# HOME-BASED SHARED VIRTUAL MEMORY

Liviu Iftode

A DISSERTATION
PRESENTED TO THE FACULTY
OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE
BY THE DEPARTMENT OF
COMPUTER SCIENCE

June 1998

**Părinţilor mei,**
**Lucreţia şi Vasile Iftode**

# Abstract

In this dissertation, I investigate how to improve the performance of shared virtual memory (SVM) by examining consistency models, protocols, hardware support and applications. The main conclusion of this research is that the performance of shared virtual memory can be significantly improved when performance-enhancing techniques from all these areas are combined.

This dissertation proposes home-based lazy release consistency as a simple, effective, and scalable way to build shared virtual memory systems. In home-based protocols each shared page has a home to which all writes are propagated and from which all copies are derived. Two home-based protocols are described, implemented and evaluated on two hardware and software platforms: Automatic Update Release Consistency (AURC), which requires hardware support for fine-grained remote writes (automatic updates), and Home-based Lazy Release Consistency (HLRC), which is implemented exclusively in software.

The dissertation investigates the impact of memory-mapped network technology, in particular that of automatic update support, on home-based protocol design and performance. The performance evaluation of HLRC and AURC prototypes implemented on SHRIMP, the PC-based multicomputer built at Princeton, indicates that memory-mapped communication can significantly improve the efficiency of home-based protocols, and that the automatic update support is an effective SVM accelerator.

The dissertation proposes a new memory consistency model called Scope Consistency (ScC). ScC offers most of the potential performance advantages of entry consistency without requiring explicit association between data and synchronization. A simulation-based comparative evaluation of ScC and AURC protocols shows that ScC reduces false sharing for irregular applications with lock-based synchronization.

Finally, this dissertation proposes a classification of parallel applications which can serve as a framework for understanding the performance of shared virtual memory protocols. This

classification derives from an analysis of the sharing patterns induced by the interaction between the sharing patterns inherent in the algorithm, and the coherence granularity imposed by the protocol implementation. The dissertation also includes an extensive simulation-based performance evaluation of the traditional LRC and AURC protocols on the Splash-2 application suite which uses the proposed classification framework for performance understanding.

# Acknowledgements

This dissertation fulfills the dream of a child who, thirty years ago, began his journey through education in a little medieval village in Romania, called Cristian. This learning adventure from Cristian to Princeton would not have been possible without the infinite love and patience of my parents. The silent sacrifice of their life made mine shine. I thank God for giving me the chance to show that their struggle was not in vain.

I would like to thank my wife, Cristina, for her devotion and tolerance all these many years when I was never home. Thanks to all my very good friends, near and far, who believed in me and colored my life infinitely more than any computer could ever have done. I thank Gabriela Hristescu for her generous friendship and support which I deeply treasure.

I would like to thank my advisors Kai Li and Jaswinder Pal Singh and the other members of my committee, Douglas W. Clark, Edward W. Felten and Andrew W. Appel, for helping me in so many ways to successfully accomplish my dissertation research at Princeton. I was very fortunate to have Kai Li as my advisor. Who else would have tolerated a graduate student so different from his own model, with so little sense of time and deadlines? I cannot express how deep my gratitude is for his thoughtful guidance and friendship. I wish to thank JP who came to Princeton at exactly the right time to broaden my research. He taught me the Stanford research style, which I hated sometimes, but from which I have learned and benefited greatly. I thank Doug, my third reader, who I met too late to become my third advisor. His wisdom gave me the serenity to look beyond the present. I am grateful to Ed Felten for the helpful discussions we had when I began to work on my first SVM protocol and to Margaret Martonosi for her many thoughtful suggestions.

Thanks to all my distinguished Professors at Princeton whose close presence gave me a sense of modesty with which I will live for the rest of my life. I would also like to thank my Romanian Professors from Gheorghe Lazar High-School in Sibiu, and from the Polytechnic Institute of Bucharest for showing me the sky.

I have been privileged to meet so many wonderful colleagues at Princeton with whom I shared so many days and, especially, so many nights, trying to build a new version of what Mother Nature did so perfect the first time: the SHRIMP. Yuanyuan Zhou implemented the first all-software home-based protocol on the Intel Paragon and contributed with many ideas to the design of various other protocols afterwards. Matthias A. Blumrich built and, more importantly, made SHRIMP finally work. Our disputes on whether it is a hardware or a software bug only increased my respect for what he has accomplished with SHRIMP. Cezary Dubnicki contributed with many ideas to the first design of the AURC protocol, implemented the basic communication library for SHRIMP, and endured the blame for almost every software bug, including those which were in my protocol. I would like to thank Stefanos N. Damianakis, Yuqun Chen, Dongming Jiang, David L. Oppenheimer, Richard. D. Alpert, Angelos Bilas, Masato Edahiro, Mariusz Jakubowski, Sanjeev Kumar, Cheng Liao and Rudrajit Samanta for their endless help and friendship– something I took advantage of too often. I also had the invaluable chance to have my Ph.D. road intersect with those of Marios Dikaiakos, Pei Cao, Karin Petersen, James Plank who became my graduate student models. Thanks to Melissa M. Lawson, the graduate program coordinator, who put my academic life back in order whenever I messed it up. Thanks to Patricia Zabielski and Virginia Hogan for helping me start the mornings with a smile and to Sandra Barbu with whom I exercised all Romanian greetings.

My Ph.D. achievements at Princeton would not have been possible without the encouragement and support I felt all these years from a handful of special people who I deeply respect and to whom I am particularly grateful: Willy Zwaenepoel, Marc Shapiro, James Philbin, Anne Rogers, Esther Dyson, Laura Tong-Sardina and last, but not least, to Bill Joy.

Beyond a prestigious University with vertical towers and people, Princeton means to me a quiet place on Earth with charming streets and an extremely cultivated taste for living. I thank Ruth and Hans Fiuczynski, and Cornelia and Lou Sheedy for revealing me the flavor of their town on those rare occasions when I could escape from my office.

Once upon a time, I was a graduate student in Princeton. Thank you all for sharing with me those wonderful years.

This work has been sponsored in part by DARPA under grant N00014-95-1-1144, by NSF under grant MIP-9420653, and by the Intel Corporation.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Thesis

This dissertation proposes new protocols and a new consistency model for shared virtual memory which are simpler, more efficient and more scalable than conventional approaches. It also introduces a classification of shared-memory parallel applications as a framework for understanding their performance under shared virtual memory systems.

## 1.2 Shared Virtual Memory

Ever since the modern computer was invented, there has always been a demand for more computation power. Putting multiple computers to work together, or *in parallel*, is apparently a straightforward way to increase the performance of computation. However, more than thirty years of parallel computing have shown that crossing the bridge from sequential to parallel computing has been far from simple because both the architecture and programming are substantially different.

The two traditional models of parallel computing, *message-passing* and *shared memory*, expose contradictory characteristics. On one hand, message passing requires simple architectural support with a good price to performance ratio but makes parallel programming complicated because the communication is exposed to the programmer. On the other hand, by hiding the communication behind memory accesses, shared memory makes parallel programming as simple as an extension of the uniprocessor concurrent programming model. However, scalable hardware for shared memory is both complicated and expensive to build.

*Shared virtual memory* (SVM) is an attempt to combine the simplicity of shared memory programming with the inexpensiveness of message passing implementation. This idea was first proposed in 1986 by Kai Li and consists of emulating a cache-coherent multiprocessor by using the virtual memory mechanism [LH86, Li86, LS89]. The local memory of each node serves as a cache of a global virtual address space as shown in Figure 1.

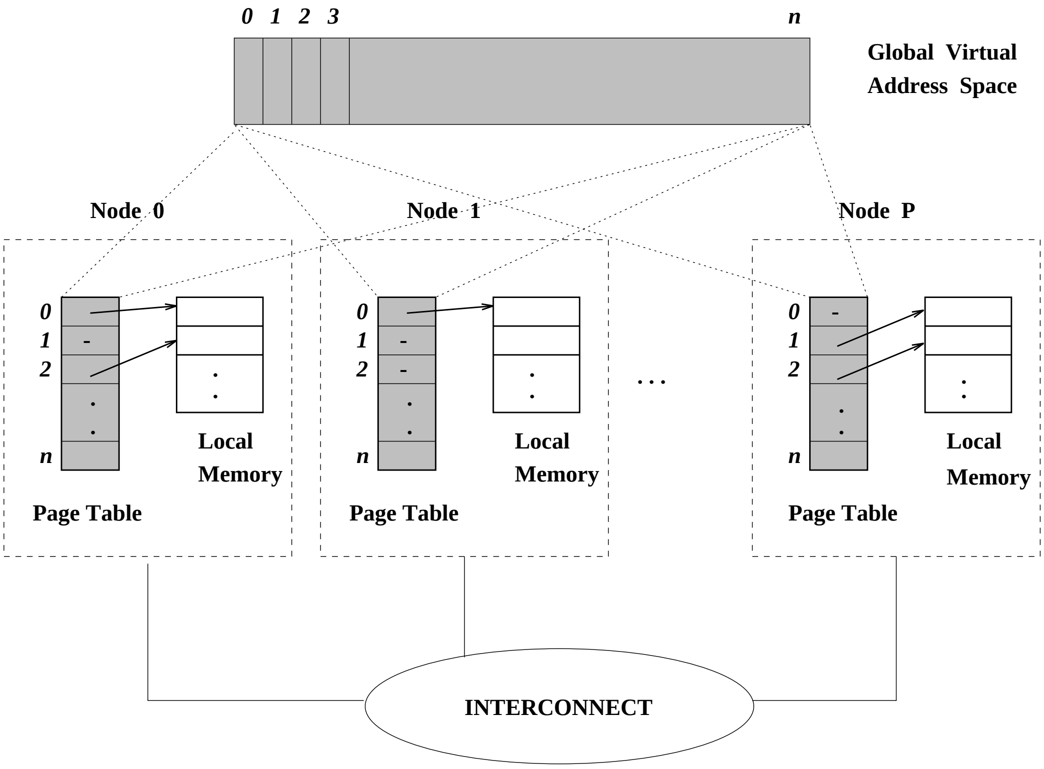



Figure 1: Shared Virtual Memory

The coherence of the virtual shared address space is maintained through the local page tables using an invalidation-based protocol: valid page table entries correspond to shared pages that are currently cached in the local memory; invalid entries correspond to pages that are not locally available, either because they were never cached at that node or because the local copy was invalidated. When a node attempts to access an invalid page, a page fault occurs. In the page fault handler the SVM protocol fetches an up-to-date copy of the page into local memory from a remote node and establishes the mapping.

There are several factors that limit the performance of shared virtual memory. First, the large coherence granularity imposed by the underlying virtual memory page size induces false sharing and fragmentation for most applications. *False sharing* occurs when multiple processors access unrelated variables co-located on the same page and at least one access is a write access. *Fragmentation* occurs when the entire page is fetched, instead of just the faulting word, at the access fault time. These effects artificially increase the traffic, therefore causing a higher communication to computation ratio under SVM than under a hardware cache-coherent shared memory for the same application. Second, the unit cost of communication of a shared virtual memory implementation is higher than that of a hardware implementation since it is performed through message passing. Third, synchronization primitives are slow because they are also implemented through explicit messages. Moreover, especially in the presence of contention, page faults occurring inside critical sections dilate the synchronization waiting time. Finally, the protocol overhead is large in SVM because it is performed in software and therefore the main computation must be interrupted in order to service remote protocol requests.

The obvious measure of the success of shared virtual memory is the closeness of its performance to the more expensive hardware shared memory approach. This is why the first decade of SVM witnessed an extremely active and fruitful research effort driven mainly by two performance improvement avenues: relaxed consistency models which aim at reducing the communication traffic and additional hardware support provided in the communication architecture which can reduce the cost of communication. Since the first solution increases the programming complexity, while the second one increases the cost of the system, the research challenge was to determine how far to go in pushing for better performance without compromising the advantages of the software approach.

## 1.3 SVM: The First Decade

This section is a survey of the first 12 years of research in SVM, placing the multi-track flow of ideas and results obtained so far in a comprehensive framework. The main contributions indicated in Figure 2 are classified in four categories: consistency models, protocols, architecture and applications.

