

A Framework to Classify Distributed Shared Memory Systems

Debzani Deb and M. Muztaba Fuad

Department of Computer Science & Engineering
Shahjalal University of Science & Technology, Sylhet-3114, Bangladesh
E-mail: {banya-cse,fuad}@sust.edu

ABSTRACT

Software Distributed Shared Memory (DSM) systems combine the ease of shared memory programming with the efficiency of message passing architectures. However, there is several design and implementation issues related to DSM software that strongly affect their overall performance. Unfortunately there is still no clear understanding of these issues and their performance tradeoffs. This paper is a survey of the most recent years of DSM research and places the various design and implementation issues in a comprehensive framework while discussing their tradeoffs explicitly. The paper also classifies the prominent DSM research projects obtained so far according to this framework.

Keywords: DSM, Coherence Granularity, Memory Consistency model, Release Consistency.

1. INTRODUCTION

Parallel computing on clusters of workstations has been receiving more attention over the last few years [6,10,21,25,26]. Since workstation clusters use off the shelf processors, they provide a cheaper solution than traditional supercomputers. Moreover, high speed interconnection network and powerful workstation clusters are narrowing the performance gap between workstation clusters and supercomputers.

The two mainstream models of parallel computing, message passing and shared memory, show contradictory characteristics. Message passing requires simple architectural support and is a cost-effective solution, but makes the programming job tedious since programmers have to deal with the communication issues. On the other hand, by providing the abstraction of one large memory, the shared memory approach in tightly coupled architecture hides the communication issues and thereby makes programming simple. However, providing scalability is a complicated and expensive issue in the case of shared memory systems. Distributed shared memory [3,5,19,21] is an attempt to combine the scalability and inexpensiveness of message passing with the convenience of the shared memory programming model. A DSM system can be generally viewed as a set of nodes or clusters, each with its own memory connected by an interconnection network.

DSM systems implement a shared memory abstraction on top of the message-passing model in networks of workstations. In DSM, the programmer can write the program as if it is executing in a shared memory multiprocessor and can access shared data with ordinary read and write operations. The chore of message passing is left to the underlying DSM system to handle. It becomes the DSM's responsibility to decide when to send messages, what to send and whom to send to. However, this simple shared model visible to the programmer is generally

supported by a complex software structure. This is because, in the message-passing paradigm, communication between processes is handled by the programmer, who has complete knowledge of the data usage pattern of the application. On the other hand, in the DSM paradigm, the underlying system has very little knowledge of the application program, and therefore must take a conservative approach to determine when and how to communicate data.

The main problems that every DSM approach must address are: mapping of the logically shared address space onto physically distributed memory modules, locating and accessing a needed data item and preserving the coherent view of replicated data. Many interesting design and implementation issues with various tradeoffs have been considered during the last few years to address these problems, resulting in varieties of implementation. However, not enough study has been done on the appropriate categorization of these design and implementation issues and on defining their tradeoffs. Since the overall performance of the DSM system is highly dependent on these issues, users should have a clear understanding of the impact of each choice. Furthermore, misunderstanding and terminology confusion commonly present in this area often leads to incomplete and inaccurate knowledge.

This paper provides extensive coverage of these issues and tradeoffs, places the multi-track flow of ideas in a comprehensive framework and classifies the existing systems according to the framework. The DSM contributions obtained so far are classified according to four categories: coherence granularity, memory consistency models, writer algorithm and coherence policy. Each successive section of this paper is devoted to a category. The last section summarizes the categorization along with the tradeoffs and then classifies the existing DSM systems according to this categorization.

2. COHERENCE GRANULARITY

Granularity refers to the unit of sharing and data transfer across the network when a fault occurs. Selecting proper grain size is an important part of the design of a DSM system as it is a measure of the granularity of parallelism explored and the amount of network traffic generated by faults. Based on the grain size, DSM systems can be classified as page- and object-based DSMs.

