An Approach for a Multi-LAN DSM Based on Lazy Release Consistency

Luciana Arantes*, Bertil Folliot and Pierre Sens
Laboratoire d'Informatique de Paris 6
Université Pierre et Marie Curie
e-mail : [arantes,folliot,sens]@src.lip6.fr

Abstract: ***In this paper we present our proposal for a multi-LAN lazy release consistency (LRC) DSM. LRC protocol usually uses vector clocks to control the causality of shared memory accesses. The use of such clocks limits to some extent the scalability of current LRC DSM systems, since their size is proportional to the number of nodes of the system. In order to offer a scalable and modular solution, we introduce a new logical clock, the barrier-lock clock, whose size varies with the number of locks used by the DSM application. The barrier-lock clocks provide support for implementing a multi-LAN DSM as an interconnection of per LAN DSMs. A hierarchical approach is given to the LRC protocol.***

1 Introduction

Distributed shared memory (DSM) systems simulate a shared address space on top of physically distributed memory systems. It is a very attractive support for parallel application since the shared memory paradigm simplifies programming development when compared to the message-passing one. However, the majority of DSM systems have been developed for local area networks (LANs) or symmetrical multiprocessors (SMPs). Considering that the number of machines of a LAN is relative low and the ever-growing improvement in networking technology, why do not span a DSM over an interconnection of multiple LANs? Such platform is extremely interesting as it provides more computing power, larger amount of physical memory, and the possibility of profiting from the idleness of underutilized remote machines. On the other hand, the cost of sending a message between processors belonging to two distinct LANs is much higher than between processors within the same LAN. The large inter-LAN latency and, sometimes, low bandwidth can strongly limit application performance or even the feasibility of the DSM itself. Because of this gap between intra-LAN and inter-LAN links, we could think that a multi-LAN DSM is not a platform for all kind of parallel applications, but only for those that present a coarse grain of parallel computation, a strong locality of data access references, and scarce synchronization operations. Undoubtedly, these are the ideal ones. However, some recent works [3] have shown that with appropriate optimizations, which take into account inter-LAN latency and bandwidth restrictions, some medium-grain shared-memory parallel programs can achieve performances compared to moderate-scale DSM ones.

This paper presents our proposal for a multi-LAN DSM system. We have chosen the lazy release consistency (LRC) model [8] because it presents good performance on top of LANs. In our multi-LAN DSM, every LAN executes

*PhD. scholar from CAPES (Brazil)

a copy of a LRC DSM, adapted for a multi-LAN platform. In principle, it is less costly to request a consistency information or shared data from a process that belongs to the same DSM than from one that belongs to a remote DSM. This concept allows the multi-LAN DSM to adopt a hierarchical approach, which is founded on the LRC consistency memory model itself, i.e., the LRC protocol is applied at both processor and LAN level. A processor profits to the utmost from the updated data already available within its local LAN, asking to a remote LAN processor only the ones that any of the processors of its local LAN does not have.

Most of current LRC-based systems employ the concept of vector clocks [5] [11] for controlling memory update partial ordering. A vector clock has an entry for each node of the system. This means that in order to be able to connect several per LAN DSMs we would need a larger vector clock, whose size would comprise the total number of nodes of all LANs. This would restrict the scalability of the multi-LAN DSM. Moreover, in order to be able to control transitive propagation of shared data updates, LRC protocol keeps in memory not only its own vector clock timestamps, but also the timestamps received from other processors which are afterwards included in many of the messages exchanged by the processors. So, larger size of vector clocks results in more processor memory storage requirements and amount of data sent across the network. We can also point out that the configuration of a multi-LAN platform can dynamically change (e.g., temporary inaccessibility of remote LANs). This dynamic reconfiguration entails the adding or disabling of vector clock entries which may lead to vector clock comparison incompatibility. Considering all these restrictions, we propose a logical clock, whose size is independent of the number of nodes of the system. We have named it the *barrier-lock* clock. Its size is proportional to the number of synchronization variables employed by the application, which is frequently small for DSM applications. The replacement of the traditional vector clocks by the *barrier-lock* ones allows the inter-connection of several DSMs, as well as the minimization of the just mentioned problems related to memory, larger size of synchronization messages, and dynamic reconfiguration. Based on the *barrier-lock* clocks, we have put forward our idea of a hierarchical LRC.