The first SVM implementation [Li86] used a *sequential consistency* (SC) [Lam79] model, which meant that coherence operations had to be propagated immediately and processes had to wait for memory operations to complete before moving on to new ones. Progress was slow until the *release consistency* (RC) model [GLL$^+$90] breathed new life into the software

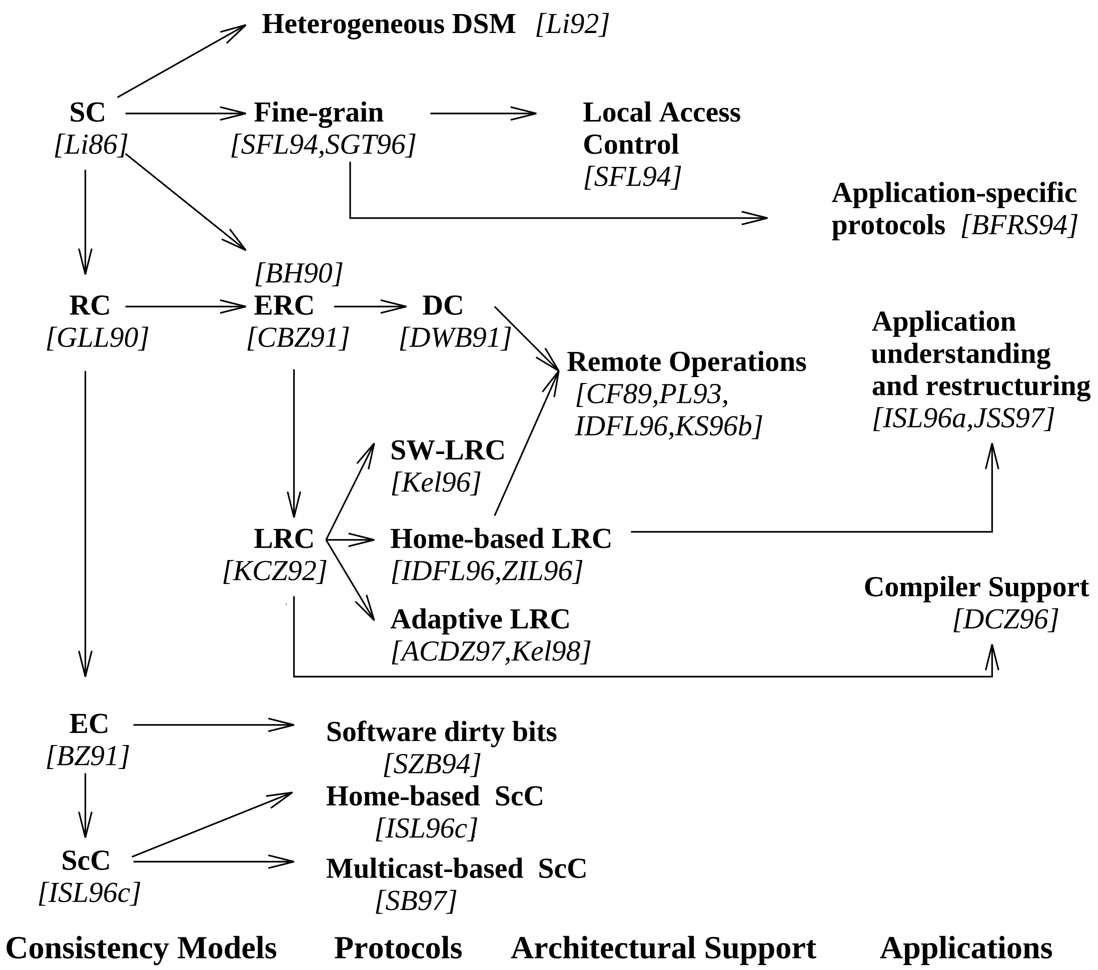



Figure 2: Research in Shared Virtual Memory

approach in the early 1990s and lead to *eager release consistency* (ERC) implementation in Munin [CBZ91] and *lazy release consistency* (LRC) in TreadMarks [KCZ92]. However, there is an early relaxed consistency software protocol [BH90] that anticipated these RC-based protocols. *Entry consistency* (EC) [BZ91] and *scope consistency* (ScC) [ISL96c] are two similar consistency models which relax the consistency requirements beyond release consistency by limiting the data to which they apply.

In the 1990s, software shared memory became a very active research area, with many groups designing new protocols, consistency models and systems [BH90, CBZ91, KCZ92, ZSLW92, BZ91, PL93, PL94, SFL$^+$94, ZIL96, ISL96c, ACDZ97, SGT96, KHS$^+$97, SB97]. These research groups inspired and motivated one another, building on each other's ideas to push performance higher. During this time, the largest number of SVM publications occurred in the protocol design. One can say that the history of SVM research in the first decade has largely been a history of coherence protocol *laziness*. Protocol laziness means delaying the propagation and application of coherence operations and data modifications as permitted by the memory consistency model. This research led to the distinction between *single-writer* (SW) and *multiple writer* (MW) protocols [Kel98], as well as between *home-based* (H) [IDFL96, ISL96c, ZIL96] and no-home or distributed (no-H) [KCZ92, SB97] write collection in LRC and ScC protocols.

With a relative maturity of protocols, SVM research has moved in the last couple of years to new areas such as the understanding of architectural support and how to use it to accelerate the protocols and the interaction among applications, compiler and protocols. Two particular hardware extensions played major roles in pushing SVM research forward: fine-grain local access control [SFL+94] and fine-grain remote access [CF89, PL93, PL94, IDFL96, IBD+98, KS96b, KHS+97]. Fine-grain sequential-consistent shared memory, initially proposed in [SFL+94], was recently revived [SGT96, SG97] as a competitive alternative to the page-based shared virtual memory approach.

This dissertation contributes to all four areas of SVM research indicated in Figure 2: a new consistency model called scope consistency [ISL96c], home-based LRC protocols [IDFL96, ZIL96], *automatic-update* architectural support for SVM [IDFL96, BAB+98, IBD+98] and application classification with respect to their behavior under SVM [ISL96a]. In the following sections I review the most important research results in these areas, summarize comparative performance results from literature and identify the lessons learned so far and the key outstanding questions.

## 1.4 Relaxed Consistency Models

A memory consistency model defines constraints on the order in which memory accesses can be performed in shared memory systems. It influences both programming complexity and performance. A strict consistency model like sequential consistency (SC) [Lam79] is intuitive for a programmer. However, the large granularity of coherence (a virtual memory page) in shared virtual memory systems can perform coherence operations very frequently under this model due to false sharing and fragmentation. In the worst case, under sequential consistency, every write to a given page may result in an invalidation and every read immediately following a write may result in a page miss. This causes much unnecessary communication and protocol overhead.

### 1.4.1 Release Consistency

Relaxed memory consistency models allow the propagation and application of coherence operations (e.g. invalidations) to be postponed to synchronization points, greatly reducing the impact of false sharing and the frequency of coherence operations. The intuition behind these models is that a user cannot distinguish the behavior of a system running under such relaxation from one that uses a sequential consistent model, as long as programs are "well written", meaning that they are data race free and use explicit synchronization. The *weak consistency* model proposed by Dubois *et al.* was the first relaxed consistency model based on the above idea [DSB86]. For this approach to work, the programmer has to distinguish between *ordinary* shared variables and *synchronization* variables.

Among the many relaxed consistency models proposed for hardware-coherent systems, release consistency [GLL+90], which separates synchronizations into *acquires* and *releases*, was the first to inspire a major breakthrough in the performance of software shared memory systems [CBZ91] (see Table 1).

| Benchmarks | SC | RC |
|---|---|---|
| MULT | 14.5 | 14.6 |
| DIFF | 8.4 | 12.3 |
| TSP-C | 11.3 | 12.6 |
| TSP-F | 4.7 | 6 |
| QSORT | 4.1 | 8.9 |
| FFT | 0.1 | 8.2 |
| GAUSS | 5.1 | 8.6 |

Table 1: Speedups for SC and RC(Munin), 16 processors [CBZ95].

### 1.4.2 Beyond Release Consistency

The consistency model game is much more difficult to play beyond protocols based on release consistency; protocols and coherence state become more complicated and even software solutions may require additional hardware or compiler support. Worse still, programming complexity also increases. However, there is some potential for performance improvement.

*Entry Consistency* [BZ91] proposed the idea of binding data to synchronization variables, and making only the data bound to that variable consistent at a synchronization event. That is, the consistency model governs not only *when* data are made coherent but also *which* data should be made coherent at a synchronization point. In entry consistency the programmer has to provide the binding, which is a major burden.

*Scope consistency* (ScC) [ISL96c], introduced in this dissertation, shows that the effect of synchronization on data binding can be achieved without including the binding requirement in the definition. Instead, synchronization variables define scopes and the association of data with scope is achieved dynamically when a write access occurs inside a scope. This reduces the programming complexity compared to EC, although there are programs where more effort is needed beyond programming for LRC. By still being page-grained, unlike EC, ScC is able to reduce false sharing while still preserving the prefetching effect of pages and keeping the required state overhead low.

Recently, ScC has been implemented in the Brazos prototype [SB97] and shown to deliver some performance improvements on irregular applications similar to earlier simulation results [ISL96c] (see Table 2). While the programming burden keeps these models outside the mainstream, EC may fit well into object-oriented languages like Java which inherently provide support for object-to-lock association and ScC's small programming burden may be alleviated by tools checking if a properly labeled program for RC also satisfies ScC.

| Benchmarks | LRC | ScC |
|---|---|---|
| SOR | 5.5 | 5.61 |
| ILINK | 2.3 | 3.4 |
| Barnes | 2.2 | 4 |
| Raytrace | 0.5 | 1.8 |
| Water | 3.8 | 3.9 |

Table 2: Speedups for LRC and ScC Protocols, 8 processors [SB97]

## 1.5 Protocol Laziness

A memory consistency model specifies when coherence operations and data *must* become visible. However, it can actually be implemented with various degrees of laziness in the propagation and application of both coherence operations and data modifications. Greater laziness typically implies greater difficulties for programmers and protocol state, but also less frequent communication and protocol operations. For example, hardware-coherent systems that use relaxed consistency tend to propagate coherence operations immediately upon data modification or writes, thus simplifying the data structures that need to be maintained for coherence. On the other hand, software systems based on relaxed consistency can afford to manage more complex protocol state and data structures and therefore to achieve better performance they often delay propagation/application until the point permitted by the consistency model.

During the past six years many SVM protocols based originally on release consistency have been developed along different but related paths, with different degrees of laziness and performance tradeoffs. Most published papers use the previous protocol along their path as the base case for comparison. However, the relevance of the performance evaluation has often been limited by two factors. First, they have typically not used a comprehensive enough set of real applications that exercise different relevant characteristics. Second, they have made assumptions about important lower-level implementation mechanisms, as well as about architectural support and performance characteristics, which can make a great difference to the higher-level tradeoff being investigated.

This section divides the evolution of protocols for maintaining consistency into three parts: laziness in propagating the coherence operations (e.g. invalidations) caused by data modifications to other processors, laziness in applying the coherence operations at those other processors and laziness in propagating data modifications.

### 1.5.1 Laziness in Propagating Coherence Information

The propagation method closest to the one used by hardware release-consistent systems is eager-propagation release consistency: invalidations are propagated to all sharing processors before the issuing processor completes its release operation. In eager release consistency (ERC) the invalidations corresponding to the pages updated before the release are sent to all the nodes holding copies of these pages (copyset) at release time. In lazy release consistency (LRC) [KCZ92], the invalidation corresponding to a write by processor $A$ is conveyed to a processor $B$ on demand upon $B$'s next acquire synchronization operation, if that acquire is causally "after" the release by the writing processor $A$. Munin [CBZ91] and TreadMarks [KCDZ94] from Rice University were the first SVM systems to propose and implement eager and lazy software release consistency protocols, respectively.

Laziness has implications in the amount and lifetime of the buffered coherence state that must be maintained. With eager propagation, coherence information is propagated at the release to all processors that might need it, so that the information can be discarded by the releaser afterwards. The only data structure needed is a *directory* to keep track of sharers, similar to the one used in hardware cache coherence. In LRC, coherence information is propagated at an acquire operation only to that acquirer and therefore cannot be discarded at a release since there is no way of predicting which processors will issue causally subsequent acquires and when. Moreover, the releaser has to provide the next acquirer with all the coherence information that the releaser has seen in its past but the acquirer has not seen. This transitive propagation of the coherence information requires an LRC protocol to store a more complex state for much longer. This state consists of *vector timestamps* to keep track of the causal dependencies (partial ordering) among synchronization events and *write-notices* for the coherence information received from other nodes during past synchronizations (for possible transmission in the future) [KCDZ94]. This may cause LRC to be more restrictive in the sizes of problems that can be run.