2.1 PAGE-BASED DSMS

The first software DSM implementation IVY [21] used virtual memory pages as the basic coherency unit, which allows the use of existing page-fault schemes (i.e. virtual memory mechanism) to trigger a DSM page fault. In the page fault handler, the corresponding consistency protocol then fetches and acquires the requested page from a remote node by exchanging message with that node. Today, most DSM systems use virtual memory

page protection to detect writes to shared data and therefore use the memory page as the coherence grain [14,19]. These page-based systems are attractive because they allow the hardware to be used to help in the maintenance of consistency and they provide easier and simpler synchronization and programming models. Moreover, the ubiquity of the virtual memory system on a wide variety of operating systems and architectures allows an application to use shared memory in a transparent way using native compilers and libraries. Page-based systems generally perform well in the case of applications with good spatial locality since validating a single object on a shared page means validating all other objects on the same page. A large percentage of prefetched data is thus useful and therefore results in fewer page faults and lower consistency and communication requirements.

However, there are some disadvantages. First of all, virtual memory operations even in an optimized system have a high cost associated with them. Zekauskas et al [28] studies the use of software write detection at a lower level using compiler and run time extensions, citing the cost of hardware faults as a significant issue in many systems. Hosking and Moss [11] also study the issue and conclude that software traps have lower overhead than hardware traps. Secondly, pages are often the wrong unit for data management and cause unnecessary communication, since their size is defined by the architecture and operating system and this size usually does not necessarily match the data object size defined in the application. In consequence, page-based systems can transmit more data than is necessary when the size of the shared data structure is less than a page, or transmit data using more messages than is necessary when a shared data structure spans multiple pages.

The most serious problem of the page-based system is called *false sharing* and can reduce the performance of the overall system significantly. False sharing happens when two nodes access two unrelated variables that reside in the same page and at least one of them is a write. For instance, two user-defined objects are allocated on the same page and two processors P1 and P2 are accessing the page concurrently. As a result, the page has to travel across the network each time P1 wants to write in it when it is being held by P2 or vice versa. This problem occurs mainly because of the difference between the programmer's view of sharing and sharing at the DSM level and can lead to the thrashing problem [25].

2.2 OBJECT-BASED DSMS

Object-based DSM systems avoid using hardware assistance to detect shared data access and therefore are not bound to the virtual memory page as the unit of sharing. These systems generally use user-defined objects as the basic grain and therefore limit the scope of consistency action to only that object's extent. Object-based systems generally rely on the runtime system [1,15] or a combination of runtime and compiler instrumentation [3] to trap writes to shared data without invoking the operating system. In compiler instrumentation, the compiler emits extra instructions that set software dirty bits when the corresponding shared data is changed. Runtime-based systems generally explore the use of program level constructs to explicitly inform the system when shared objects are referenced. The benefits of object-based systems include less data transfer across the network than page-based approach, fine grain sharing

and elimination of the false sharing problem. Object-based systems can use a simple sequential consistency model without suffering false sharing. However, since access control and coherence are provided by instrumenting memory operations in the code rather than through the virtual memory mechanism, the object-based approach adds overhead for the programmer. Moreover, software write detection is not always faster than the virtual memory mechanism in systems with lower hardware overhead [28] and has a potential disadvantage. The overhead of software write detection occurs on every write at the time of multiple stores to the same item, where in page-based, only the first storage to the page should be detected. Finally, since object-based communication may be more frequent, the approach depends critically on low-latency messaging. Recent optimizations on modern architectures reduce instrumentation overheads [13] and message passing is becoming more and more inexpensive due to the significant network improvement. Both of these factors favor the object-based approach.

3. MEMORY CONSISTENCY MODEL

The simultaneous existence of multiple copies of the same data imposes the additional problem of keeping those copies consistent. In sequential consistency, a read always returns the value written by the most recent write, typically increase the memory access latency and the bandwidth requirements, while simplifying programming. On the other hand release consistency models, where memory becomes consistent only at synchronization points indicated by the programmer, allow reordering, pipelining and overlapping of memory which consequently results in higher performance. However, the increase in performance comes at the expense of the higher involvement of programmer in synchronizing accesses to shared data.