Section 2 of this paper gives an overview of the lazy release consistency protocol. In section 3, we present the *barrier-lock* clocks and some performance measures obtained with a first LRC DSM prototype, which employs the *barrier-lock* clocks. The hierarchical and modular approach of the LRC protocol is described in section 4. This latter also includes a discussion of other important points that must be considered in order to provide satisfactory application performance. In section 5, we present some related works while the last section summarizes the contributions of this work.

2 Lazy Release Consistency Model

In lazy release consistency memory model, originally defined by TreadMarks [1] [8], ordinary memory accesses are distinguished from synchronization ones, i.e., the acquire and release operations. An acquire operation is used to gain access to shared variables while a release operation is used to grant them. Information related to the updates made by a process on its local page copies are propagated out to a second process only when this latter performs an acquire operation. This postponement reduces much of the communication that is required to make shared data consistent. If the DSM adopts an invalidate protocol, as in the case of TreadMarks, the updates themselves are only sent when the acquiring processor accesses one of these pages. In this section and the following ones, we consider that a given processor runs only one DSM process which means that the words processor and process are interchangeable.

The execution of each process is divided into intervals. A new interval begins at each acquire or release operation. Intervals define an order between the acquire/release operations, i.e., they set up the causality of shared memory updates. Operations on a lock and barrier can be mapped onto acquire/release operations: a lock operation corresponds to an acquire, an unlock to a release while an arrival at a barrier can be modeled as a release and the departure from it as an acquire operation. Partial ordering of the intervals is controlled by assigning a per processor vector timestamp to each interval. Each processor P_j keeps a vector clock v_j of N entries, where N is the total number of processors of the system. P_j controls the intervals created by itself using the *jth* entry of its vector clock. The other entries store the current knowledge that this processor has of the updates made by other processors. Processor P_j updates its vector clock v_j, at each synchronization operation, as follows: it adds 1 to the *jth* entry of its vector clock ($v_j[j]$); if it is a remote acquire operation, P_j updates its vector clock with the maximum of its value and the releaser's v_r, i.e., $v_j = max(v_j, v_r)$. This last operation means that processor P_j, at an acquire operation, sends its current vector value to the releaser processor P_r. This latter, then, sends back all the intervals "covered" by P_r vector clock but not by P_j's, including the identification of the pages that have been modified in each interval. Each identification is stored in a structure called *write-notice*. When receiving a *write-notice*, the acquiring processor invalidates its corresponding local page (invalidate protocol). The first access to an invalid page will cause a page fault. The faulting processor will then ask for all the updates that it does not have yet, applying them in the order defined by the causality of the intervals. These updates come in the form of *diffs*, a word-by-word comparison between a copy of the original page and its last version. Each *diff* is associated to a *write-notice* which in its turn is associated to an interval. If a processor has modified a page during an interval, than it has all the *diffs* of the intervals that precede the former interval. Thus, the faulting processor needs to send *diff* requests only for those processors for which their most recent interval is not covered by the most recent interval of any processor [1].

3 Lazy Release Consistency Implementation based on Barrier-lock Clocks

At a barrier, each processor sets its local vector clock to the same value, incorporating consistency information from other processors. Thus, we could think of dividing the execution of a program into *barrier-intervals*. Each *barrier-interval* could then be subdivided into *lock-intervals*. A new *barrier-interval* would begin at each barrier call while a *lock-interval* at each acquire or release operation on a lock. This is exactly the idea behind the *barrier-lock* clock. The original vector clock timestamp v_i of a processor P_i is replaced by the timestamp (b_i, vl_i), where b is a barrier call counter and vl a per lock vector variable. At each barrier call, the scalar barrier counter b of all processors is incremented while their lock vector variable vl is reset. In this way, b catches the causality between two barrier calls.

For controlling the lock movement among the processors within a *barrier-interval*, we use the partial order *poset-diagram*, presented by Mattern [11]. This diagram represents the logical relationship of events. Applied to *lock-intervals*, the diagram shows that it is possible to follow a "path of causality" from a lock acquire/release operation to another if they are causally related. As locks are used to control processors' accesses to shared data critical region, lazy release consistency protocol assures that only one processor holds a single lock at a time. Its behavior is like a token that can only be held by a single processor at a time. This implies that if two lock acquire/release operations are concurrent, they correspond to acquire/release operations on distinct locks. Therefore, for stamping *lock-intervals*, each entry of the vector vl can be associated to a lock and not to a processor. An entry is monotonically increased at each acquire/release operation. Due to this "lock token" behavior, it is possible to guarantee that the same entry of 2 different vectors vl are not concurrently incremented and, consequently, *lock-interval* partial ordering is precisely captured. Thus, processor P_j, at each acquire/release operation of lock l, updates its vl vector variable as follows: if it is a remote acquire operation, $vl_j = max(vl_j, vl_r)$, where vl_r is the per lock vector vl of the release processor P_r; P_j always adds 1 to the entry of vl_j that corresponds to lock l ($vl_j[l]$).