On the other hand, LRC amplifies the advantages of relaxed consistency. First, fewer coherence messages are transferred, as is the case if a lock is re-acquired many times in succession by the same node. Second, LRC is likely to produce fewer page misses because ERC protocols usually apply the invalidations earlier than LRC potentially exposing more false sharing. However, if the savings in coherence propagation and page faults are small, the complexity of LRC may even cause it to perform worse than ERC.

| Benchmarks | ERC | LRC |
|---|---|---|
| SOR | 7.2 | 7.4 |
| TSP | 6.1 | 7.2 |
| Water | 3.8 | 4.6 |
| Barnes | 2.6 | 3.2 |
| FFT | 4.2 | 3.6 |
| IS | 4.4 | 5.8 |
| ILINK | 5.9 | 5.8 |
| MIP | 4.2 | 5.6 |

Table 3: Speedups for ERC and LRC, 8 processors [KCDZ95]

Table 3 reports the results of a comparison of a particular eager propagation protocol similar to Munin and the TreadMarks LRC protocol [KCDZ95]. However, the comparison is not purely about laziness since there are other differences. This ERC protocol uses a hybrid update-invalidation approach while the LRC protocol is invalidation-based. Both protocols are also multiple-writer protocols a feature that will be discussed in Section 1.5.2. The LRC protocol outperforms the ERC protocol for all applications, except for ILINK and FFT, in which the update nature of the ERC protocol reduces the page fault latency.

### 1.5.2 Laziness in Applying Coherence Information

While a protocol may *propagate* coherence operations (e.g. invalidations) eagerly to other nodes at a release, as in ERC above, various degrees of laziness can be used for actually *applying* or performing the coherence operations and in processing acknowledgments. These are often glossed over when comparing eager propagation protocols with LRC, but they can have significant performance effects and even affect the consistency model.

*Delayed consistency* (DC) [DWB+91] is an example of eager propagation but lazy application. In this case, incoming invalidations are not applied right away but are *queued* at the destinations until they perform their next acquire operation (whether causally related to that release or not). Acknowledgments are sent back as soon as invalidations are received so that the releaser can proceed. It is also possible to reduce release latency in DC by queuing the incoming acknowledgments at the releaser, and by allowing the releaser to resume immediately. In this case, the releaser has to check for the receive of acknowledgments at the next acquire. By queuing incoming invalidations, a DC protocol avoids the false sharing misses which might otherwise have occurred due to accesses between the time the invalidations are received and the next acquire operation is performed. However, it is less lazy than LRC in that it does not postpone that invalidation application until the next causally related acquire. The queue mechanisms needed by DC can be used in hardware protocols as well as in software DC protocols [DWB+91, ZIS+97].

A DC protocol and an LRC protocol (similar to the one proposed in [Kel96]) have been compared with sequential consistency on a number of SPLASH-2 applications that cover most of the key sharing patterns [ZIS+97]. Table 4 shows some of the speedup comparisons, measured on the Wisconsin Typhoon-zero hardware [SFL+94]. The lazier application and propagation of the LRC protocol have significant advantages over the DC protocol, especially in complex irregular applications that use substantial lock synchronization. The study is limited by the fact that the platform provides hardware rather than VM-based access control as well as by some performance features of the platform.

| Benchmarks | SC | DC | LRC |
|---|---|---|---|
| LU | 8.6 | 8.6 | 8.4 |
| Ocean | 2.7 | 3.8 | 5.7 |
| Water-Nsquared | 11 | 11.3 | 11.3 |
| Volrend | 0.8 | 1.7 | 2.9 |
| Water-Spatial | 4.9 | 5.8 | 7.3 |
| Raytrace | 6.6 | 8 | 9 |
| Barnes | 0.9 | 1.9 | 2.2 |

Table 4: Speedups for Single Writer Protocols: SC, DC and LRC, 16 processors [ZIS+97].

The significant difference between the protocols presented in Table 3 (Munin and TreadMarks) and those presented in Table 4 has to do with the number of writable copies of a page which are allowed to coexist and be written by different processors between synchronization points. Both Munin and TreadMarks are *multiple-writer* (ERC and LRC, respectively) protocols while the DC and LRC protocols evaluated in [ZIS+97] are *single-writer* protocols. A multiple-writer DC protocol is implemented in the Cashmere system [KS96b, KHS+97].

In *single-writer* protocols, one writer may coexist with other readers, but the moment another processor tries to write the page, it must obtain *ownership* from the previous writer since the two cannot coexist as writers. This means coherence operations are indeed invoked between synchronization points in the presence of write-write sharing. In *multiple-writer* protocols, multiple writers can coexist and locally modify their copies of a page and the copies are made consistent only at synchronization points. Overall, the trend is clearly toward lazy, multiple-writer protocols, but the performance/storage/complexity tradeoffs in laziness are not yet clear for emerging platforms and real applications.

### 1.5.3 Laziness in Propagating Data Modifications

Similarly to coherence information, data modifications can be propagated and applied with varying degrees of laziness. There are two ways of defining the laziness of data propagation: with respect to when coherence operations are propagated and with respect to when local writes or releases are performed. I will use the latter definition of laziness of data propagation.

Single-writer protocols provide the simplest solution to data propagation [Li86, BH90, Kel98]. The single writable copy of a page migrates along with the ownership, from one writer to another. Multiple-writer schemes require a mechanism to keep track of the writes by different processes to a page, and to merge them before another process needs to see them according to the consistency model.

Unlike the coherence information for which laziness in propagation and laziness in application can be separated, buffering data modifications and applying them later is not likely to provide much benefit unless this can be done without involving the main processor. Otherwise, if the main processor is interrupted or involved, it may as well apply the data right away. Since most schemes apply the incoming data as soon as they arrive, we do not discuss laziness in applying data modifications separately but incorporate it into the discussion of laziness in propagation when relevant.

A popular way to manage the tracking and propagation of data modifications entirely in software under multiple-writer protocols is to use *diffs*. At a release point, a writer determines the changes it has made to a page since the last synchronization (called synchronization *interval*) by computing the differences (diff) between the version of the page at the beginning of the interval and the version at the end of the interval. Diffs can be propagated with varying degrees of laziness.

**Eager versus Lazy Update Propagation**

The most eager form of propagating diffs is to send them eagerly along with the coherence information. This is like an update-based protocol but with less useless traffic than in hardware update protocols since updates are not propagated upon all writes but only at releases. To further reduce useless updates, early ERC protocols, like in Munin, use a hybrid update-invalidate scheme that distinguishes between *active* nodes on a page which are eagerly updated and *inactive* nodes whose copies are invalidated.

The laziest form of data propagation is to create and fetch the diffs when they are actually needed by the next sharer. This scheme can be used with eager consistency but is best exploited in conjunction with LRC. When a processor incurs a page fault, it has to obtain the diffs from the relevant writers and merge them into its copy of the page. Since diffs are not propagated unless a diff request comes in, the computation of diffs can also be delayed until then.

While ingenious, this distributed multiple writer scheme has some potential problems. First, when there is indeed a write-write sharing, a processor will need to get diffs from *multiple* writer nodes and apply them. This can lead to a large number of messages that are on the critical path and therefore increase the page fault latency. Second, the need to keep diffs and the write notices at the writers can generate very large *memory consumption*, even larger than the application data set size itself [ZIL96]), thus greatly limiting scalability. Periodic garbage collection is needed, which adds more overhead. Third, with a migratory sharing pattern diffs can *accumulate* rapidly and increase communication traffic. This is because each process that acquires a lock fetches the diffs, applies them and then generates a new one. Diffs cannot be discarded because they may need to be passed on to the next acquirer processor. For a certain sharing pattern this problem has been addressed, as we will see later in this section, by the adaptive LRC protocols. Finally, diff-ing requires several *memory-intensive operations*: a page copy to create a clean version of the page at the beginning of the synchronization interval, word by word comparison to compute the diff and applying the diff upon a page fault. In addition to their direct costs, these operations substantially pollute the primary cache and hurt performance further.

### Home-Based Protocols: An Intermediate Approach

An intermediate form of laziness in data propagation is to propagate the updates in two stages. In the home-based protocols proposed in this dissertation a *home* node is selected for each shared page and updates are propagated eagerly to the home only [IDFL96]. Updates are sent to the home either as soon as the corresponding local writes are performed, assuming that a communication support for fine-grain remote writes is available [BLA+94, BAB+98], or at the synchronization time when they are detected using an all-software technique such as diff-ing. Unlike the home copy, the non-home copies of a shared page are kept coherent through invalidations. On a page fault, a non-home processor obtains the whole page from the home.

This dissertation introduces two home-based protocols with lazy release consistency coherence propagation [IDFL96, ZIL96] and other two home-based protocols with scope consistency [ISL96c]. The Automatic-Update Release Consistency (AURC) [ISL96a] uses hardware support for fine-grained remote writes called *automatic update* provided in SHRIMP, a PC-based multicomputer [BLA+94, BDFL96, FAB+96, BAB+98] built at Princeton. The all-software Home-based LRC protocol (HLRC) [ZIL96] propagates the updates to the home as diffs. Diffs are created at the release time, immediately sent to the home and then discarded. At the home, diffs are applied as soon as they are received, and then discarded there too. Diffs can be propagated to the home eagerly, at the release, or lazily, at the next incoming acquire. However, unlike in traditional LRC protocols [KCZ92], diffs must be propagated at least by the time of the next causally related acquire, so that the data reach the home by the time the page fault occurs.

The potential disadvantages of the home-based multiple-writer scheme are (i) that the whole page is transferred on a fault even if only one word of it has been modified and (ii) that performance can be poor for some applications if homes are not assigned properly. The performance advantages are (i) fewer messages since page faults can be always satisfied with a single round-trip to the home, (ii) diffs are applied only once at the home and (iii) writes performed at the home do not generate diffs. This last feature can make a big difference to performance for applications in which a page is mostly written by a single processor. Adaptive no-home LRC protocols [ACDZ97, Kel98] described in the next section can also avoid diff-ing if a page is written by a single process.

| Benchmarks | SW-LRC | HLRC |
|---|---|---|
| LU | 8.0 | 7.9 |
| Ocean | 6 | 8.7 |
| Water-Nsquared | 11.6 | 11.8 |
| Volrend | 3 | 9 |
| Water-Spatial | 7 | 12 |
| Raytrace | 9 | 13 |
| Barnes | 3 | 6 |

Table 5: Speedups for Single Writer and Multiple Writer HLRC, 16 processors [ZIS$^+$97].

Tables 5 and 6 compare the home-based LRC (HLRC) with a single-writer LRC and with a traditional no-home, multiple-writer LRC, respectively. Multiple-writer protocols are effective for irregular applications that tend to exhibit write-write false sharing and lock-based synchronization, and are expensive for applications whose patterns of page access are mostly single-writer [Kel96]. In this dissertation, I show that an all-software home-based multiple-writer protocol (HLRC) substantially outperforms a no-home traditional LRC and that the performance gap between HLRC and LRC increases with the number of processors.

Earlier simulation-based comparisons between AURC and a no-home LRC can be found in [IDFL96, ISL96a].

| Benchmarks | LRC | HLRC |
|---|---|---|
| LU | 11.5 | 13.9 |
| SOR | 13 | 22.7 |
| Water-Nsq | 11.7 | 18.9 |
| Water-Spatial | 14 | 20 |
| Raytrace | 10.6 | 26.8 |

Table 6: Speedups for traditional LRC and HLRC, 32 processors [ZIL96].

An alternative multiple-writer scheme that captures some properties of the home-based update collection was used in [KCDZ95] to implement an ERC protocol (this is compared with TreadMarks-style LRC in Table 3). On a release, invalidations are sent eagerly to sharers as usual; on receiving invalidations, the sharers that have modified their copies send back their diffs to the releasing processor, bringing it up to date. A subsequent page fault fetches the full page only from this last releaser, just as home-based protocols fetch it only from the home.