3.1 SEQUENTIAL CONSISTENCY

The most widely used form of consistency is called *sequential consistency (SC)* where synchronization accesses are treated as ordinary write and read operations. The sequential consistency model was first proposed by Lamport [20] and used in the first software DSM system, IVY. Sequential consistency enforces the constraint that the result of any execution has to be the same as if the operations of all the processes were executed in some sequential order. Sequential consistency in DSM systems can be implemented by ensuring that no memory operation is started until all the previous ones have been completed. A sequentially consistent memory provides single copy semantics because all the processes sharing a memory location always see exactly the same contents stored in it.

This form of consistency is attractive because it can be implemented with a reasonable effort, it supports the most intuitively expected semantics for memory coherence and it does not impose any extra burden on the programmers. The most important reason for its popularity is that a sequentially consistent DSM system allows existing multiprocessor programs to be run on multicomputer architectures without modification. However, this model restricts many hardware and compiler optimizations such as reordering, batching or coalescing that could increase performance [22] and therefore suffers from lower concurrency. Moreover, since this approach sends out messages whenever writing to a shared variable

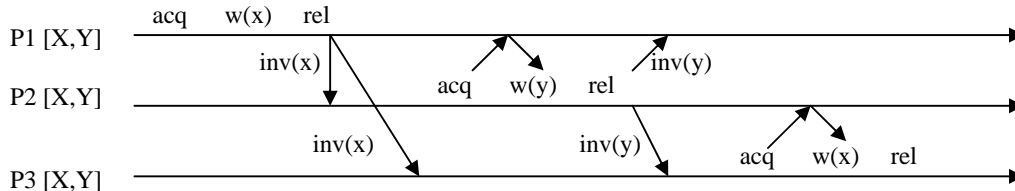


Figure 1: Eager release consistency

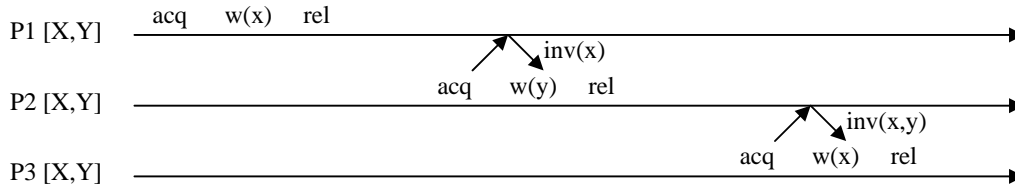


Figure 2: Lazy release consistency

that is also remotely cached, the frequency of communication is very high.

3.2 RELEASE CONSISTENCY

To cross the performance barrier imposed by sequential consistency model, *release consistency (RC)* models have been introduced [9]. Unlike the sequential consistency model, release consistency allows the propagation and application of coherence operations to be postponed until synchronization points, greatly reducing the frequency of coherence operations and the impact of false sharing. In the RC model, shared memory accesses are categorized either as *ordinary* or as *synchronization* accesses, with the latter category further divided into *acquire* and *release* accesses. RC requires ordinary shared memory updates by a process p to become visible to another process q only when a subsequent release by p become visible to q via some chain of mutual synchronization. In practice, this model allows a process to buffer multiple writes to shared data in its private memory until the release. Acquire is used by a process to tell the system that it is about to enter a critical section, so all changes made to the memory by other processes are propagated from other nodes to the process's node. On the other hand, release is used by a process to tell the system that it has just exited a critical section, so all changes made to the memory by the process are propagated to other nodes. However, programmers are responsible for locating acquire and release invocations at suitable places in their programs and therefore, RC programming model becomes more restricted and complicated.

Release consistency is a major breakthrough in the performance of DSM. Extensive research during past few years has resulted in many variations of release consistency [3,5,7,14,18]. The simplest one is *eager release consistency (ERC)* [5], which is a typical RC and performs the consistency operation eagerly at the release time. *Lazy release consistency (LRC)* [18] is a further enhancement, where instead of propagating modifications to the shared address space on each release (as in eager release consistency), modifications are further delayed until the next relevant acquire. Figure 1 and 2 show three processes exchanging synchronization in an eager and lazy release consistent DSM using an invalidate protocol. Each processor caches pages X and Y. In case of ERC, at first processor P1 modifies the page X and then releases a synchronization variable. According to the definition of ERC, this modification should be performed on every other processor in the system, therefore invalidate messages are sent to

processors P2 and P3 at release time. When processor P2 modifies the page Y and performs a release, invalidate message must again be sent to all other processors that cache the page.

