# Making a DSM Consistency Protocol Hierarchy-Aware: an Efficient Synchronization Scheme

Gabriel Antoniu, Luc Bougé, Sébastien Lacour

# Making a DSM Consistency Protocol Hierarchy-Aware: An Efficient Synchronization Scheme

Gabriel Antoniu, Luc Bougé, and Sébastien Lacour
PARIS Research Group, IRISA/INRIA and ENS Cachan
Campus de Beaulieu, F-35042 Rennes, France
Contact: Gabriel.Antoniu@irisa.fr

## Abstract

*We consider the design of DSM consistency protocols for* hierarchical architectures*. Such architectures typically consist of a* constellation *of loosely-interconnected* clusters*, each cluster consisting of a set of tightly-interconnected* nodes *running* multithreaded *programs. We claim that high performance can only be reached by taking into account this interconnection hierarchy at the very core of the protocol design. Previous work has focused on improving locality in* data management *by caching remote data within clusters. In contrast, our idea is to improve locality in the* synchronization management*. We demonstrate the feasibility through an experimental implementation of this idea in a home-based protocol for Release Consistency, and we provide a preliminary evaluation of the expectable performance gain.*

## 1. Introduction

Many recent high-performance computing platforms have been built by assembling together a large number of commercial off-the-shelf PCs. Such architectures are usually made of a *constellation* of loosely-connected *clusters*, each of them being made of a set of tightly-connected *nodes*. To exploit these architectures efficiently, many modern applications use *multithreaded* programming techniques to overlap communication delays with computation smoothly. Thus, the grand picture is a hierarchical interconnection structure with (at least) three levels of latency: 1) *inter-cluster* communication, through low-cost, medium-latency Local Area Networks (FastEthernet, etc.); 2) *inter-node* communication through specific low-latency System Area Networks (SCI, Myrinet, etc.); and 3) *inter-thread* communication, through direct memory-level interaction. The ratio of latency between each level typically ranges from 10 to 100.

Designing middleware for such large, hierarchical configurations is a major scientific and technical challenge, as most existing solutions have been designed within a completely different context: a small to moderate number of nodes, *e.g.*, a few dozens; a *flat*, hierarchy-unaware interconnection, where the communication latency may be considered as *uniform* across the partners. The common observation is that such approaches do not scale well in general, and that the hierarchical nature of the configuration must be taken into account at the very early design steps. An extensive re-thinking of the design is unavoidable.

Significant work has already been carried out to adapt MPI implementations on such large-scale, hierarchical architectures [11, 10, 8], but very little study has been devoted to improve the performance of Distributed Shared Memory systems in such hierarchical configurations. A number of efforts have been dedicated to the design of efficient DSM systems for (flat) clusters of SMP nodes, like Cashmere-2L [16] and HLRC-SMP [14], which exhibit a two-level hierarchy: 1) message-passing communication at the higher, inter-node level; and 2) physically shared memory at the lower, intra-node level. In contrast, we are interested here in *constellations* of clusters of nodes, should these nodes be mono- or multi-processors. Message-passing is used for constellation-level and cluster-level communications, whereas node-level communication between threads relies on virtual-memory sharing.

When used "blindly" on such architectures, traditional consistency protocols intended for flat configurations usually fail to deliver good performance. Even the widely-accepted Multiple-Writer DSM protocols designed for relaxed consistency models, are affected. Indeed, these protocols heavily rely on data transfers to and from nodes sharing a page, to inform each other about the modifications. Should a page be shared among different clusters, the performance of the whole system is limited by the high-latency inter-cluster links. An early work in this domain was for instance the Dosmos system [6]. One of the most advanced work we are aware of on this topic is the Clustered-LRC [4]

DSM system. That system extends a protocol for Lazy Release Consistency (LRC) used by TreadMarks [1], by taking into account the hierarchical interconnection in the management of replicated data. The main idea is to introduce cluster-based proxies, whose role is to cache page modifications at the level of each cluster. Successive accesses to the same page by nodes located within the same cluster can re-use the locally cached *diffs*, in contrast to the original TreadMarks DSM system.

We claim that this data management-directed approach is not the only way to obtain high performance from DSM systems on large, hierarchical architectures running multithreaded programs. Indeed, another major source of inefficiency in this context is *synchronization*, since DSM protocols for relaxed consistency models heavily rely on system-wide locks. Any access to shared data involves acquiring a lock, so that the latency of this operation is a crucial performance factor. Therefore, our purpose is to explore alternative, *hierarchy-aware* implementations for locks, where the acquisition time can take advantage of locality. Then we show how consistency protocols can benefit from this optimized implementation.