### Adaptive Protocols

Two recent papers [ACDZ97, Kel98] independently propose *adaptive* no-home LRC protocols which dynamically switch between single-writer and multiple-writer update collection schemes based on access patterns. If write-write sharing is not detected within a synchronization interval, the protocol switches from multiple-writer to single-writer for that page, transferring the whole page when necessary and eliminating diffs and extra messages. These adaptive protocols are particularly well-suited to detecting and handling migratory sharing patterns, which are single-writer in a given interval. Due to its static assignment of homes to pages, a pure home-based protocol does not handle migratory true sharing well. It generates one and a half roundtrip messages for each migration: an eager propagation of the diff from the last writer to the home plus a roundtrip to get the page from the home to the next writer. A single-writer protocol or an adaptive no-home LRC require a single roundtrip message in this case. On the other hand, home-based multiple-writer protocols can handle write-write sharing more efficiently than an adaptive no-home LRC. While adaptive protocols outperform the traditional multiple-writer LRC (Table 7), the comparison with home-based protocols is not conclusive yet; a recent study [Kel98] compared an adaptive LRC with an HLRC implementation which is different from the one proposed in this dissertation.

| Benchmarks | LRC | A-LRC |
|---|---|---|
| IS | 1.2 | 2.1 |
| 3D-FFT | 3.4 | 4.2 |
| SOR | 6.7 | 6.8 |
| Water | 3.4 | 3.5 |
| TSP | 5.8 | 5.2 |
| Shallow | 5.1 | 5.6 |
| Barnes | 3.8 | 3.9 |
| Ilink | 5.1 | 5.1 |

Table 7: Speedups for LRC and Adaptive LRC, 8 processors [ACDZ97].

**Alternatives Update Propagation Methods**

Finally, to eliminate the overhead of diffs, other methods for tracking and merging modifications in multiple-writer protocols were proposed. One alternative is to maintain per-word dirty bits in software, as introduced in the Midway system [BZ91]. This requires instrumenting memory operations in the program, incurring some runtime overhead. This has been shown to perform slightly better than diffs for migratory sharing patterns. However, the only comparison in the literature shows that a TreadMarks-style LRC protocol with diffs performs overall better than the one with software dirty bits due to the cost of instrumentation [ACD+96]. The other alternative to incurring software diff-ing overhead is to use hardware support for either computing and applying diffs [BKP+96] or for propagating fine-grained writes to the home, thus eliminating diffs altogether [IDFL96, KS96b].

## 1.6 Architectural Support

Simple architectural support can be used to accelerate SVM protocols. It can also greatly impact the design choices for protocols and the results of performance comparisons, as well as the values of architectural performance parameters. High-level tradeoffs among protocols are clearly quite dependent on lower-level implementation mechanisms used both on architectural support and system parameters, and evaluation studies should take these into account. These effects and their influence on protocol comparisons are currently not very well understood.

### 1.6.1 Broadcast and Multicast

Interconnects that support broadcast or multicast may make an ERC protocol attractive since invalidations and even diffs in an update protocol can be multicast to sharers on a release. Eager protocols may then outperform lazy protocols. For example, the Cashmere protocol uses the broadcast support provided by the DEC Memory Channel interconnect to handle directory updates and synchronization [KHS+97]. More importantly, Cashmere uses the broadcast communication medium to ensure a total ordering on the directory operations which additionally simplifies the protocol by eliminating the need for acknowledgments and techniques to preserve the partial order.

Brazos [SB97] is a good example of using the benefit of multicast support and at the same time minimizing the side-effects of employing it when it is unnecessary. For the latter purpose Brazos uses two adaptive techniques: an early update protocol aiming at reducing the burst in communication after a global synchronization and a dynamic copyset reduction which removes inactive copies from the update copyset and switches them back to the default invalidation protocol.

### 1.6.2 Fine-grain Remote Operations

Support for fine-grain remote operations in the network interface may also affect some protocol design decisions. The fine-grain remote write mechanism investigated in this dissertation is called *automatic update*. With automatic update, writes to pre-mapped shared pages are snooped on the memory bus and transparently propagated to the home copy of the page if it is located remotely. As a result, a home-based multiple writer scheme can potentially generate larger traffic on the memory bus and network. Shared pages are mapped as write through in the caches to allow local writes be snooped on the bus. Support for automatic update was provided in the SHRIMP multicomputer through a snooping device on the memory bus connected to the SHRIMP network interface [BLA+94, BAB+98].

The results of home-based LRC implementation with automatic update support (AURC) on a 16-node SHRIMP multicomputer presented in this dissertation indicate that AURC outperforms the all-software home-based LRC (HLRC) [IBD+98, BAB+98]. Other studies have addressed the idea of using remote memory operations as a support for software shared memory with different degrees of hardware intrusion [CF89, SMV95, BR90, JJT93, PL93, PL94, KS96b, KHS+97, BIS98].

### 1.6.3 Programmable Network Interface

In an SVM implementation without a dedicated protocol processor incoming requests interrupt the compute processor and are handled by it. The interrupt overhead is the most significant parameter of the communication architecture in determining the performance scalability of a SVM protocol [BS97]. Using polling as an alternative to interrupts in handling remote requests may improve performance, or it may not, depending on the ratio of interrupt cost to polling overhead on a particular platform. The overhead of interrupts or message handling in general can affect the tradeoff between delayed consistency and an eager-application ERC. If this overhead is high, the advantage of queuing over immediately performing invalidations will be smaller since the overhead dominates and must be incurred anyway. If messages can be queued and acknowledged without interrupting the compute processor, then DC is likely to show its performance benefit over ERC. Similarly, the interrupt overhead affects the tradeoff between LRC and ERC. Higher handling costs favor LRC since LRC reduces the number of messages. Invalidations without interrupts favor ERC.

In addition to support for fine-grained remote writes, a smart network interface or co-processor that carries out some protocol or DMA operations can help accelerate SVM protocols by avoiding processor interrupts without making the processor poll. Such a facility can be used for handling page and synchronization requests as well as for directly transferring incoming data to user memory and bears further research.

### 1.6.4 Latency Tolerance

Since communication costs in SVM systems are large, latency tolerance plays an important role. The traditional latency tolerance approach used in SVM systems is to relax the consistency model. Recent studies indicate that prefetching and multithreading can effectively reduce the communication latencies when they are large, as in the ATM or Ethernet-based interconnects [MCL98, SB98]. However, when most of the cost is in end-point overhead of message handling and interrupts, latency tolerance is likely to be more effective when much of this processing is offloaded to a network interface or communication processor.

## 1.7 Application-driven Performance Evaluation

Another major area enabled by SVM progress is research in applications for these systems. A significant limitation of most of the studies performed in the past has been with applications. "Standard" applications that are used for hardware-coherent shared memory have only recently begun to be used, and most programs that have been used have been kernels with mostly very simple behavior.

In my dissertation, I show that performance evaluations in software shared memory should use a wider range of applications which can be classified according to their *induced* and not *inherent* sharing patterns [ISL96a]. The induced sharing patterns are a result of the interaction between the sharing patterns inherent to the application and the coherence granularity at which the shared memory protocol works. The gap between hardware cache coherence and software shared memory is therefore quite large (e.g. [ISL96a]), and software shared memory performance has not been proven to scale except for a limited class of applications. Some key bottlenecks that occur in irregular applications have been shown to generate cascading effects. Expensive page faults cause *dilation* of critical section (see section 2.5) which, in turn, amplifies lock contention and the waiting time for synchronization. Protocols should be adapted to alleviate these and other faults that are discovered.

Structuring the applications or data structures more appropriately for page-grained SVM can change the performance dramatically [JSS97]. Simultaneous research in applications and systems can help us in several ways: (i) understanding whether SVM can indeed work well for a wide range of applications; (ii) improving protocols, systems and architectural support to match the applications better; and (iii) understanding how to program applications to better interact with the protocols and to port well in performance across hardware and software shared memory systems. In the second area, early work done in application-specific protocols for fine-grained software shared memory showed fairly dramatic performance gains [BFRS$^+$94]. In the SVM context, early research has been done in having the compiler detect mismatches between application sharing patterns and protocols and adjust the protocol accordingly [DCZ96]. For example, the compiler may be able to detect write-only pages and prevent the SVM protocol from invalidating them at synchronization points or it can indicate to the protocol which data should be eagerly propagated at a barrier in order to reduce the following read miss latency [DCZ96]. Alternatively, the user may indicate the sharing patterns to the system.

For restructuring applications, a study was performed to examine application restructuring for SVM systems and the performance portability of applications across SVM and hardware-coherent systems [JSS97]. The study found that in most cases the applications can indeed be restructured to deliver good performance on moderate-scale SVM systems. For example, since synchronization is particularly expensive because of communication, parallel algorithms can be used that trade some computational imbalance for reduced synchronization frequency. However, the improvements needed are usually quite substantial, far more toward algorithmic changes than simple padding and alignment. Fortunately, most of the changes are performance portable to hardware-coherent systems, though with much smaller impact. More research is needed in developing guidelines for structuring applications in performance-portable ways for the different types of available systems.

Another very important area is the development of tools [LJI+98] for understanding performance bottlenecks in applications, as well as correction tools for checking violations of consistency.

## 1.8 Other Directions in SVM

This section presents two recent directions for research in SVM: (i) comparing page-grained SVM with recently re-popularized fine-grained software shared memory approaches that do not rely so heavily on relaxed consistency models and (ii) using widely available multiprocessor rather than uniprocessor nodes to reduce the frequency of software involvement.

### 1.8.1 Fine-grain Software DSM

An alternative approach to page-grained SVM is to build software DSM systems with coherence performed at fine or variable grain, such as that of cache blocks. In these cases, access control and coherence are provided by instrumenting memory operations in the code rather than through the virtual memory mechanism [SFL+94].

The advantage of the fine-grained approach is simplicity since it can use a sequential consistency model without suffering much false sharing and communication fragmentation. The programming model is exactly the same as in the more mainstream hardware DSM.

In addition, tools for code instrumentation operate on executables rather than on source code, which represents a substantial advantage over traditional SVM when source code is not available. However, code instrumentation is not always portable and adds overhead on each load and store, and since communication may be more frequent, the approach depends critically on low-latency messaging.

While Blizzard-S [SFL+94] was the first all-software fine-grain DSM, the recent Shasta system breathed new life into the approach by demonstrating quite good performance (Table 8) on a high-performance configuration [SGT96]. This was possible due to a number of subtle optimizations in instrumentation, some of which are specific to RISC or even to Alpha architectures. Among protocol-level optimizations, support for multiple coherence granularities on a per data structure basis was found to be particularly useful. It helps optimize the prefetching effect of large granularity communication without inducing false sharing, though it does rely on support from the programmer.

| Benchmarks | Shasta |
|---|---|
| LU-Cont | 4.8 |
| Water-Nsq | 4.2 |
| LU | 3.8 |
| Barnes | 3.4 |
| Volrend | 2.8 |
| Water-Spatial | 2.0 |

Table 8: Shasta, 8 processors [SGT96]

A recent study compared the fine-grained and page-grained approaches using four different consistency protocols on a fairly large and varied set of applications [ZIS+97]. The prototypes were performed on the Typhoon-0 platform, which allows protocols to be run in software but uses a uniform *hardware* access control mechanism for both approaches. The page-grained approach was represented by the home-based LRC protocol proposed in this dissertation. The results show that for almost all applications the page-grain home-based LRC protocol performs similarly to or better than a fine-grain SC protocol, except in one case where there is an overwhelming amount of synchronization (Table 9). However, this result corresponds to a particular platform and does not use full software access control. More research is needed to understand the tradeoffs between the two approaches under different communication and architectural parameters, and to determine whether one is clearly superior to the other given future trends in communication performance.

| Benchmarks | FG- SC | HLRC |
|---|---|---|
| LU | 6.0 | 7.9 |
| Ocean | 6.1 | 8.7 |
| Water-Nsquared | 12.6 | 11.8 |
| Volrend | 6 | 9 |
| Water-Spatial | 12.7 | 12 |
| Raytrace | 14 | 13 |
| Barnes | 7 | 6 |

Table 9: Fine-grain SC and HLRC, 16 processors [ZIS+97]

### 1.8.2 SVM across Multiprocessor Clusters

Another recent development in software DSM was motivated by the increasing popularity of small-scale hardware-coherent multiprocessors and the development of "systems of systems" or clusters that use these multiprocessors as their nodes [ZSLW92, CDK+94, KS96a, ENCH96, YKA96, SISL96, BIMS96, BIS98, SDH+97, SG97, SBIS98]. The home-based LRC protocol proposed in this dissertation was adapted and implemented on a cluster of SMPs [SISL96, SBIS98]. There are two strong arguments for using a software DSM layer across the nodes in such a cluster. First, it provides a uniform, coherent shared memory programming interface rather than a hybrid message-passing/shared-memory interface for this increasingly important high-performance computing platform. Second, because the local coherence within the multiprocessor is performed in hardware, the overall performance of the system is expected to be better than the performance of software DSM on clusters of uniprocessors, due to less frequent software involvement in interprocessor communication. Software shared memory may be a good way to extend this attractive programming model from moderate-scale hardware-coherent systems built in industry to much larger systems of systems.