On the other hand, in LRC, data transfer only occurs between the acquiring and releasing processor. In Figure 2, lock grant from P2 to P3 carries invalidation for both pages X and Y. Therefore, with ERC, a message needs to be sent to all other processors informing the change at release time. In other words it propagates all the local modifications globally before a local release. In contrast, LRC only passes consistency information between processes that synchronize with each other instead of global transmission. In addition, in LRC the consistency information is piggybacked to the lock grant message from the releasing to the acquiring process. The amount of data exchanged can be minimized in this way and so too the number of messages, as modification and lock acquires can be combined into a single message.

A number of other variants of release consistency have been suggested, each making the shared memory abstraction more complex and less transparent in return for better performance. *Entry consistency (EC)* [3,23] proposes the idea of binding shared data to synchronization variables using language level annotation and making only the data bound to that variable consistent at a synchronization event. The granularity in entry consistency is that of user defined objects rather than pages, which makes it fit well into object oriented languages. However, performance improvement is achieved at the expense of higher programmer involvement in specifying synchronization information for each shared data object.

Scope consistency (ScC) [14] tries to achieve the benefits of entry consistency without the burden of explicit binding. Synchronization variables define scope through which memory is viewed, and when write access occurs inside a scope, an implicit association of data with scope is achieved dynamically. Unlike EC, the granularity is page-based in ScC. Another modified release consistent DSM model is *version consistency (VC)* [7], which supports object-based granularity, where applications are permitted to acquire specific versions of objects. Versions are immutable and an acquire for a specific version will block until a corresponding release has been performed. In VC, a language-level object can be represented by a composition of fragment objects, in order to achieve fine-grain sharing, while in the other object-based systems language-level objects are the individual units of coherence.

4. WRITER ALGORITHM

Another important difference between protocols, whether lazy or eager, is whether they are *single writer* or *multiple writer* [27]. In multiple reader single writer (MRSW) protocols (read replication), multiple readers may coexist, but not multiple writers. The single writable copy of a page or object migrates along with its ownership from one writer to another. In the case of multiple reader multiple writer (MRMW) protocols (full replication), multiple writers can coexist and independently modify their copies of replicated page or object and at the synchronization time, communication and coherence can be done. A popular way of managing propagation of data modifications entirely in software for multiple writer protocols is to use *diffs* [17,18]. At the time of release, a writer determines the changes it has made to each page or object during the current synchronization interval by computing the differences (*diffs*) between the versions of the page or object at the beginning and at the end of the interval. The *diffs* can then be propagated to other nodes.

Single writer protocols are easy to implement since all copies of a given page or object are identical and a fault can be satisfied by any node, that has a valid copy. Unfortunately, this simplicity comes at the expense of higher message traffic due to the invalidation of each other copy during a write. The false sharing problem can be significant in the case of a single writer page-based system since the single writable page has to travel across the network each time one processor wants to access objects in the page while it is kept by another. Multiple writer protocols can minimize the effect of false sharing by allowing multiple processors to write simultaneously on the same page. However, this approach also has some potential problems. Firstly, to make a page or object consistent, a processor needs to obtain *diffs* from multiple writer nodes, generating numerous messages that increase the fault latency. Secondly, the need to retain *diffs* at the writer nodes can cause very large memory consumption, so periodic garbage collection is needed and scalability is greatly reduced. Thirdly, execution requires several memory intensive operations, which affect application performance. Finally, the *diffs* to be transferred can accumulate rapidly and increase communication traffic. Although the trend today is toward lazy and multiple writer protocols, these problems should be considered when choosing an appropriate protocol for an application.