Notice that a *barrier-lock* clock size is $L+1$, where L is the number of locks employed by the application. It is then proportional to the number of locks, independently of the number of nodes of the system. Similar to the original LRC, *diffs* and *write-notices* are associated to intervals. As usually the number of locks used by a DSM application is not very high, the *barrier-lock* clocks, as opposed to the traditional vector clocks, can considerably reduce the size of clock timestamp, particularly if the number of nodes of the system is important. Naturally, our solution is less appropriate for applications that employ a huge number of locks.

Figure 1 shows the partial ordering, corresponding to the acquire and release operations of 2 locks made by 5 processors. A barrier is called, defining a new *barrier-interval.* Read or write accesses to shared memory or the sending of *write-notices* and *diffs* are not shown for the sake of simplifying the figure. The notations $a_{ij}(l)$ and $r_{ij}(l)$ respectively represent the jth acquire or release operation performed by processor P_i on lock l. The timestamp of each interval is also indicated by the notation $b - vl[0], vl[1]$.

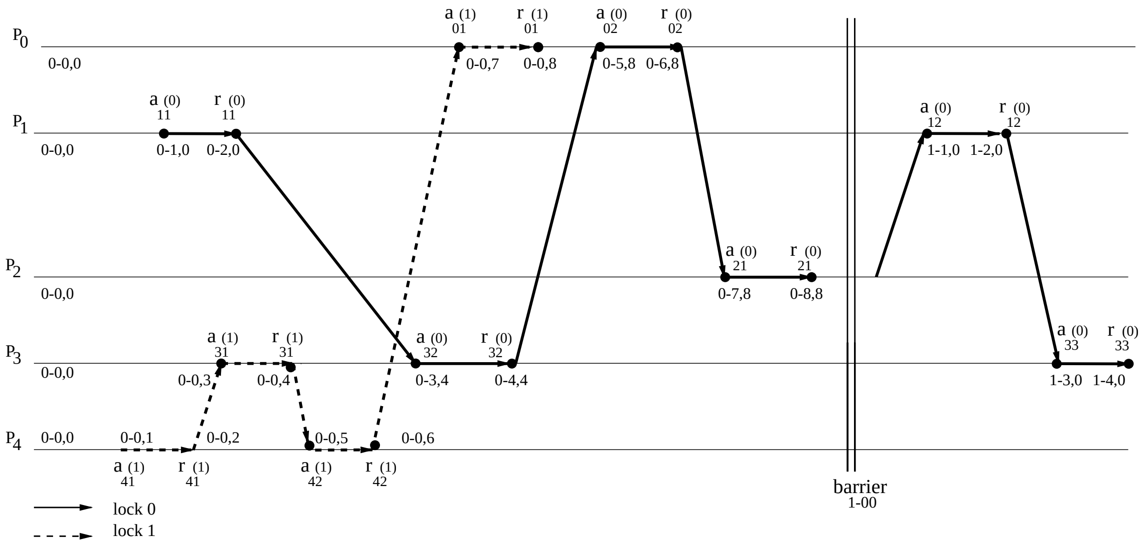

Figure 1: LRC based on barrier-lock clocks

In order to verify the feasibility of the *barrier-lock* clocks, we have developed a prototype by replacing the traditional vector clocks by the *barrier-lock* clocks in TreadMarks DSM, version 0.10.1. We used the TSP and SOR applications in order to evaluate our implementation. TSP performs a considerable number of remote acquires on 2 locks and rarely calls a barrier while SOR does not use any lock at all, but it calls a barrier a great number of times. The tests have been made on top of 8 Sunsparc-5 stations linked by a 100 Mbit/s Ethernet backbone. For showing the scalability of of the *barrier-lock* clocks, we have simulated a platform with different number of nodes by varying the constant that specifies the number of processors in the system. This directly implies that the size of the traditional vector clock increases, but not the size of the *barrier-lock* clock. We have then measured the number of bytes exchanged between the processors at synchronization operations and the amount of memory that TreadMarks and the *barrier-lock* LRC prototype consume for keeping partial ordering of the intervals. The number of messages exchanged between the processors is the same for both DSMs. Figure 2 shows the ratio: *number of bytes of synchronization messages on top of TreadMarks / the same number of bytes on top of the barrier-lock clock prototype.* The same ratio is shown for the total amount of memory requested by those LRC data structures that control interval causality and are directed influenced by the size of the logical clock.