Building an efficient software DSM on a multiprocessor cluster is not as simple as porting a uniprocessor protocol. Issues like laziness, multiple writers and protocol handling must be revisited. With respect to laziness, the question is how quickly does the incoming coherence information from outside propagate within the multiprocessor. For example, when one processor has to invalidate a page, should the page also be invalidated for other processors in that node [SBIS98, SDH$^+$97]? The answer to this question also depends on the execution model (processes or threads) used within a node. Moreover, there is a tension between the need for some software involvement even within a node to provide laziness and the need to reduce software involvement and exploit hardware sharing within a node. For multiple-writer schemes, the question is how to prevent other processors in the same node from writing into the page while one processor fetches the page as a result of an invalidation. TLB shootdown within the node is one solution but can be expensive [ENCH96]. An alternative scheme called *incoming diffs* was suggested in [SDH$^+$97]. For protocol handling, there are questions about whether to use a dedicated protocol processor within the node and, if not, how to distribute protocol processing in response to incoming messages across processors [KS96a]. Systems have also been developed to implement SVM across hardware DSM rather than bus-based machines [YKA96], which is a promising direction for building truly large scale shared memory systems.

Studies have shown that using multiprocessors rather than uniprocessor nodes in software DSM systems indeed improves performance, with the extent of the improvement depending on the localization of communication in the application [BISS98]. The caveat is that cross-node communication per unit time actually increases since the multiprocessor node speeds up computation faster than it reduces cross-node communication, thus the node-to-network bandwidth may have to be scaled with node size. The use of multiprocessor nodes may also change the tradeoffs in laziness especially as nodes become larger.

## 1.9 Summary of Dissertation Contributions

This dissertation has four main contributions in the area of shared virtual memory which were published in the *Proceedings of the 2nd International Symposium on High-Performance Computer Architecture* [IDFL96], the *Proceedings of the 23rd Annual International Symposium on Computer Architecture* [ISL96a], the *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures* [ISL96c], and the *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation* [ZIL96].

This dissertation proposes the *home-based* approach to build lazy release consistency protocols simpler, more effective and more scalable than the traditional lazy release consistency. Two home-based protocols were developed and evaluated. Automatic Update

Release Consistency (AURC) [IDFL96] is a home-based protocol that exploits a hardware support for fine-grained remote writes called *automatic update.* Home-based Lazy Release Consistency (HLRC) [ZIL96] is an all-software home-based protocol that uses diff-based update detection and collection. The results of a comparative evaluation on a 64-node Intel Paragon [ZIL96] shows that the HLRC protocol performs and scales better than a traditional LRC implementation.

This dissertation also investigates the impact of memory-mapped network technology, in particular of automatic update support, on home-based protocol design and performance. By trading potentially higher bandwidth requirement for lower memory and protocol overhead the home-based protocols are better suited to the current network technology than the traditional LRC. AURC and HLRC prototypes were implemented on SHRIMP, a PC-based multicomputer that supports user-level memory-mapped communication and automatic update through a custom-designed network interface [BLA+94]. The protocol evaluations on SHRIMP platform [IBD+98, BAB+98] indicate (i) that zero-copy memory-mapped communication can significantly improve the efficiency of home-based protocols, and (ii) that the automatic update support is an effective SVM accelerator.

This dissertation introduces *scope consistency* (ScC) [ISL96c], a consistency model that offers most of the potential performance advantages of entry consistency without requiring explicit association between data and synchronization. Instead, scope consistency exploits the implicit and dynamic association between data and synchronization defined by the memory accesses that occur inside scopes. A scope is delimited by the critical sections protected by the same lock. Simulations of ScC implemented using automatic update show that ScC reduces false sharing for irregular applications with lock-based synchronization.

Finally, this dissertation includes a classification of parallel applications [ISL96a] which can serve as a framework for understanding the performance of shared virtual memory protocols. This classification is based on the sharing patterns that are *induced* by the interaction between the sharing patterns inherent in the algorithm and the coherence granularity imposed by the protocol implementation. My dissertation includes an extensive simulation-based evaluation of AURC and traditional LRC protocols on the Splash-2 application suite using this classification framework for performance understanding.

## 1.10 Contributions to the Dissertation

Matthias Blumrich designed and implemented the SHRIMP network interface [BLA+94, BDFL96, IBD+98]. Cezary Dubnicki designed and implemented the virtual memory-mapped communication (VMMC) library for SHRIMP [DIFL96]. Yuanyuan Zhou implemented and evaluated the HLRC protocol on the Intel Paragon [ZIL96].

## 1.11 Organization of the Dissertation

My dissertation begins with an overview of the SVM research which places various ideas, systems and results in a comprehensive framework.

In chapter 2, I introduce the home-based lazy release consistency approach and describe two home-based protocols Automatic Update Release Consistency (AURC), and Home-based Lazy Release Consistency (HLRC).

In chapter 3, I develop an application classification which will serve as a framework for understanding the performance of shared virtual memory protocols. This chapter also includes an extensive simulation-based evaluation of AURC and traditional LRC protocols along with a cache coherent hardware shared memory.

In chapter 4, I present the implementation of the home-based protocols and the experimental evaluations on two different hardware and software platforms: Intel Paragon and Princeton's SHRIMP. The evaluation focuses on two issues: scalability and the impact of memory-mapped communication and automatic update on home-based protocol design and performance.

In chapter 5, I propose a new memory consistency model called scope consistency (ScC) and present a simulation-based comparative performance evaluation of ScC and AURC.

My dissertation ends with conclusions and future work.

# Chapter 2

# Home-based Protocols

Lazy release consistency (LRC) is an efficient way of implementing shared virtual memory. Traditional multiple-writer LRC protocols have demonstrated reasonably good performance for certain classes of applications on small and medium-scale systems [KCDZ94], but protocol overhead becomes substantial on large-scale systems. A traditional multiple-writer LRC protocol does not scale well because its distributed write collection scheme can generate substantial communication and memory overhead. Moreover, as the performance of the interconnect and messaging layer has significantly improved compared with the performance of the memory system, the LRC trading of memory overhead for communication bandwidth is less motivated today than it was at the time when LRC was proposed.

In this chapter, I propose a home-based multiple-writer approach to implement lazy release consistency in a more scalable way. In home-based protocols each shared page has a home to which all writes are propagated and from which all copies are derived. The home-based approach shifts the overhead from the memory system to the communication system. Home-based protocols can be implemented either exclusively in software or with hardware support for fine-grained remote writes.

Automatic Update Release Consistency (AURC), presented in this dissertation, is the first protocol to take advantage of memory-mapped communication to implement an LRC protocol [IDFL96]. AURC uses memory-mapped communication in two ways. First, it maps non-home copies of a shared page to the home page. Once these mappings are established, the network interface ensures that writes performed to the local copy of the page are transparently propagated to the home copy by the automatic update (AU) mechanism. Second, it uses memory-mapped communication to transfer data from the home page to a non-home page. Such a data transfer requires very little software overhead at the sender node, no memory copy, and no additional overhead at the receiving node.

A home-based LRC protocol can also be built entirely in software. In this dissertation, I present Home-based Lazy Release Consistency (HLRC), an all-software LRC protocol that uses diffs as in traditional LRC for write detection and propagation. HLRC has several advantages over the traditional no-home LRC protocol. It maintains a simpler state, sends fewer messages, has a lower diff overhead and a much smaller memory consumption. This profile of the home-based protocol overhead is capable of providing substantial improvements in performance and scalability over the traditional LRC protocol. It also allows HLRC to profit more than LRC from the memory-mapped communication.

## 2.1 Lazy Release Consistency

Lazy release consistency [KCZ92] (LRC) is a variant of release consistency [GLL+90] (RC) intended for software implementation.

Release consistency guarantees memory consistency only after explicit synchronization points. This allows an SVM system to hide the latency of update propagation by using either buffering or pipelining techniques. The release consistency protocol requires programs to be properly labeled by explicitly marking all synchronization points either as *acquire* or *release* operations.

Lazy release consistency delays the propagation of page invalidations until the latest possible acquire time. To achieve memory coherence with such a degree of laziness, the LRC protocol uses *vector timestamps* to maintain the "happened-before" partial ordering between synchronization events. A timestamp is a logical counter of the local *intervals* which are delimited by the synchronization events. A vector timestamp indicates the latest interval that a node has seen from every other node.

### 2.1.1 Coherence Operations in LRC

When a node acquires a lock, the node sends its *vector timestamp* in its lock request message. The last releaser of the lock uses the vector timestamp to determine the differences between the set of intervals seen by itself and the set of intervals seen by the acquirer. The differences are sent back with the lock reply as *write notices* which are lists of page invalidations, one for each interval. Thus a write notice that a requester receives for an interval contains the identities of the pages that were modified in that interval. The requester needs to invalidate them in order to see the writes performed in that interval. After the acquirer invalidates all pages according to the write-notices received, it updates its vector timestamp to reflect the intervals seen so far. In this way an LRC protocol implements the *happened-before* ordering between the events performed before the release and the events that occur after the acquire, thus satisfying the RC memory model.

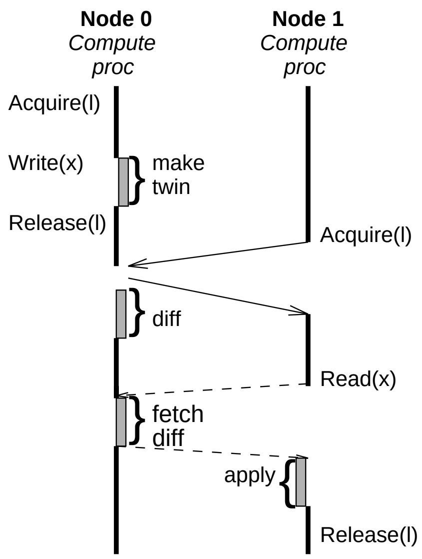



Figure 3: Lazy Release Consistency

The LRC protocol used for comparison in this dissertation is a multiple-writer protocol using a *diff*-based scheme for write detection and collection. Every writer locally records the changes it makes to every shared page during each interval. When a processor first writes a page within a new interval, it saves a copy of the page, called a *twin*, before writing to it. When the interval ends, the processor compares the current (dirty) copy of the page with the twin to detect updates and records these in a structure called a *diff*. The LRC protocol creates diffs either eagerly, at the end of each interval, or lazily, on demand. In our implementation of the LRC protocol, diffs are created at the end of intervals.

On a page fault which occurs at the first access to an invalidated page, the faulting node collects the diffs from all the "last" writers to that page and applies those diffs to its local copy to reconstitute the coherent page. There can be more than one last writer because *happened-before* is a partial order relation.

Figure 3 shows how the traditional LRC protocol works with a simple example. Assume node 0 acquires and releases lock $l$ followed by node 1. Initially all shared pages are write-protected. When node 0 writes $x$, a protection fault occurs. The LRC protocol copies the page into a twin-copy, adds it to the list of updated pages in the current interval and unprotects the page. When node 1 acquires the lock from node 0, node 0 creates the diff by comparing the twin-copy with the current (dirty) copy of the page. This diff contains the updated value of $x$. At the same time node 1 invalidates the page containing $x$. Thus, when node 1 reads $x$ a page fault occurs. In the page fault handler node 1 fetches the diff from node 0 and applies it on its local copy to incorporate the update of $x$.

### 2.1.2 LRC Limitations

The LRC multiple-writer protocol was a major advance in shared virtual memory. Since multiple nodes can update the same page simultaneously, the protocol can greatly reduce the protocol overhead due to false sharing. By delaying coherence actions until an acquire event, the protocol can reduce the number of messages for protocol and data, and hence reduce software protocol overhead. Furthermore, the protocol can reduce the communication traffic due to data transfer: instead of transferring the whole page each time the protocol transfers diffs to propagate updates.