5. COHERENCE POLICY

Another important distinction among protocols comes from the issues involved in maintaining the coherent view of shared data. It must be ensured that no processor reads stale data once a write has been completed on any processor. There are two main approaches: *write-update*, where a processor broadcast writes to all other processors those have copies of the same data, and *write-invalidate*, where the processor invalidates all other copies before doing an update. The choice of coherence policy is related to the granularity of shared data. For very fine-grained operations, the cost of an update message is approximately the same as the cost of invalidation. As a consequence, the update mechanism is generally used in systems with very fine - grained sharing. For coarse - grained

sharing, write update is usually considered too expensive since a broadcast may be needed on every write. On the other hand, invalid mechanism only propagates data when they are read and several updates can take place (because of locality of reference property of many applications) before communication is necessary. Therefore, write invalidation mechanisms with *copy set*, a list of processors with a copy of the data, are very popular for coarse-grained systems [16]. There are schemes available to compress and represent very large copy sets in large DSMs. However, if the write is allowed to proceed before the invalidation is complete there will be a violation of consistency model, so invalidation needs to be acknowledged.

6. DESTINATION OF PROPAGATION

Protocols can also be classified according to the destination of propagation, such as *home-based protocols* [12] and *non-home-based protocols*. In the former, a home node is selected for each shared page or object at the beginning of execution and modifications are propagated eagerly to the home only. On a miss, a non-home processor obtains the whole page or object from the home rather than from the previous writer. Since in most software DSM systems *diff* is used for data propagation, in a home-based system, the *diffs* created at release time are immediately sent to the home and then discarded. At the home node, *diffs* are applied as soon as they received and discarded too, so *diff* storage requirements are small in home-based protocols. The other advantages are firstly, a *diff* is applied only once (at home) instead of applying on multiple processes which ask for the modification, secondly, writes performed at the home node do not propagate *diffs* and, finally, there are no remote data fetches at the home. The last two factors can make a large difference when a page or object is mostly written by a single processor. The potential disadvantage is, substantial performance degradation can result for the applications where homes are not assigned properly and for the applications with migratory sharing pattern. On the other hand, non-home-based protocols use the usual invalidation model to maintain coherency and generally work well for varied set of applications.

7. SUMMARY

The classification discussed above is the result of a survey of the most active years of the DSM research. This classification is based on different aspects of DSM implementation techniques. Choosing an appropriate implementation technique in DSM is always a tradeoff between different aspects. The multidimensional classifications along with the tradeoffs are summarized as follows.

From coherence granularity standpoint, DSM systems are classified as page- and object-based. The page-based systems offer some hardware assistance in this complex process of DSM implementation in return for lower performance due to the false sharing. However, in locality-based applications, large granularity can get the prefetching advantage, which eventually decreases the number of page faults. Object-based systems eliminate false sharing problem by defining the sharing unit as user-defined object but needs more programmer involvement in annotating accesses to shared data. So choosing an

Characteristics	Classification	IVY [21]	Mirage [8]	Mermaid [29]	Clouds [24]	Orca [1]	[7]	Amber [4]	CRL [15]	[14]	Munin [5]	Midway [3]	DiSOM [23]	RTL [2]	TreadMarks [19]
Coherence Granularity	Page based	✓	✓	✓											✓
	Object based				✓	✓		✓	✓		✓	✓	✓	✓	
Consistency Model	Sequential	✓	✓	✓	✓			✓	✓						
	Release	Eager										✓			
		Lazy												✓	✓
	Entry											✓	✓		
	Scope									✓					
	Version						✓								
Writer Algorithm	Single writer	✓	✓	✓	✓	✓		✓	✓		✓				
	Multiple writer										✓	✓		✓	✓
Coherence Policy	Invalidate	✓	✓	✓					✓					✓	✓
	Update					✓					✓	✓	✓		✓

Table 1: Classification of existing software DSM systems

appropriate granularity is a critical issue and depends on several factors.

According to consistency model, DSM systems mainly can be categorized as sequential consistency and release consistency. Sequential consistency supports most intuitively expected semantics for memory coherence and allows the existing multiprocessor programs to run on multicomputer architectures without modification. However, this model causes higher communication and lower concurrency. Release consistency model, which use explicit synchronization variables, provides better concurrency and also support the intuitively expected semantics. However, it requires the programmers to use the synchronization variable properly, which impose an overhead for the programmer. There are varieties of release consistency models, each tries to improve the performance by increasing the amount of relaxation in memory access order constraint. This performance improvement is achieved in return for more complexity and less transparency. Therefore, choosing an appropriate consistency model is a tradeoff between increasing performance and complexity of the programming model as well as memory model itself.