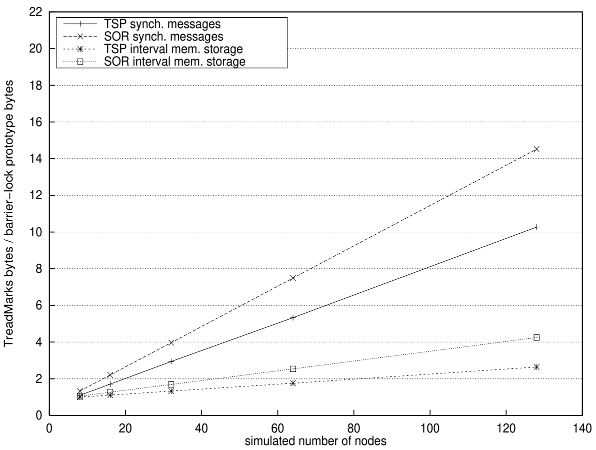

Figure 2. *Synch msg. size and memory.*

We can remark a considerable grow in the size of the synchronization messages as well as the amount of memory, specially if there is a huge number of nodes, as it may happen in a multi-LAN platform. We have not observed any significant gain in performance. Both TSP and SOR have a high computation to communication ratio [9] and the cost of sending a message is too high compared to the per byte cost. However, for platforms where inter-LAN links have a low bandwidth, a reduction in synchronization messages size can imply in better performance for those applications that present a high communication to computation ratio.

4 Hierarchical Lazy Release Consistency approach

In our multi-LAN LRC DSM, as a LAN level cache is simulated, each LAN behaves like a physically shared-memory SMP node. Basically, the idea is to apply the lazy release consistency hierarchically, i.e., both at processor and LAN levels. All available partial ordering information, *diffs* and copies of pages within a local LAN are profit to the utmost as an attempt to avoid inter-LAN message exchanges. When a processor makes an acquire to a releaser that belongs to a different LAN, the LRC principle is applied at LAN level: the acquiring LAN informs the releasing LAN about all the partial ordering information that its processors have already received. Thus, the acquiring processor receives from the remote releasing processor only the information that its LAN (i.e., any of the other local processors) does not have. The intra-LAN available causality information that is missing to the acquiring processor is directly sent to it by the local processors that own them (in fact the last processors which grant locks to other LANs).

For illustrating the multi-LAN LRC hierarchy idea, we could consider that processors P_0 and P_1, of figure 1 belong to a LAN (LAN_0) while P_2, P_3 and P_4 belong to a second one (LAN_1). For instance, at the acquiring operation $a_{21}(0)$, processor P_2 requests lock 0 whose releaser, P_0, belongs to another LAN (LAN_0). Although P_2's *barrier-lock* clock has, at this moment, the value 0-0,0, LAN_1 informs LAN_0 that it keeps *lock-interval* causality information till 0-4,6 (information held by P_3 and P_4). Thus, processor P_0 will only send to P_2 the causality information that is bigger than this value (*lock-intervals* 0-0,7; 0-08; 0-5,8; 0-6,8). The part of the causality information that is missing to P_2 is directly sent to it by P_3 and P_4, i.e., the last processors of LAN_1 that have granted locks to a processor of a remote LAN. This principle is also used for the requesting of *diffs* whenever a processor gets a page fault. Obviously, the faulting processor must apply the *diffs* to the page, respecting the partial order sequence of the related intervals. However, as the first *diffs* to be applied are those from the local LAN, the requesting processor will, with great probability, receive them first than the ones from a remote LAN. The reduction of inter-LAN data transfer and the overlap of *diff* requests can substantially minimize the problem of *diff accumulation*. *Diff accumulation* occurs in the case of migratory data, i.e., when the same shared data, protected by the same lock, are sequentially updated by a set of processors [2]. When the lock holder processor gets a page fault, it will receive the sequence of all *diffs* since its last acquiring of this lock. Even after applying the *diffs*, the processor has to keep them for retransmitting them for the next acquirer. The *diff accumulation* problem can significantly increase communication and memory consumption.