In this paper, we focus on Release Consistency protocols. We consider a *flat* protocol based on an eager, home-based approach, and we show that it is possible to make it *hierarchy-aware* while preserving the original semantics of the consistency model. The price to pay is a more elaborate management of fairness at the inter-node and inter-cluster levels. Though this paper focuses on a specific protocol, we claim that the approach is generic and can be applied to other similar protocols for relaxed consistency models.

The paper is organized as follows. In Section 2, we describe the original *flat* protocol and we show how it can be made *hierarchy-aware*, by introducing the notion of *partial lock release*. Section 3 illustrates some noteworthy implementation details related to this technique and Section 4 reports preliminary performance results obtained on the DSM-PM$^2$ generic implementation platform [2] for DSM protocols. Unfortunately, the experiments could only be run on a PC cluster far too small with respect to modern large-scale architectures. Nevertheless, we can already demonstrate significant performance improvements on synthetic benchmarks.

## 2. A Hierarchy-Aware Protocol for Release Consistency

### 2.1. Hierarchy Awareness

The performance criterion we consider is *latency*: while bandwidth increases with technology, the latency of remote requests will always remain a function of switch delays and distance, becoming the limiting factor of a network connection [9]. Another reason why we focus on latency rather than bandwidth is that we are dealing with small messages of a few bytes (invalidation messages, acknowledgements) and relatively small messages of 4 kB or 8 kB (a page). We assume that we have only *one process per node* for the sake of simplicity, and we consider a 3-level hierarchy defining two gaps in the communication performance (see Figure 1):

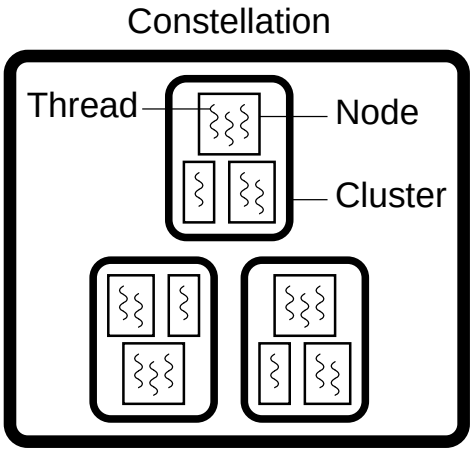


**Figure 1. Hierarchy of interconnections between threads, nodes and clusters: the thicker the line, the higher the latency.**

- several threads running on a node, sharing the same address space within a single process;
- several nodes inside the same cluster communicating through a low-latency network such as SCI [15];
- several clusters in a constellation communicating through a higher-latency network such as FastEthernet.

Our experimental platform is made of PCs (PentiumII at 450 MHz) connected over a fully-switched FastEthernet network, and equipped with relatively old SCI cards (type D310). On this platform, we observe an inter-cluster FastEthernet/TCP latency of 100 $\mu$s and an intra-cluster SCI/SISCI latency of 8 $\mu$s. Thus, the ratio of latency is around 12.

To cope with such a gap in the latencies, we try to minimize the number of messages sent over the high-latency links, and to make synchronization and consistency operations as local as possible.

### 2.2. A *flat* Home-Based Protocol

Our goal is to illustrate how a consistency protocol can benefit from hierarchy-aware synchronization. For the sake

of simplicity, we use an eager variant of the HLRC [7, 19] (Home-Based Lazy Release Consistency) protocol as a starting point.

HLRC allows multiple writers (*i.e.*, concurrent threads running on different nodes) to modify different parts of a page *simultaneously*. These modifications are made within critical sections. Each page is statically attached to a particular node (called *home* node), which is in charge of maintaining an up-to-date version of the page. When a thread has finished its write accesses to a page, it exits its critical section by releasing a lock. At that moment, the thread computes its modifications to the page (*diffs*) and sends them to the home node, which applies them immediately. The other possible copies of the modified page are invalidated by sending invalidation messages to the nodes holding the copies. Later, on a page fault following such an invalidation, the faulting nodes fetch the whole page from the home node.

