. Single pair of messages between acquirer and releaser, only have overhead of piggybacks. Amount of data is smaller than for the update protocol. Reduced number of access misses.

The trade-off between these protocols can be illustrated in the following table.

	Lock Latency	Remote Access Misses	Messages	Data	Diffs	Protocol Complixity
Eager Invalidation	Low	High	High	High	Low	Low
Lazy Invalidation	Low	Medium	Medium	Low	Medium	Medium
Lazy Hybrid	Medium	Low	Low	Medium	High	Medium

(Table 2) Protocol Trade-offs [8]

4. Examples

4.1. TreadMarks

TreadMarks is a distributed shared memory system for standard UNIX systems in the network environment consist of ATM and Ethernet. To get communication speedup, it utilized the standard low-level protocol, AAL3/4, on ATM networks, bypassing the TCP/IP protocol stack. Its developers at Rice University extended the concept of release consistency to propose lazy release consistency and employed an invalidate protocol. TreadMarks implements shared memory entirely in software and use threads to express parallelism. Both data movement and memory coherence are performed by software using the message passing and virtual memory management hardware. TreadMarks uses the virtual memory protection mechanism in modern microprocessors to communicate information about memory operations.

4.2. Midway

Midway is a software-based distributed shared memory system which optimizes the propagation of updates by using Entry Consistency. In Midway, there is an explicit binding of locks to the data that is logically guarded by each lock. As the application acquires a lock for its own synchronization, Midway piggybacks the memory updates on the lock acquisition message, thus no extra messages sent. Furthermore, the updates are sent only to the acquiring processor and only for the data explicitly guarded by the acquired lock. This serves to batch together updates and minimize the total amount of data transmitted. Local memories on each processor cache recently used data and synchronization variables. Midway's entry consistency takes into account both synchronization behavior and the relationship between synchronization objects and data. This allows the runtime system to hide the network overhead of memory references by folding all memory updates into synchronization operations [16].

5. Discussion and Conclusion

From the discussion above, we can find that distributed shared memory system provides an environment for easy programming and parallel computing. But the communication cost inherited in the underlying network is very expensive, thus limits the scalability of distributed shared memory systems and create other problems. Due to this characteristic, the granularity of memory unit is restricted in a certain range to prevent false-sharing or excessive message-passing. However, the inflexibility of granularity has a negative effect on computation speedup for some program with high communication requirement [10].

For the comparison of the system design, memory models and protocols used in TreadMarks and Midway, the result can be concluded as follows:

- For the programming ease, TreadMarks needs no special requirement while Midway requires the programmers to explicitly associate a lock with a shared data object.
- For write detection, TreadMarks system has to scan the entire shared data region, although only a small portion of it may have been updated. In Midway, system only scans the dirty bits of the shared data object.
- Midway only make those data associated with the lock consistent at a lock acquire stage. In contrast, TreadMarks needs to ensure consistency for all data objects, which results in less data being transferred in Midway than in TreadMarks.
- The avoidance of TCP/IP protocol stack hurts the portability of TreadMarks, especially in the Internet era where TCP/IP is a dominant protocol.

Obviously, there is no dominant system between these two discussed in the paper. For example, Entry Consistency outperforms Lazy Release Consistency if its coherence unit is larger than a page. If Entry Consistency's coherence unit is smaller than a page, then Entry Consistency outperforms Lazy Release Consistency if there is a false-sharing while Lazy Release Consistency outperforms Entry Consistency if there is spatial locality resulting in a prefetch effect [15]. Therefore, the choice of implementation has to be made according to the need of users or programmers as well as other conditions.

In addition to the algorithms, protocols and memory models, new network technologies may play an important role in improving Distributed Shared Memory Systems efficiency since the communication cost is still the major factor that affects system performance.

Reference:

[1] John B. Carter, Dilip Khandekar, and Linus Kamb, Distributed Shared Memory: Where We Are and Where We Should Be Headed, the Fifth Workshop on Hot Topics in Operating Systems, May 1995.

[2] Kai Li and Paul Hudak, Memory Coherence in Shared Virtual Memory Systems, ACM Transactions on Computer Systems, Vol. 7, No. 4, November 1989

[3] Ajay Mohindra and Umakishore Ramachandran, A Comparative Study of Distributed Shared Memory Design Issues, GIT-CC-94/95, August 1994.

[4] Cristian Amza, Alan Cox, Karthick Rajamani, and Willy Zwaenepoel, Tradeoffs between False Sharing and Aggregation in Software Distributed Shared Memory, Proceedings of ACM SIGPLAN Conference on Principles and Practices of Computer Programming, 1997.

[5] B Nitzberg and V Lo, Distributed Shared Memory: A Survey of Issues and Algorithms, IEEE Computer August 1991, pp. 52-60.

[6] John Hennessy, Mark Heinrich and Anoop Gupta, Cache-Coherent Distributed Shared Memory: Perspectives on Its Development and Future Challenges, Proceedings of the IEEE, VOL. 87, No. 3, March 1999.

[7] Masaaki Mizuno, Michel Raynal, James Z. Zhou, Sequential Consistency in Distributed Systems, Proc. of the Int'l Workshop on Theory and Practice in Distributed Systems, October 1994.

[8] Pete Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. An evaluation of software-based release consistent protocols. Journal of Parallel and Distributed Computing, 29(2):126--141, September 1995.

[9] Vijay Karamcheti, Architecture and Programming of Parallel Computers, Lecture 11, Future Directions Project presentations: December, 1998

[10] Pete Keleher, Sandhya Dwarkadas, Alan Cox, and Willy Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In Proceedings of the 1994 Winter Usenix Conference, pages 115--131, January 1994.

[11] Pete Keleher, Lazy Release Consistency for Distributed Shared Memory, PhD thesis of Rice University, Huston, Texas, January, 1995.

[12] A. Judge, P.A. Nixon, V.J. Cahill, B. Tangney, S. Weber, Overview of distributed shared memory, October, 1998

[13]Sarita V. Adve, Kourosh Gharachorloo, WRL Research Report 95/7: Shared Memory Consistency Models: A Tutorial, September 1995.

[14] J. Carter, J. Bennet and W. Zwaenepoel, Techniques for reducing Consistency-Related Communication in Distributed Shared Memory Systems, ACM Transactions on Computer Systems, Vol. 13, No. 3, , pp 205-243, August 1995.

[15] S. Adve et al. A Comparison of Entry Consistency and Lazy Release Consistency Implementations. In IEEE HPCA, February 1996.

[16] Brian N. Bershad Matthew J. Zekauskas, Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors, Research on Parallel Computing, ARPA Order No. 7330, September 1991