4.1 Prefetching of updates

Considering the inter-LAN latency restriction, the number of messages exchanged between LANs should be reduced to the maximum. Invalidate protocol, as described in section 2, tries to minimize the amount of data transfered across the network by sending the *diffs* of a page only when they are requested by a processor, i.e., when this latter incurs a page fault. However, this laziness in propagating the updates implies in extra messages, if we consider that the *diffs* could have been included in the releaser's answer message to the acquirer's request. So, instead of including only the *write-notices* in the message to an acquiring LAN, the releasing LAN could coalesce with this message the *diffs* that it believes the acquiring LAN will demand in the near future. This protocol is called *lazy hybrid* protocol [9]. Good prediction of future accesses of remote LAN processors can significantly decrease the number of access misses that these processors would get and, consequently, the number of messages sent across the inter-LAN link. On the other hand, bad prediction results in useless data sent across the inter-LAN link. Several works have proposed some heuristics for data prefetching or for the use of update protocol instead of the invalidate one [2] [9] [12]. For instance, a good heuristic for programs with lock-protected migratory data is to send to an acquiring processor those *diffs* which this latter does not have and which correspond to updates protected by the lock being acquired [12]. In the case of the hierarchical LRC, the releasing LAN will send to the acquiring LAN all the "lock-protected *diffs*" which any processor of the acquiring LAN possesses. This may imply that the releasing LAN (in fact the processor that will grant the lock) will need to gather *diffs* from different local processors before sending them to the acquiring LAN. The ideal is to send them in a single message, as discussed in the comparative performance study made between PVM and TreadMarks as described in [10]. The authors point out that it is a better strategy to aggregate a large amount of data in a single message than to separate them in several messages. For barrier-like programs, a good heuristic may be the set of *diffs* whose pages were modified by the majority of processors of the remote LAN in the previous barrier call. It is worth mentioning that these heuristics can be easily implemented using the *barrier-lock* clock and its concept of *lock-interval* and *barrier-interval*.

4.2 Open issues

Obviously there are many other problems that must be tackled in order to provide an efficient multi-LAN LRC DSM. Inter-LAN latency and bandwidth can directly influence application performance. Therefore, operations like global synchronization (e.g., barriers, garbage collection mechanism), lock serialization and initial distribution of valid pages must be reconsidered.

The idea of implementing the execution of a barrier hierarchically, as presented in HLRC-SMP [13], is quite interesting and appropriate for an inter-LAN DSM. Barrier or garbage collector global synchronization time can be significantly reduced as the inter-LAN traffic is reduced as well.

At the startup phase, a DSM usually assigns all the valid pages to the same processor. So, at the first access to a page, any other processor has to request the page to this owner, which can belongs to a remote LAN. A more equally or heuristic-based redistribution of pages among the processors could minimize the number of these costly remote requests.

Hierarchy of shared memory and locks could also be proposed, i.e., locks and shared variables that are local to a LAN and others that are global to all of them. They would be very suitable for cooperative applications, which in their turn are quite adequate for a multi-LAN platform. A hierarchy of locks will reduce the size of synchronization and *diff* request messages that cross the inter-LAN link. In order to minimize the problem of lock serialization, we could also think of having, opposed to the original TreadMarks, a dynamic per lock manager. A manager of a given lock could be assigned to another processor of a remote LAN whenever the number of subsequent requests for the lock, made by the local processors of this remote LAN, achieved a certain threshold.

5 Related Works

TreadMarks [8] was the first DSM to implement the lazy release consistency memory model. Automatic Update Release Consistency (AURC) [6] and the Home-based Lazy Release Consistency (HLRC) [15] from Princeton University are variations of the LRC protocols. The three of them employ vector clocks to ensure the causality of the acquire/release operations. The last two adopt a "home-based" protocol in which updates of a page are eagerly propagated to a home node associated to this page. This propagation is either supported by a hardware automatic update (AURC) or is all-software (HLRC).