In the original HLRC protocol, the replicated pages are *lazily* invalidated at the acquire operation following the critical section in which a page was modified. In contrast, we consider an eager variant of HLRC, in which invalidation messages are *eagerly* (*i.e.*, immediately) sent out to the nodes holding copies of the modified pages. This variant is simpler, since it avoids the need for timestamps for page version handling. A multithreaded version of this eager variant of HLRC, called HBRC [3] (Home-Based Release Consistency), has been previously designed and implemented on top of the DSM-PM$^2$ generic implementation platform [2] for DSM protocols. We used this protocol as a starting point and derived a new, hierarchy-aware protocol, which we implemented and evaluated using the DSM-PM$^2$ platform.

The efficiency of the HBRC protocol is limited by two key factors. First, we note that in "traditional" implementations of distributed locks, the scheduling policy used to handle lock requests does not take into account locality [13, 18]. Our experience shows that using *locality-based lock acquisition scheduling* can significantly improve the overall efficiency. The second key factor is related to the *management of the invalidation acknowledgements* following a lock release. In the HBRC protocol, upon receipt of diffs during a release operation, the home node sends out invalidation messages to *all* the nodes holding copies of the modified pages. In order to ensure memory consistency, the thread which releases the lock must not grant it to another thread before receiving *all* invalidation acknowledgements. This operation is system-wide, since acknowledgements may come from local nodes of the same cluster and from remote nodes of distant clusters. Thus, the delay for such an operation is limited by the latency of the inter-cluster communications.

The main contribution of this paper is to show how a hierarchy-aware approach can be used to tackle these two limitations.

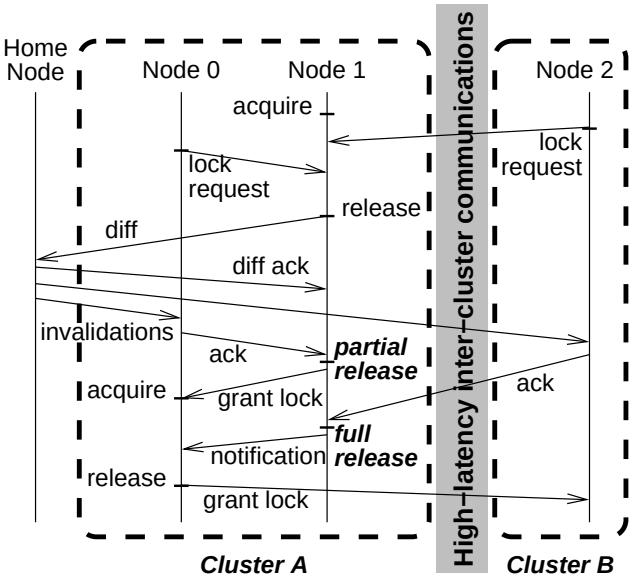


**Figure 2. Reordering of lock acquisitions and partial release.**

## 2.3. Hierarchy-Aware Synchronization

Let us consider Node 1 in Cluster A which has acquired a lock and is currently executing in critical section (see Figure 2). Now, Node 2 in Cluster B and then Node 0 in Cluster A want to acquire the same lock. A hierarchy-unaware implementation of the distributed mutual exclusion may grant the lock to the first requester (remote Node 2), so we may incur two high-latency communications to exchange the lock between both clusters. In our hierarchy-aware implementation of distributed mutual exclusion, the lock will be granted first to Node 0 in the *local cluster* A, even if the request from Cluster B arrived first. Thus, we incur one low-latency communication for the lock to travel within Cluster A plus one high-latency communication for the lock to go from Cluster A to Cluster B. We re-order the lock acquisitions at the cluster level to trade a high-latency message for a low-latency one.

We reproduce the same priority mechanism between the threads of a node as between the nodes of a cluster. Thus, we re-order the lock acquisitions at the node level to minimize the number of communications between nodes.

## 2.4. Partially Releasing Locks Within Clusters

In the flat version of HBRC, while releasing a lock, Node 1 must wait for *all* the page invalidation acknowl-

edgements prior to granting the lock to another node. Yet, it is likely that the acknowledgements coming from the *local cluster* will arrive *before* those coming from *remote clusters* through a higher-latency network. So, in our hierarchical protocol, Node 1 will grant the lock to Node 0 (see Figure 2) *in the local cluster* A after receiving the acknowledgements from the local cluster, without waiting for those coming from remote clusters: we call that *Partial Release*.