However, the traditional LRC multiple-writer protocol based on distributed diffs has several drawbacks. The first problem is that a node may have to obtain diffs from several nodes when there is substantial write-write false sharing - precisely the case in which the multiple writer mechanism is useful. Obtaining diffs may require multiple round-trip messages upon a page fault. When all nodes are last writers, the faulting node has to send $n$ - *1* messages to get the diffs (where $n$ is the number of nodes). In the best case, when the sharing pattern is migratory, diffs are totally ordered and can be fetched in one message from the last writer. However, without using an adaptive scheme of the type recently proposed [ACDZ97, Kel98] the sizes of the messages increase during the migration due to diff accumulation. This is because the faulting processor has to fetch not only the diffs created by the last writer but also those created by previous writers in the intervals that it has not yet seen.

Another problem with distributed diffs is that the same diff may need to be applied as many as $n$ - *1* times, once at each node that fetches that diff. Diff application and diff creation incur a lot of cache misses, so diff application should be minimized.

Finally, distributed diff schemes usually cause diff storage in the worst case for every page, every interval, and at every node. This is because the node that creates a diff needs to store it locally until no other node needs it, which is difficult to predict. Moreover, a node also stores the diffs it fetched from other nodes. This is to allow the protocol to reduce the number of nodes it has to visit for diff collection upon a page fault. When implementing the protocol on a large-scale machine, memory consumption can become a severe problem. Garbage collection is usually triggered at global barriers to free up the diff storage, but this garbage collection is itself an expensive global operation that increases protocol overhead.

## 2.2 Coherence in Home-based Protocols

In this dissertation, I propose a home-based scheme as an alternative approach to supporting multiple concurrent writers in lazy release consistency. This approach combines eager propagation of data at release time with lazy invalidations. For every shared page, a single node is designated as the page's *home*. The home's copy of the page is eagerly updated before or at the next synchronization point. Non-home copies are invalidated on the next acquire operation according to the standard LRC algorithm and they will be updated on demand at page fault time by fetching the whole page from the home.

There is a coherence problem in supporting a release consistency memory model with a home-based protocol: the home's copy of a page must contain the updates required by the consistency model when the page is accessed at or fetched from its home.

If the interconnect guarantees a global ordering of events (such as an Ethernet-based network), then the consistency is implicitly guaranteed as long as the updates arrived at home are performed on the home's copy before subsequent page fetches are performed.

In the general case, when the interconnect does not provide a global serialization of events, home's copy update completion must be guaranteed by the software protocol. There are several possible techniques: write flushing at the end of intervals, keeping scalar page versions generated by the home, or using vector timestamps to represent the page version. They all assume that the network delivers the messages in the order that they are sent.

### 2.2.1 Write Flushing

The simplest approach to maintain the coherence in a home-based protocol is to delay the completion of the release events until all the updates propagated to the corresponding homes are completed. This can be ensured either by having the home acknowledge receiving the diffs or by having the releaser generating a null request to the home to flush the communication link. This solution does not require any page version bookkeeping but suffers two drawbacks.

First, by not keeping version numbers for the pages cached in the local memory the protocol cannot detect prefetching. When a page is fetched from the home it may contain more recent updates than the ones necessary according to lazy release consistency. However, due to the lazy propagation of invalidations in LRC, the page can be invalidated again at a subsequent acquire thus causing an unnecessary page fault and re-fetch. Second, by having to wait for the completion of update propagation to homes the latency of release operation increases.

### 2.2.2 Page Version

A more efficient way to ensure the coherence in a home-based protocol is to use version numbers for pages. The page version number is incremented at home whenever the home receives the updates performed by a non-home writer within an interval.

Using page versions a non-home node can easily detect a prefetching effect and avoid unnecessary invalidations. The home sends the page version to a non-home node either in reply to a diff or flush message, or in reply to a fetch request along with the page itself.

At the same time, the page version numbers are included in the write notices which are sent in a lock or barrier message. By comparing the version number sent in the write notice with the current version number of the local copy, the acquire node can invalidate pages selectively. The local page is not invalidated if the local page version is greater than or equal to the required page version indicated in the write notice. This corresponds to the situation when the page does not have to be fetched from the home because it already incorporates the required updates.

### 2.2.3 Vector Timestamps

Using scalar version numbers does not eliminate the second problem, the increase of release operation latency due to waiting for updates to complete at home. The alternative approach to guarantee lazy release consistency ordering in a home-based protocol is to use *vector timestamps* to represent the page versions in a distributed fashion. In this case, the release operation can return immediately since the vector timestamp represented the page version can be updated without visiting the home.

One vector timestamp in the per-page data structure can be used to represent two different things depending on whether or not the corresponding page is valid. As long as a page is valid, the vector timestamp indicates the current version of the page in memory. We call this the *flush timestamp* vector. At home, flush timestamps are updated each time the updates corresponding to a remote interval arrive and are performed. At a non-home node, flush timestamps are updated either at the end of intervals during which the page was written, or when the page is fetched from home.

When a page is invalid the timestamp vector represents the page version that the node has to fetch from home at page fault. This is called the *lock timestamp* vector and it is updated at acquire time as a result of applying the write notices received from the last releaser. The lock timestamp vector is presented to the home as part of the fetch request. The home can delay page shipping to the faulting node if the required version is not yet available at home because the corresponding updates are not completed. Since the home sends back flush timestamp vector along with the page, prefetching can be detected and future invalidations may be avoided.

## 2.3 Automatic Update Release Consistency

One way to propagate the writes to homes is to use fine-grained remote writes (*automatic updates*) provided by the network interfaces such as SHRIMP [BLA+94]. Automatic Update Release Consistency (AURC) implements a home-based multiple-writer scheme using automatic update (AU) hardware to propagate to the home page, and merge writes performed by concurrent writers (Figure 4). The basic idea of AURC is to designate for each page a node as the home of that page, and to establish an *AU mapping* from the non-home copies of a shared page to the home copy so that local writes to these copies at the non-home nodes are automatically propagated to the home. Since automatic update is a unidirectional point-to-point remote write mechanism, updates are propagated using a hybrid hardware-software, eager-lazy scheme (see Figure 5).

1. From non-home to home, updates propagate eagerly, at word granularity through automatic update mappings.

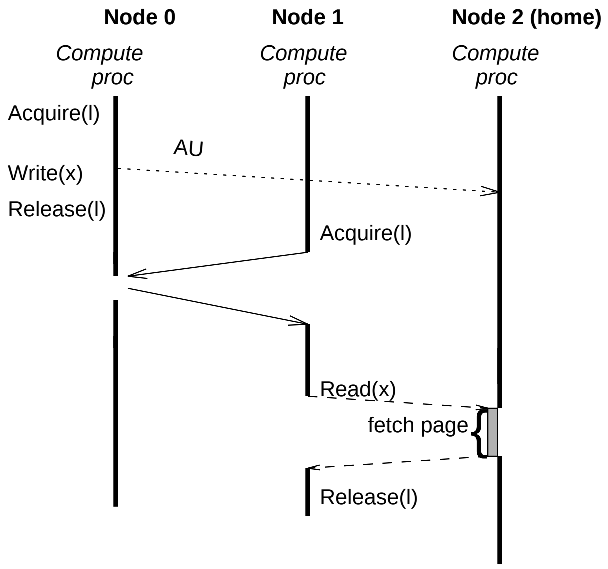



Figure 4: Automatic Update Release Consistency

2. From home to non-home, updates are fetched as a whole page at the time of page fault.

In order to guarantee update completion on the home's copy, AURC uses the flush timestamp to *flush* each data link through which automatic updates have been propagated in the last interval. This scheme assumes that the network interface delivers automatic updates in the order that they were sent. At release, flush timestamps of all the pages updated in the last interval are incremented. Since the local flush timestamp for a given page is mapped to the home's flush timestamp, the flush timestamp is propagated to the home as an automatic update when the local timestamp is incremented. The flush operation ensures that once the flush timestamp incremented by the writer is performed on the home's flush timestamp vector, all the previous writes issued by that writer are also performed on the home's copy.

## 2.4 AURC with Copyset-2 Optimization

Previous research in hardware support for update-based shared memory protocols has suggested multicast support for a full-fledged update-based coherence scheme, with updates propagated in circular rings [JJT93, BR90]. The automatic update hardware can propagate

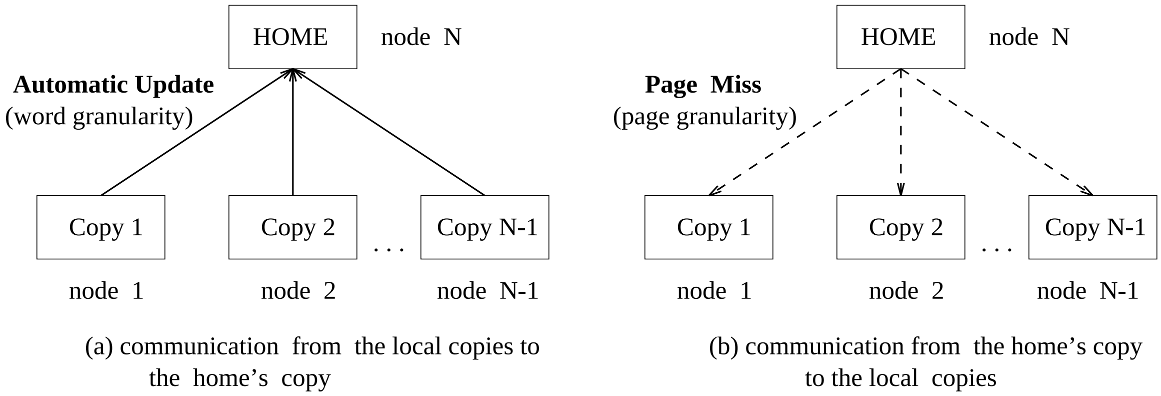



Figure 5: Update Schemes in AURC

a write to a single destination, and therefore lends itself to a home-based protocol in which writes from potentially multiple writers are propagated to a single home. Whole pages are fetched from the home through explicit messages.

The standard AURC scheme maps non-home copies of a shared page to the home copy, but does not map the home copy out to any other node. Even without multicast, an opportunity exists to map the home's copy to one other copy using the one-to-one mapping support. The *copyset-2* optimization is an extension to the AURC protocol (AURC-2) that takes advantage of the fact that the home node can also propagate its updates to a non-home page. The non-home node must still fetch the page from the home in order to receive updates from other non-home nodes.

The most valuable use of this protocol is for pages that are shared by only two nodes, one of which is the home. In this case, AURC-2 can keep the two copies consistent through the AU communication. Flushing is still needed to guarantee the completion of the remote updates, but no full-page transfers are needed after the initial page transfer.

The AURC-2 protocol is particularly useful for applications that exhibit nearest-neighbor communication [IDFL96], but it can reduce the total number of page faults for other applications as well. The possible drawback of this method, compared to standard AURC, is that AURC-2 increases the amount of overall network traffic since the home nodes also generate AU write traffic.

## 2.5 Home-based Lazy Release Consistency

Another way to propagate the writes to the home is to compute diffs at the next synchronization point and then send them to the home. This method can be used to implement a home-based protocol entirely in software. Unlike traditional LRC, in which diffs are created on demand, stored and provided possibly many times, in HLRC diffs are created at the end of intervals, immediately sent to the home, then discarded. At home diffs are applied in the arrival order and then discarded. Therefore, the lifetime of diffs is extremely short, both on the writer nodes and the home nodes. This allows memory consumption for write collection in HLRC not to increase in time or with the problem size.

The rest of the protocol is similar to AURC. The main difference between AURC and HLRC is that AURC propagates writes immediately, through the AU mappings, while HLRC detects modified words and propagates them at the next release time. AURC will therefore propagate multiple updates to the same memory location multiple times, whereas HLRC will propagate only the last change at the next release time.

Figure 6 shows an example of how an all-software HLRC protocol works. Unlike in a no-home LRC protocol where the diff is stored at the writer node and provided on demand to the faulting node, in HLRC the diff containing the new value for $x$ is sent to the home as soon as it is produced at release or acquire time. The flush timestamp of the page containing $x$ is incremented and sent with the diff. At home, the diff is applied on the home's copy of the page and the corresponding home's flush timestamp is updated. At page fault time, node 1 fetches the whole page from home. In order to incorporate the new value of $x$, the faulting node must indicate a lock timestamp that matches the flush timestamp of the interval when $x$ was updated.

## 2.6 Overlapped Protocols

To understand the potential benefit of overlapping an SVM protocol with computation, we designed an overlapped HLRC protocol which eliminates the diff overhead by using the communication co-processor present in the Intel Paragon's architecture. For comparative reasons we have also designed and implemented an overlapped traditional LRC protocol.