Several works [4] [13] [14] describe the extension of a uniprocessor software DSM to networks of SMPs. Contrary to a LAN, a SMP node provides a physical shared memory that is automatically shared by all the processors within a single SMP node. As a consequence of this physical sharing, the performance of a SMP-network DSM is expected to be better than the performance of its equivalent uniprocessor DSM. In HLRC-SMP [13], a SMP node caches the list of pages updated by the processors of a SMP node in structures called *bins*. *Bins* are stored in the SMP physical shared memory and, therefore, all processors of the SMP can have access to them. When an acquiring processor receives a new *bin* from a distant SMP node releaser, this *bin* is automatically available to all processors within the same SMP. On the other hand, processors have separate page tables. In this way, each processor can keep its own view of a given page.

6 Conclusions

In this paper we have presented the *barrier-lock* clocks. The results obtained with our first prototype show that the *barrier-lock* clocks can be employed for precisely capturing the partial order of synchronization operations of lazy release consistency protocols. As *barrier-lock* clocks present a very good scalability and provide modularity, they will be used by our multi-LAN DSM prototype for the implementation of the hierarchical LRC protocol.

It is worth mentioning that the *barrier-lock* clocks are extremely adequate for implementing *scope consistency* [7], as the concept of a scope is associated to critical regions and synchronization variables. Scope consistency relaxes a bit more the LRC consistency as it associates, not explicitly, shared data to synchronization variables. A consistency scope can be seen as the set of all critical sections protected by the same lock or as a global one defined by a barrier call. This concept can respectively be mapped to *lock-intervals* and *barrier-intervals*.

References

[1] C. Amza, A. L. Cox, S. Dwarkadas, and P. Keleher. TreadMarks: Shared memory computing on networks of worksations. *IEEE Computer*, 29(2):18–28, February 1996.

[2] C. Amza, L. Cox, S. Dwarkadas, and W. Zwaenepoel. Software DSM protocols that adapt between single writer and multiple writer. In *Third International Symposium on High Performance Computer Architecture*, pages 261–271, February 1997.

[3] H. Bal, A. Plaat, M. Bakker, P. Dozy, and R. Hofman. Optimizing parallel applications for wide-area clusters. In *12th International Parallel Processing Symposium*, April 1998.

[4] A. Cox, Y. Hu, H. Lu, and W. Zwaenepoel. OpenMP on networks of SMP. In *13th International Parallel Processing Symposium*, April 1999.

[5] C. Fidge. Logical time in distributed computing systems. *IEEE Computer*, pages 28–33, July 1991.

[6] L. Iftode, C. Dubnicki, E. Felten, and K Li. Improving release consistency shared virtual memory using automatic update. In *2nd IEEE Symposium on High-Peerformance Computer Architecture*, February 1996.

[7] L. Iftode, J.P. Singh, and K. Li. Scope consistency: A bridge between release consistency and entry consistency. In *8th Annual on ACM Symposium on Parallel Algorithms and Architectures*, June 1996.

[8] P. Keleher. *Lazy Release Consistency for Distributed Shared Memory*. PhD thesis, Texas University, Texas (EUA), 1995.

[9] P. Keleher, A.L. Cox, Dwarkadas, and W. Zwaenepoel. An evaluation of software-based release consistent protocols. *Journal of Parallel and Distributed Computing*, 29:126–141, October 1995.

[10] H. Lu, S. Dwarkadas, A.L. Cox, and W. Zwaenepoel. Quantifying the performance differences between PVM and TreadMarks. *Journal of Parallel and Distributed Computation*, 43(2):65–78, June 1997.

[11] F. Mattern. Virtual time and global states in distributed systems. In *Workshop on Parallel and Distributed Algorithms*, Elsevier (Holland), October 1988.

[12] L.R. Monnerat and R. Bianchini. Efficiently adapting to sharing patterns in software DSMs. In *Fourth International Symposium on High Performance Computer Architecture*, February 1998.

[13] R. Samanta, A. Bilas, L. Iftode, and J. Singh. Home-based SVM protocols for SMP clusters: Design and performance. In *4th Symposium on High Performance Computer Architecture*, February 1998.

[14] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, H. Kontothannassis, Parhasarathy, and M. Scott. Cashmere-2L: software coherent shared memory on a clustered remote-write network. In *16th ACM Symposium on Operating System Principles*, pages 170–183, October 1997.

[15] Y. Zhou, L. Iftode, and K. Li. Performance evaluation of two home-based lazy release consistency protocols for shared virtual memory systems. In *2nd Symposium on Operating Systems Design and Implementation (OSDI 96)*, October 1996.