Then, when Node 0 enters the critical section, its memory accesses are ensured to be consistent because the pages modified during the previous critical section have been invalidated. We are certain that the DSM pages on Node 0 have actually been invalidated, since Node 1 received the acknowledgements from all the nodes in its local cluster before granting the lock.

Thus, the lock can travel from node to node *within* the same cluster several times, without requiring receipt of the invalidation acknowledgements from remote clusters, without wasting time waiting for remote acknowledgements. As and when invalidation acknowledgements are received from remote clusters, the lock gets *fully released* on the nodes which successively acquired it in Cluster A, and a notification is propagated along that chain of nodes. The lock will not leave Cluster A until after being fully released by all the nodes of the chain.

### 2.5. Avoiding Useless Modification Propagations

In the flat version of HBRC, while releasing a lock, a thread *systematically* sends to the home nodes the modifications made on the DSM pages in the latest critical section. In contrast, in our hierarchical protocol, when a thread releases a lock, it does *not* send the modifications if another thread *on the same node* is granted the lock immediately.

That scheme is correct from the perspective of memory consistency because two threads on a node share the same address space. The modifications made by any number of threads which acquired the same lock successively on the same node will be sent *all at once* when the lock is granted to another node.

In Section 2.3, we saw that a thread on a node prefers granting a lock to another thread *on the same node* rather than to another node. That priority mechanism makes it more frequent for a thread to grant a lock to another thread on the same node. Therefore, it exhibits more situations where it is useless to send the modifications to the home nodes.

### 2.6. Handling the Lack of Fairness

As we saw in Section 2.3, our hierarchy-aware implementation of the distributed mutual exclusion prefers granting a lock to local threads and to local nodes: when two threads are in competition with each other to acquire a lock, the thread currently holding the lock will prefer granting it to another thread on the *same node* rather than to *another node*; similarly, when two nodes are in competition with each other to acquire a lock, the node currently holding the lock will prefer granting it to a node in the *same cluster* rather than to a node in a *remote cluster*.

That re-ordering of lock acquisitions may lead to unfairness. That lack of fairness may result in situations of starvation. For instance, a node in Cluster B may want to acquire a lock while some nodes in Cluster A also want to acquire the same lock indefinitely: if the lock is initially in Cluster A, then it will never travel to Cluster B because of the mechanism of priority.

To overcome that shortcoming, bounds have been set and can be tuned to limit the number of consecutive acquisitions of a lock by threads on the same node while other nodes have requested the lock; similarly, bounds have also been set and can be tuned to limit the number of consecutive acquisitions of a lock inside the same cluster while nodes in other clusters have requested the lock. Note that if those bounds are set to 1, then we have a flat implementation of distributed mutual exclusion, with no priority granted to local threads or local nodes; when the bounds are set to infinity, then fairness is not enforced any longer.

Thus, we trade fairness for performance in terms of execution time, without sacrificing correctness or leading to deadlocks.

## 3. Noteworthy Implementation Details

Conceptually, a list of requesting threads is attached to each lock. That list is *ordered* in function of the order in which the lock requests were received. Two counters are also attached to the lock: `thread_privilege` and `node_privilege`, initially set to zero.

When a thread wants to release a lock to exit a critical section, it searches the list of requesting threads for another thread *on the same node*. If such a requester is found, then the modifications made on the DSM pages are *not* sent to the home nodes (Section 2.5), and the lock is *immediately* granted to the local thread. If the new owner of the lock was *not the first* requester in the list, then the counter `thread_privilege` is incremented by 1, meaning that this thread was selected to the detriment of the first requester of the list. When the counter `thread_privilege` reaches a limit `max_tp` (which can be tuned by the user application for more fairness), the priority to the local thread is *not* applied: the lock will be granted to a thread on another node if any, and the counter `thread_privilege` will be reset to zero.

In case the list does not contain a requesting thread on the same node, or if the counter `thread_privilege`

reached its limit, then the modifications made on the DSM pages are sent to the home nodes. After receiving the invalidation acknowledgements from the nodes in the *local cluster* (Section 2.4), the thread exiting the critical section searches the list of requesters attached to the lock for a thread *in the local cluster*. If such a requesting thread is found, then it is granted the lock. If the new owner of the lock was *not the first* requester in the list, then the counter `node_privilege` is incremented by 1, meaning that this thread was selected to the detriment of the first requester of the list. When the counter `node_privilege` reaches a limit `max_np` (which can be tuned by the user application for more fairness), the priority to the local node is *not* applied: the lock will be granted to a thread in another cluster, and the counter `node_privilege` will be reset to zero.