According to the implementation of replication strategy, DSM systems can be categorized as single writer and multiple writer. Single writer gives a single copy semantics by allowing a single writer to modify any given page at a time and provides a straightforward way of implementation. An application, where a page or object is mostly written by a single node has a performance advantage in this case. However, disadvantages are, poor performance in the face of false sharing and higher bandwidth requirements since small modification can no longer be summarized in a diff rather than sending the entire page or object. On the other hand, multiple writer allows multiple writer nodes to simultaneously access the same page without communication. Here, diffing is used to summarize modifications and to resolve multiple concurrent modifications to the same page. However, this form of higher concurrency support becomes possible in return for more complexity and

additional memory requirements due to diff storage. Therefore, deciding a writer algorithm is a tradeoff between concurrency and complexity along with memory overhead.

The other main issue in maintaining a strictly consistent data set is synchronization of write accesses to the data. There are two main approaches: write update and write invalidate. In write update, a node broadcasts writes and in write invalidate a node invalidates all other copies before doing an update. The choice critically depends on the size of coherence granularity. In general, update policy is used with fine-grained approach since the cost of broadcasting a write is approximately equal to the cost of sending invalidation. On the other hand, invalidation is appropriate for coarse-grained operation since updates are only propagated when it is necessary. Locality-based application can get the advantage of invalidate mechanism since several modifications can take place before propagation. Therefore, choosing an appropriate synchronization mechanism critically depends on granularity size and locality and applications specific issues.

The above framework unifies many aspects about DSM implementation. To illustrate the usefulness of this framework, available software DSM systems have been categorized according to this multidimensional classification in Table 1. This framework allows the user to have a clear idea about the implementation techniques used in these DSMs and compare and reason about their performances.

8. CONCLUSIONS

This paper presents a comprehensive framework to describe, understand and compare distributed shared memory implementations and their performance issues. This framework is based on several issues such as coherence granularity, consistency models, writer algorithm, coherence policy and destination of data propagation. Tradeoffs related to each of these issues are also discussed explicitly. Therefore, the user can have a clear idea about the implications of each choice. To

overcome the frequent terminology confusion and misunderstanding in this area, existing DSM systems are classified according to this framework.