This section describes Overlapped Home-based LRC (OHLRC) and Overlapped LRC (OLRC), two overlapped protocol that extend the use of the communication co-processors on

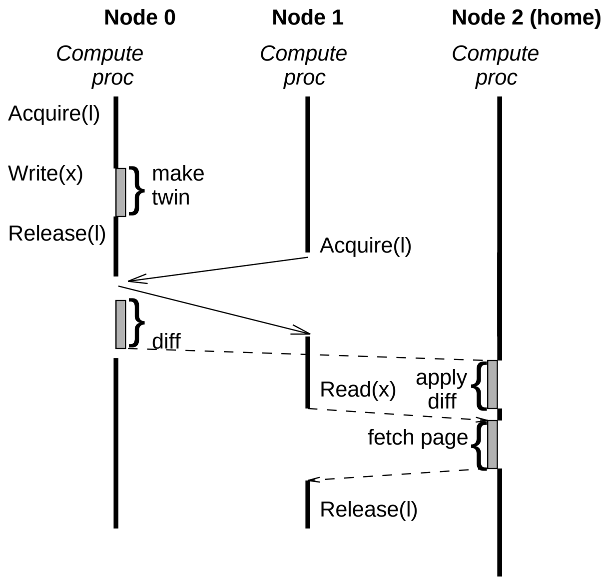



Figure 6: Home-based Lazy Release Consistency

the Paragon to protocol processing, in order to overlap some of the SVM protocol overhead with computation.

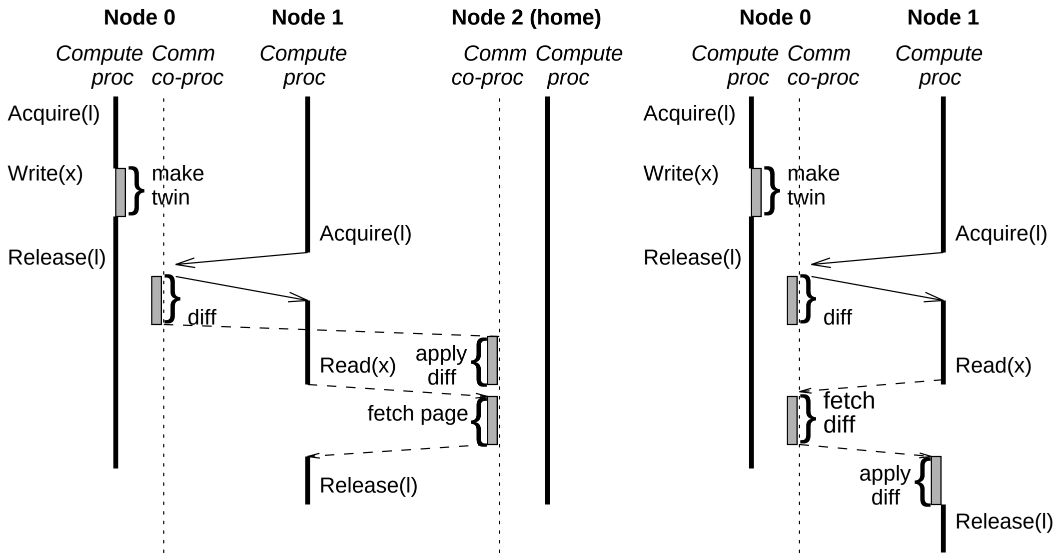
### 2.6.1 Overlapped HLRC

The Overlapped HLRC (OHLRC) uses the communication co-processor to perform three operations:

- Compute diffs after each interval and send them to the home nodes.
- Apply diffs on the local copy (at the home node).
- Service the remote requests for pages at home.

After completing a diff for a given page, the co-processor of the releaser sends the diff to the co-processor of that page's home. There the diff is applied and the timestamps of the corresponding page are updated. When servicing a page fetch remote request, the co-processor compares the timestamps in the request with the local timestamps of that page to ensure that the required updates are in place. When all the necessary diffs have been applied, the co-processor sends the page to the faulting node.

Figure 7(a) shows an example of the OHLRC protocol. To service remote acquire requests on node 0, the current interval is ended, the co-processor starts to compute a diff,




(a) Overlapped Home-based LRC (b) Overlapped LRC

Figure 7: Overlapped Home-based LRC and Overlapped LRC.

and a reply is sent back immediately to node 1, with the corresponding write notices. Node 1 invalidates the page holding x and then continues its computation. In the meantime, the co-processor of node 0 computes diffs and sends them to the appropriate home (node 2). The co-processor of node 2 receives the diffs and applies them to its local page. The read of $x$ on node 1 causes a page fault that triggers a request sent to the home node (node 2). The co-processor of node 2 services the page request after checking the timestamps.

### 2.6.2 Overlapped LRC

The OLRC approach is similar to the standard LRC protocol. The Overlapped LRC (OLRC) protocol uses the communication co-processor for two tasks: to perform diffs and to service remote fetch requests. For the first task the compute processor asks its co-processor to compute diffs at the end of each interval for all pages that have been updated during that interval. Once performed, diffs are stored and propagated to remote nodes on demand.

The second task of the co-processor is to service remote requests for diffs and pages without interrupting the compute processor. These requests are sent by remote nodes on a page access fault in order to bring their copies up-to-date according to the information previously provided by the write-notices. The co-processor sends the requested diffs if available. If a diff computation is in progress, the co-processor queues the request until the diff is ready. Full-page requests are issued only when the faulting node does not have a local copy for the page.

Other operations, such as twin creation and diff application, are still performed by the compute processor. Usually, these do not expose enough overlapping potential to justify co-processor involvement. Also, the remote lock acquire operation, which requires coherence control handling, is still serviced by the main processor. This decision was made to preserve the similarity of the overlapped HLRC with the functionality of an automatic-update based protocol.

Figure 7(b) shows an example of how the OLRC protocol overlaps its overhead with computation. Before performing the write of $x$ on node 0, the compute processor creates a twin for the page holding `x`. Node 0 then releases the lock. Later, when node 1 tries to acquire lock l, it sends a lock request to node 0. After servicing the request, the compute processor of node 0 asks its co-processor to compute diffs. After acquiring the lock, the compute processor of node 1 invalidates the page holding `x` and continues its computation. The read of $x$ on node 1 causes a page fault, which triggers a diff request sent to node 0. The co-processor of node 0 handles the request. Finally, the compute processor of node 1 receives the diff, applies it to its local copy, and continues its computation.

OHLRC and OLRC are both overlapped protocols, but their degrees of overlapping are different. Both protocols overlap diff computation and fetching with computation, but OHLRC also overlaps diff applications by performing them eagerly at the home.

## 2.7 Summary

In this chapter, I proposed a novel, home-based, write-collection for lazy release consistency SVM protocols. Compared with the traditional LRC scheme, the home-based protocol is simpler and more scalable in terms of communication and memory overhead. Moreover, the home-based efficiently exploits the memory-mapped network technology.

Two home-based protocols were described. Automatic Update Release Consistency (AURC) propagates the updates to the home using hardware support for fine-grained remote writes (automatic update). Home-based Lazy Release Consistency (HLRC) implements a home-based write collection entirely in software using the same diff mechanism as the traditional LRC. Because HLRC sends the diffs to the home as soon as they are produced rather than keeping them distributed, it has several advantages over the traditional LRC: simpler state, fewer messages, lower diff overhead and dramatically lower memory consumption due to protocol data.

# Chapter 3

# Simulation-based Performance Evaluation

While many SVM systems have been proposed, their performance on real applications is not well understood. Studies that have proposed protocols or extensions have evaluated them usually with only a small set of simple programs. The programs, problem sizes and machine parameters used have also been limited and different across studies, making comparisons difficult. Despite a few comparative studies [CDK+94, ACD+96] it is not well understood whether the SVM approach works well for important classes of applications, whether the approaches that use a small amount of hardware support have substantial advantages over all-software implementations, or how architectural and application parameters affect the answers to these questions.

In this chapter, I introduce an application classification which serves as a framework to understand the performance of shared virtual memory. This classification is based on the sharing patterns that are induced by the interaction between the sharing patterns inherent to the algorithm and the coherence granularity imposed by the protocol implementation [ISL96a]. Based on this classification framework I use detailed simulation to understand the performance of two SVM approaches— a no-home (LRC) protocol and a home-based (AURC) protocol—and compare it with the performance of the hardware-coherent shared memory approach. The hardware shared memory is a cache-coherent non-uniform memory access (CC-NUMA) architecture. To obtain a good coverage, I use ten applications from the SPLASH-2 suite [WOT+95b] that display a range of different communication patterns and granularities. In addition to sharing patterns, I examine how problem size, machine size, key system parameters, and the degree of optimization of the parallel programs affect the performance of the two approaches.

The simulation study shows that SVM protocols can indeed perform quite well on systems with 16-32 processors for several classes of applications at reasonable problem sizes. However, performance is much more variable across applications than with CC-NUMA, and the problem sizes needed to obtain good parallel performance are substantially larger. The results show that the hardware-assisted AURC performs significantly better than LRC, at least under our next-generation architectural assumptions, and the performance gap between these two protocols increases when realistic cache hierarchies are used.

## 3.1 Simulation Environment

I use execution-driven simulation driven by the TangoLite [Her94] reference generator. I integrate TangoLite with three different types of memory system simulators: one for LRC, one for AURC, and one for hardware-coherent CC-NUMA. Since the focus of this study is to understand the interactions of protocols with applications , I assume a common, generic but aggressive architectural model.

The basic model assumes a 300 MHz 1 CPI processor, with 200 ns memory latency, 1 MB of single-level two-way set associative cache with a 64 byte line size, and an 8 cache-line deep merging write-buffer. The write buffer merges writes and retires them in FIFO order. Since it is potentially important, particularly in AURC, its size is varied in Section 3.5.6. All instructions in our simulated processors, as well as cache hits cost 1 CPU cycle.

The use of a large, one-cycle access cache is unrealistic. I choose it for the bulk of our discussion because it removes the cache size as a parameter for sensitivity. In fact, it tends to slightly underestimate the speedups for AURC and hardware CC-NUMA by favoring the sequential execution, and to benefit the all-software LRC by reducing the effect of cache pollution due to diff-ing. Diffs performed in LRC not only cause misses themselves but also pollute the cache. A large cache allow the diffs to not perturb the application's working set in the first-level cache too much. However, in order to see the effects of a more realistic cache hierarchy, in Section 3.5.5, I present results for a two-level hierarchy with a 16 KB direct-mapped first-level data cache and a 1 MB two-way set-associative second-level cache (with 10-cycle hit cost).

The cache policy can be programmed on a per-page basis to be either write-back with write-allocate or write-through with write-around. The latter is used only for shared pages and only in the AURC case. In the AURC simulator, I assume a memory-mapped network interface that snoops the memory bus and is connected to the I/O bus. The overhead of initiating a deliberate transfer (e.g. a page transfer) is 1 microsecond, and the overhead of initiating an automatic update is a single CPU cycle when the write buffer is not full. The effective bandwidth of a page transfer is limited by the speed of the I/O bus, which is 32 bits wide. The page size is 4KB, so a page transfer takes 1K I/O bus cycles. The network interface has a mechanism that combines consecutive automatic updates into a single network packet, whose size cannot exceed 40 words. This feature is similar to what SHRIMP network interface supports [BLA+94]. The network topology is a two-dimensional mesh. The simulator's model contention of buses, memory, network interface in detail, except for contention in the network topology itself.

A fast interrupt handling mechanism takes 150 cycles, a reasonable cost of 4 cycles per word for twin creation in LRC, and 8 cycles per word for diff computation or processing. These and all protocol operations are actually simulated by running the protocol handlers, and the simulator also charges for the cache misses incurred during the operations.

Since both the memory and I/O buses are involved in AURC and LRC, I used two combinations of speeds called *fast bus* and *slow bus*. The fast bus setup is an 800 MB/sec memory bus and a 256 MB/sec I/O bus, while the slow bus setup is a 400 MB/sec memory bus and a 128 MB/sec I/O bus.

The hardware CC-NUMA simulator we use is a modified version of the detailed FlashLite simulator built for the Stanford FLASH machine. It is the same as the simulator used in [HHS+95], and models a hardware rather than flexible communication controller. It uses almost exactly the same parameters as described above (with an 800MB/s memory bus; the I/O bus is irrelevant), and is used only as a reference point for comparison. The cost of a remote miss to a clean line at the home is about 180 cycles and that of a miss to a dirty remote line is about 300 cycles.