# 4. Preliminary Performance Evaluation

This section presents the experiments we carried out to quantify the performance gains due to hierarchy-aware synchronization and to the hierarchical consistency protocol. Those experiments were run on the platform described in Section 2.1: the PCs we used were PentiumII's at 450 MHz under Linux 2.2.18, connected over a fully-switched FastEthernet network (for the inter-cluster links) and equipped with relatively old SCI cards (type D310) for the intra-cluster links. On this particular platform, we observed an inter-cluster FastEthernet/TCP latency of 100 $\mu$s and an intra-cluster SCI/SISCI latency of 8 $\mu$s. Thus, the ratio of latency was around 12.

Our experiments have been conducted using the DSM-PM$^2$ experimental implementation platform for multithreaded DSM consistency protocols. This user-level platform provides basic building blocks, allowing for an easy design, implementation and evaluation of a large variety of multithreaded consistency protocols within a unified framework. It relies on the PM$^2$ (*Parallel Multithreaded Machine*, [12]), a runtime system for distributed, multithreaded applications. PM$^2$ provides a POSIX-like programming interface for thread creation, manipulation and synchronization in user space, on cluster architectures. PM$^2$ is available on most UNIX-like operating systems, including Linux and Solaris. For network portability, PM$^2$ uses a communication library called Madeleine [5], which has been ported on top of a large number of communication interfaces: high-performance interfaces, like SISCI/SCI and VIA, but also more traditional interfaces, like TCP and MPI. DSM-PM$^2$ inherits this portability, since all its communication routines rely on Madeleine. Communication operations provided by Madeleine are guaranteed to be reliable (no message loss). We used the version 3 of Madeleine, which copes with hierarchical, network-heterogeneous clusters. This feature allowed us to perform our experimental evaluation without any network-specific code at the protocol level.

## 4.1. Hierarchy-Aware Synchronization

The table below demonstrates the performance gain obtained from our hierarchy-aware implementation of distributed mutual exclusion using a synthetic application. That program runs on four nodes connected over our SCI network in a single cluster: each node runs four threads. Each thread executes an empty critical section 10,000 times using a unique lock: executing an empty critical section consists in acquiring the lock and releasing it immediately. Varying the parameter `max_tp` which limits the number of consecutive acquisitions of a lock by the threads of a node (Section 3), we quantify the effect of the priority given to local threads for granting locks.

| `max_tp` | 1 | 5 | 15 | 25 | $\infty$ |
|---|---|---|---|---|---|
| Time ratio | 1 | 3.4 | 5.8 | 6.9 | $\simeq$ 60 (unfair) |

We measured the time $T_{\text{flat}}$ it takes to execute our synthetic application using a flat version of our implementation of mutual exclusion, with no priority given to local threads. Then we measured the time $T_{\text{hierarchy}}$ it takes to execute the same program using our hierarchy-aware implementation of mutual exclusion for different values of `max_tp`. The time ratios $\frac{T_{\text{flat}}}{T_{\text{hierarchy}}}$ given in the table show the speedups due to hierarchy-awareness in granting locks. As expected, the greater `max_tp`, the greater the speedup, because more threads can acquire the lock in a row on the same node, so less messages are exchanged between the nodes to transmit the lock.

## 4.2. Partially Releasing Locks Within Clusters

Figure 3 reports the performance gain obtained from the partial release mechanism using a synthetic application. The program runs on a varying number of clusters: each cluster has two nodes connected over SCI, and the clusters are interconnected through a fully-switched FastEthernet network as described in Section 2.1. There is just one thread per node executing 10,000 critical sections consisting in acquiring a unique lock, incrementing a unique shared integer and releasing the lock.