REFERENCES

- [1] H. E. Bal, M. F. Kaashoek & A. S. Tanenbum, "Orca: A language for Parallel Programming of Distributed Systems", *IEEE Transactions on Software Engineering*, vol. 18, no.3, pp. 190-205, March 1992.
- [2] T. Brecht & H. Sandhu, "The Region Trap Library: Handling Traps on Application-Defined Regions of Memory", *In the Proceedings of the 1999 USENIX Annual Technical Conference*, June, 1999.
- [3] B. N. Bershad, M. J. Zekauskas & W. A. Sawdon, "The Midway Distributed Shared Memory System", *In Proceedings of The '93 CompCon Conference*, pp. 528-537, February 1993.
- [4] J. S. Chase, F. G. Amador, E. D. Lazowska, H. M. Levy, & R. J. Littlefield, "The Amber system: Parallel programming on a network of multiprocessors", *In Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pp. 147-158, December 1989.
- [5] J. B. Carter, J. K. Bennett & W. Zwaenepoel, "Implementation and Performance of Munin", *In Proceedings of The Thirteenth Symposium on Operating Systems Principles*, pp. 152-164, October 1991.
- [6] Cheriton, D. R., "The V Distributed System", *Communications of the ACM*, vol. 31, no. 3, pp. 314-333, 1988.
- [7] M. J. Feeley & H. M. Levy, "Distributed Shared Memory with Versioned Objects", Technical Report UW-CSE-92-03-01, Department of Computer Science and Engineering, University of Washington, March 1992.
- [8] B. Fleisch & G. Popek, "Mirage: A Coherent Distributed Shared Memory Design", *In Proceedings of 14th ACM symposium on Operating System Principles*, pp. 211-223, 1989.
- [9] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta & J. Hennessy, "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors", *In the Proceedings of the 17th Annual Symposium on Computer Architecture*, pp. 15-26, 1990.
- [10] W. Gropp, E. Lusk, & A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, MIT Press, Cambridge, Mass, 1994.
- [11] A. L. Hosking & J. E. B. Moss, "Protection traps and alternatives for memory management of an object oriented language", *In Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, ACM Operating Systems Review, vol. 27, no. 5, pp. 106-119, December 1993.
- [12] L. Iftode, "Home-based Shared Virtual Memory", Ph.D. Thesis, Princeton University, June 1998.
- [13] L. Iftode & J.P. Singh, "Shared Virtual Memory: Progress and Challenges", Technical Report 552-97, Princeton University, Computer Science Department, pp. 87-100, October 1997.
- [14] L. Iftode, J. P. Singh & K. Li. "Scope Consistency: A Bridge between Release Consistency and Entry Consistency", *In proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '96)*, pp. 277-287, June 1996.
- [15] K. L. Johnson, M.F. Kaashoek & D.A. Wallach, "CRL: High-Performance All-Software Distributed Shared Memory", *In Proceedings of the 15th Symposium on Operating Systems Principles*, pp. 213-228, 1995.
- [16] A Judge, P.A Nixon, V.J Cahill, B. Tangney & S. Weber, "Overview of Distributed Shared Memory", Technical Report dsG-112, Distributed Systems Group, Department of Computer Science, Trinity College Dublin, October 1998.
- [17] P. J. Keleher, "The relative Importance of Concurrent Writers and Weak Consistency Models", *In Proceedings of the 16th International Conference on Distributed Computing Systems*, May 1996.
- [18] P. Keleher, A. L. Cox, & W. Zwaenepoel, "Lazy Release Consistency for Software Distributed Shared Memory", *In Proceedings of the 19th Annual International Symposium on Computer Architecture*, pp. 13-21, May 1992.
- [19] P. Keleher, S. Dwarkadas, A. Cox & W. Zwaenepoel, "TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems", *In Proceedings of The 1994 Winter Usenix Conference*, pp. 115-131, January 1994.
- [20] L. Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocessor Programs", *IEEE Transactions on Computers*, vol. 28, no. 9, pp. 690-691, 1979.
- [21] K. Li. "IVY: A Shared Virtual Memory System for Parallel Computing", *In Proceedings of the 1988 International Conference on Parallel Processing*, pp. 94-101, August, 1988.
- [22] D. Mosberger, "Memory Consistency Models", Technical Report No. TR93/11, Department of Computer Science, University of Arizona, 1993.
- [23] N. Neves, M. Castro & P. Guedes, "A checkpoint Protocol for an Entry Consistent Shared Memory System", *In Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*, pp. 121-129, August 1994.
- [24] U. Ramachandran & M.Y.A. Khalidi, "An Implementation of Distributed Shared Memory", *Software Practice and Experience*, vol. 21, no. 5, pp. 443-464, May 1991.
- [25] P. K. Sinha, *Distributed Operating Systems Concepts and Design*, IEEE Press, 1996.
- [26] V. Sunderam, "PVM: A Framework for Parallel Distributed Computing", *Concurrency: Practice and Experience*, vol. 2, no. 4, pp. 315-339, December 1990.
- [27] M. Stumm & S. Zhou, "Algorithms Implementing Distributed Shared Memory", *Computer*, vol. 23, no. 5, pp. 54-64, May 1990.
- [28] M. J. Zekauskas, W. A. Sawdon, and B. N. Bershad, "Software Write Detection for Distributed Shared Memory", *In Proceedings of the 1st OSDI Symposium*, pp. 87-100, November 1994.
- [29] S. Zhou, M. Stumm & T. McInerney, "Extending Distributed Shared Memory Heterogeneous Environments", *In Proceedings of 10th International conference on Distributed Computing Systems*, pp. 30-37, 1990.