We measured the time it takes to execute a single critical section in our synthetic application without the partial release, *i.e.*, waiting for *all* the invalidation acknowledgements before releasing a lock. Then we measured the time it takes to execute a single critical section in the same program using our hierarchical protocol with the partial release. Both measurements were performed without any limit on the number of consecutive acquisitions of a lock inside a cluster (`max_np` $= \infty$). Figure 3 shows the timings for

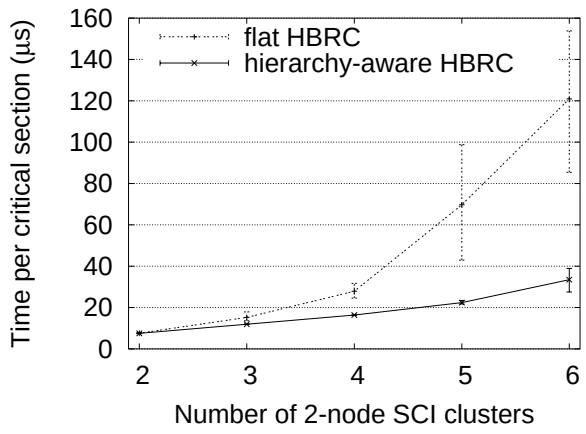


**Figure 3. Impact of the mechanism of partial release.**

different numbers of 2-node clusters. The hierarchical protocol performs 3 times as fast as the flat protocol with 5 clusters, and 4 times as fast with 6 clusters: the more we have clusters, the more we have high-latency links and the more our hierarchical protocol can take advantage of the partial release. Indeed, as the number of clusters grows, the flat protocol will be more likely to wait for remote acknowledgements, while our hierarchical protocol does not need to wait for the remote acknowledgements.

### 4.3. Avoiding Useless Modification Propagations

The table below demonstrates the performance gain obtained from not sending modifications to the home node when a thread exiting a critical section grants the lock immediately to another thread on the same node. We use the same configuration as in Section 4.1: four nodes connected over SCI in a single cluster, each of which running four threads. Each thread executes 10,000 critical sections consisting in acquiring a unique lock, incrementing a unique shared integer and releasing the lock.

| `max_tp` | 1 | 5 | 15 | 25 | $\infty$ |
|---|---|---|---|---|---|
| Time ratio | 1 | 2.1 | 4.7 | 7.3 | unfair |

We measured the time $T_{\text{flat}}$ it takes to execute our synthetic application, with a flat protocol which *systematically* sends the modifications to the home node at the release operation. Then we measured the time $T_{\text{hierarchy}}$ it takes to execute the same program using our hierarchical protocol, *i.e.*, *without* sending page modifications when a thread exiting a critical section grants the lock immediately to another thread on the same node. The table gives the time ratios $\frac{T_{\text{flat}}}{T_{\text{hierarchy}}}$ for different values of `max_tp`: the higher the priority to local threads, the greater performance gain our protocol achieves by not sending the modifications systematically, but the less fairness is enforced. Once again, our hierarchical protocol exchanges less messages between the nodes and waits less often for invalidation acknowledgements.

## 5. Conclusion

Our objective is to obtain high performance from DSM systems on large, hierarchical architectures, typically constellations of loosely-connected clusters of tightly-connected nodes running multithreaded programs. We claim that this goal cannot be reached without considering this hierarchical architecture at the very core of the design. Some work has already been carried out regarding the management of data, more precisely caching the diffs at cluster-level in a Lazy Release Consistency protocol [4]. In this paper, we have explored an alternative, complementary approach: managing synchronization in a hierarchy-aware manner, by taking into account the lower communication latency between partners located closer within the hierarchy. Our contribution is twofold. First, we propose a *hierarchy-aware approach* to distributed synchronization. Second, we introduce the concept of *Partial Release* for locks, which allows consistency protocols to efficiently exploit the earlier delivery of acknowledgements issued by closer partners.

We believe that this concept of *partial release* is general enough to be applied to other synchronization objects than locks, such as *semaphores* or *monitors*. However, it cannot be used in conjunction with *barriers*: by definition, no thread can exit a barrier before *all* the participating threads all over the system have synchronized with each other. We also suggest that the *partial release* concept could be applied to any *Eager* Release Consistency protocol, where the thread exiting a critical section must wait for some kind of acknowledgements before actually releasing the lock.

Our preliminary experiments with micro-benchmarks demonstrate significant improvements in terms of performance. Unfortunately, the experiments could only be run on a platform far too small with respect to modern, large-scale architectures. Also, we still need to carry out further tests using realistic applications such as *Splash-2* programs [17]. We anticipate that *lock-intensive* applications will especially take advantage of our specific partial release implementation. Among the *Splash-2* applications, `Ocean` and `Cholesky` look particularly favorable.

We anticipate that the greater the ratio between inter- and intra-cluster latencies, the better performance gain our solution will yield. For instance, with recent hardware, the ratio of latency between Local-Area and System-Area communications can exceed 20; between Wide-Area and Local-Area communications, the ratio of latency can be as high as 500.

Our work focused on *synchronization locality*, as op-

posed to the work done by Arantes *et al.* [4], which concentrates on *data locality*. It would be interesting to consider merging these two approaches to add up their respective performance improvements.

## References


[1] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, Feb. 1996.

[2] G. Antoniu and L. Bougé. DSM-PM2: A portable implementation platform for multithreaded DSM consistency protocols. In *Proceedings of the 6th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'01)*, volume 2026 of *LNCS*, pages 55–70, San Francisco, CA, Apr. 2001. Springer-Verlag.

[3] G. Antoniu and L. Bougé. Implementing multithreaded protocols for release consistency on top of the generic DSM-PM2 platform. In *Proceedings of the International Workshop on Cluster Computing (IWCC'01)*, volume 2326 of *LNCS*, pages 182–191, Mangalia, Romania, Aug. 2001.

[4] L. B. Arantes, P. Sens, and B. Folliot. An effective logical cache for a clustered LRC-based DSM system. *Cluster Computing Journal*, 5(1):19–31, Jan. 2002.

[5] O. Aumage, L. Bougé, J.-F. Méhaut, and R. Namyst. Madeleine II: A portable and efficient communication library for high-performance cluster computing. *Parallel Computing*, 28(4):607–626, 2002.

[6] L. Brunie and L. Lefèvre. New propositions to improve the efficiency and scalability of DSM systems. In *1996 IEEE 2nd International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP'96)*, pages 356–364, Singapore, June 1996. IEEE.

[7] L. Iftode. *Home-based Shared Virtual Memory*. PhD thesis, Princeton University, NJ, June 1998.

[8] N. T. Karonis, B. R. de Supinski, I. Foster, W. Gropp, E. Lusk, and J. Bresnahan. Exploiting hierarchy in parallel computer networks to optimize collective operation performance. In *Proceedings of the 14th International Parallel and Distributed Processing Symposium (IPDPS)*, pages 377–384, Cancun, Mexico, May 2000.

[9] P. Keleher. Consistency maintenance in large-scale systems. Department of Computer Science, University of Maryland, May 1997.

[10] T. Kielmann, R. F. H. Hofman, H. E. Bal, A. Plaat, and R. A. F. Bhoedjang. MagPIe: MPI's collective communication operations for clustered wide area systems. In *7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'99)*, pages 131–140, Atlanta, GA, May 1999. ACM Press, New York, NY.

[11] T. Kielmann, R. F. H. Hofman, H. E. Bal, A. Plaat, and R. A. F. Bhoedjang. MPI's reduction operations in clustered wide area systems. In *Proceedings of the Message-Passing Interface Developer's and User's Conference (MPIDC'99)*, pages 43–52, Atlanta, GA, Mar. 1999.

[12] R. Namyst. *PM2: an environment for a portable design and an efficient execution of irregular parallel applications*. PhD thesis, LIFL, Université Lille 1, France, Jan. 1997. In French.

[13] M. Raynal. *Algorithms for Mutual Exclusion*. MIT Press, Cambridge, MA, Aug. 1986.

[14] R. Samanta, A. Bilas, L. Iftode, and J. P. Singh. Home-based SVM protocols for SMP clusters: Design and performance. In *Proceedings of the 4th IEEE International Symposium on High-Performance Computer Architecture (HPCA-4)*, Las Vegas, NV, Feb. 1998.

[15] IEEE Standard for Scalable Coherent Interface (SCI), Aug. 1993. IEEE Standard 1596.

[16] R. Stets, S. Dwarkadas, N. Hardavellas, G. C. Hunt, L. I. Kontothanassis, S. Parthasarathy, and M. L. Scott. Cashmere-2L: Software coherent shared memory on a clustered remote-write network. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, pages 170–183, Saint-Malo, France, Oct. 1997.

[17] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36, Santa Margherita Ligure, Italy, June 1995.

[18] M.-Y. Wu and W. Shu. An efficient distributed token-based mutual exclusion algorithm with central coordinator. *Journal of Parallel and Distributed Computing (JPDC)*, 62(10):1602–1613, Oct. 2002.

[19] Y. Zhou, L. Iftode, and K. Li. Performance evaluation of two home-based lazy release consistency protocols for shared memory virtual memory systems. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI'96)*, pages 75–88, Seattle, WA, Oct. 1996.