## 3.2 Classification of Sharing Patterns

In this section, I provide a general classification of data sharing and communication patterns that is relevant to understanding performance on SVM systems. The dominant *inherent* *sharing patterns* in the applications is the sharing patterns at word granularity. These patterns interact with the large granularity of coherence in an SVM (a page) to construct the *induced sharing patterns* of pages, which may be very different than the inherent patterns. It is these induced sharing patterns that determine the communication actually seen in an SVM execution, exhibit *false sharing* and *fragmentation*, and hence impact performance. False sharing and fragmentation effects in SVM were defined in Section 1.2. The induced pattern for an application is a function of its inherent sharing pattern and the spatial granularity in the address space of the read and write accesses made by processors. For AURC (like CC-NUMA), the communication incurred also depends on how data are allocated and distributed among processor memories, but this is irrelevant in LRC. Table 10 summarizes the inherent and induced sharing patterns for the different applications.

| Application | Inherent Pattern | Read Access Granularity | Write Access Granularity | Induced Pattern | Induced Effect |
|---|---|---|---|---|---|
| Barnes | migratory 1P-MC | fine | fine | MP-MC | false sharing fragmentation |
| Cholesky | migratory 1P-MC | fine-medium | fine-medium | MP-MC | false sharing fragmentation |
| FFT | 1P-1C | medium | coarse | 1P-MC | fragmentation |
| LU | 1P-MC | coarse | coarse | 1P-MC | node |
| Ocean | 1P-1C | fine | coarse | 1P-1C | fragmentation |
| Radix | 1P-1C | coarse | fine-medium | MP-1C | false sharing |
| Raytrace | migratory | fine | fine | MP-MC | false sharing fragmentation |
| Volrend | migratory | fine | fine | MP-MC | false sharing fragmentation |
| Water-Nsquared | migratory 1P-MC | coarse | coarse | MP-MC | none |
| Water-Spatial | 1P-MC | coarse | coarse | 1P-MC | node |

Table 10: Classification of Sharing Patterns in the Applications

Several previous studies have proposed classifications of data objects based on access patterns, although in different contexts. Gupta and Weber [WG89] proposed a classification to characterize cache invalidation patterns in shared memory multiprocessors. Bennett *et al.* [BCZ90] expanded this classification and used it to perform adaptive software cache management on distributed shared memory machines. My classification has a different purpose: to understand the observed performance of SVM systems.

### 3.2.1 Inherent Sharing Patterns

The inherent sharing patterns are determined by the parallel algorithms and partitioning schemes used. There are three types of inherent sharing patterns relevant to the classification: one producer with one consumer (*1P-1C*), one producer with multiple consumers (*1P-MC*), and *Migratory*. *Migratory* sharing is a frequent *1P-1C* pattern in irregular applications where data are consecutively accessed in read-write sequences by different processors. It is also encountered in accumulations and the use of global counters in regular applications. The access patterns to task queues when enqueuing and dequeuing tasks are common examples. There is also read-only data, which can be thought of as 0P-1C or 0P-MC.

### 3.2.2 Access Granularity

An important characteristic that helps determine how an inherent sharing pattern is converted to an induced pattern is the granularity at which the read and write accesses to shared data from different processes are interleaved in the address space. There is coarse-grained accesses when a processor reads or writes a whole contiguous page at a time. Alternatively, when a processor's reads and writes are scattered at a much finer grain, it reads or writes only parts of a page. The read/write access can be *coarse*, *medium*, or *fine* grained. Medium means not very fine-grained but with contiguity smaller than a page. There are applications with many different combinations of read and write access granularity, and in many cases the access granularity depends on the relationship between the problem size and the number of processors.

Finally, some applications allocate many of data objects that are each much smaller than a page. If these allocations are performed on separate pages, then SVM protocol operates and communicates at the granularity of the allocations rather than whole pages. In these cases, accesses are considered coarse-grained or fine-grained at the granularity of these allocated objects rather than whole pages.

### 3.2.3 Induced Sharing Patterns and Effects

When both read and write access patterns are coarse grained in terms of pages, the induced communication pattern is very similar to the inherent pattern. In general, fine or medium grained read accesses to nonlocal data cause *fragmentation* which often, but not always, induces a multiple-consumer or MC pattern. Fine- or medium-grained writes cause *false sharing* of a page which often induces a multiple-producer or MP pattern. Fine-grained reads cause fragmentation in the FFT and Ocean with 1P-MC and 1P-1C respectively induced sharing patterns. Fine-grained writes cause false-sharing and induces MP-1C in Radix. If both the read and write access patterns of processors are fine-grained, as in many irregular applications, the induced pattern is MP-MC and exhibits both false sharing and fragmentation regardless of the inherent sharing pattern. Nonetheless, the inherent pattern is still useful in serving as a contrast to the induced pattern.

Different induced sharing patterns are handled differently by the two protocols. With proper data placement, AURC can take advantage of 1C induced patterns by making the consumer the home of that page. This avoids page misses completely in 1P-1C cases like Ocean due to the specialized protocol discussed earlier. A 1P-MC pattern allows LRC to amortize the cost of diff computation at the producer over the page misses at multiple consumers. Finally, in LRC, an induced MP pattern for a page is particularly bad because it means that upon a page fault diffs for the page will have to be obtained from multiple processors, especially if it occurs after a barrier synchronization.

## 3.3 Applications Descriptions

In this section I describe the applications focusing mostly on their dominant sharing patterns relevant to SVM systems. More detailed descriptions can be found in the references cited in [WOT$^+$95b].

**Barnes** uses the Barnes-Hut hierarchical N-body method to simulate the interactions among a system of particles over time. The computational domain is represented as an octree of space cells. The leaves of the octree contain particles, and the particles and space cells are distributed to processors based on their positions in space. At each time step of the simulation the octree is rebuilt based on the current positions of the bodies, and each processor partially traverses the tree to compute the forces on the particles assigned to it. The interaction between a particle and a group of distant particles is computed as the force between the particle and the center of mass of the distant particles; the tree structure allows the latter to be determined easily. At the end of each time step the positions of the bodies are updated based on the computed forces. The properties of particles and cells are updated in an order that has little to do with their placement in the particle and cell arrays, leading to false sharing at page granularity. This false sharing is particularly problematic in the tree-building phase because many of the updates in that phase are protected by per-cell locks.

The particle update and force calculation phases in the application are inherently 1P-MC sharing patterns. While the calculation of forces is by far the dominant component of Barnes's execution time on hardware CC-NUMA systems, on SVM systems the time required to calculate the forces is approximately equal to that required to rebuild the tree because SVM's higher communication and synchronization costs drive up the time required to rebuild the tree.

From the SVM point of view, tree-building is the interesting phase. This phase exhibits both 1P-MC and migratory inherent sharing patterns between synchronization events. However, both read and write access patterns to the global particle and cell arrays are fine-grained. The induced pattern at page granularity is therefore MP-MC, which causes substantial fragmentation and false sharing to occur.

**Cholesky** is a sparse matrix factorization kernel with inherent sharing pattern 1P-MC: the pivot row blocks once produced are consumed by several other processors. However, the sparse structure of the matrix can lead to multiple processors' assigned blocks falling on the same page. This medium granularity of block accesses induces an MP-MC pattern, and causes false sharing and fragmentation. More significant problems occur in the accesses to distributed task queues, which are inherently migratory but are fine- rather than medium-grained in both reads and writes.

**FFT** is a high-performance FFT kernel. The communication in an $n$-point FFT is essentially a matrix transposition of a $\sqrt{n}$ by $\sqrt{n}$ matrix. FFT distributes matrices so that every processor is assigned a contiguous set of $n/p$ rows, and the source and destination matrices are reversed for every transpose. In this version of FFT, each processor in a transpose reads an $\frac{\sqrt{n}}{p}$ by $\frac{\sqrt{n}}{p}$ submatrix from every other processor and writes it into its local partition or set of rows. Consequently, the write access granularity is coarse while the read access granularity depends on $n$ and $p$; this is because a processor reads subrows of size $\frac{\sqrt{n}}{p}$ from the other processors. For typical values of these parameters the read granularity is often moderate, leading to fragmentation but not false sharing. Due to the extra communication caused by the fragmentation effect, as well as the relatively high communication to computation ratio, the performance of FFT on a page-based software shared memory system is generally modest unless the problem size is very large, which decreases fragmentation.

In other words, for reasonable problem sizes FFT under SVM induces 1P-MC pattern. For very large problems it may become 1P-1C if $\sqrt{n}/p$ is a multiple of the page size and generates neither false sharing nor fragmentation.

**LU** is a kernel that factors a dense matrix into the product of a lower triangular and an upper triangular matrix. To enhance data locality and to avoid false sharing, we used a version with an optimized data structure for the matrix, namely a 4-d array, in which each 32x32 element block is allocated contiguously in virtual memory and the blocks assigned to a processor are also allocated contiguously. Barriers are used to ensure synchronization between the processor producing the pivot row blocks and the processors consuming them, as well as between outermost loop iterations. This kernel exhibits coarse-grain sharing and low synchronization to computation frequency, but the computation is inherently unbalanced. The inherent sharing pattern of blocks is 1P-MC, as in Cholesky. However each 32-by-32 block is exactly 8KB (2 pages) large, so both the read and write granularities are coarse. The induced pattern remains 1P-MC, and there is no false sharing or fragmentation.

**Ocean** is a fluid dynamics application that simulates large-scale ocean movements. Spatial partial differential equations are solved at each time-step of the program using a restricted Red-Black Gauss-Seidel Multigrid solver. Ocean is representative of the class of problems that are solved using regular grid near-neighbor computations. At each iteration the computation performed on each element of the grid requires the values of its four neighbors. Work is assigned to processors by statically splitting the grid and assigning a partition to each processor. Nearest-neighbor communication occurs between processors assigned adjacent blocks of the grid. I ran two versions of ocean with two different partitioning methods: ocean-square uses square sub-grids while ocean-row partitions the matrix in blocks of $n$/p contiguous full rows. Although the inherent communication to computation ratio is smaller when square-shaped sub-grids are used, the communication induced by the page-based granularity of the SVM protocol makes the overall communication to computation ratio worse for ocean-square than for ocean-row due to fragmentation and false sharing at the columnar partition boundaries.

In Ocean the inherent pattern is mostly 1P-1C, except at the corners of partitions. Write accesses are local and coarse-grained since a processor writes only to its own partition, which is allocated contiguously; however, the nonlocal read accesses that generate communication of border elements are medium- or coarse-grained along row-oriented partition borders but a single element along column-oriented borders. Thus, there is little false sharing but significant fragmentation in the induced patterns. The inherent 1P-1C pattern is preserved if $n/p$, which is one subrow of a partition for $n$-by-$n$ grids, is a multiple of a page; otherwise, it can becomes 1P-2C as the partitions on either side of a processor's partition access elements on the same page. While in FFT the fragmentation can be eliminated by running large enough problems, in Ocean the fragmentation remains since a processors still needs only its immediate neighbor elements.

**Radix** sorts a series of integer keys into ascending order. The dominant phase of Radix is the key permutation phase. In Radix a processor reads its locally-allocated $n/p$ contiguous keys from a source array and writes them to a destination array using a highly scattered and irregular permutation. For a uniform distribution of key values, a processor writes contiguous sets of $\frac{n}{r*p}$ keys in the destination array where $r$ is the radix used; the $r$ sets a processor writes are themselves separated by $p-1$ other such sets, and a processor's writes to its different sets are temporally interleaved in an unpredictable way. This write pattern induces substantial false-sharing at page granularity.

**Raytrace** renders complex scenes in computer graphics using an optimized ray tracing method. The accesses to the scene data, into which rays are shot in this program, are read-only and relatively uninteresting other than the fact that they cause fragmentation. The interesting communication occurs in task stealing using distributed task queues, and in updating pixels in the image plane as part of a task. Both types of access patterns are fine-grained and cause considerable false sharing and fragmentation at the page level. The original Splash-2 application was modified to reorganize the task queues and remove unnecessary synchronization to alleviate the problems observed in [ISL96a].

**SOR** is a kernel from the suite of programs used in TreadMarks. It corresponds to the red-black successive over-relaxation (SOR) method for solving partial differential equations. The black and red arrays are partitioned into roughly equal size bands of rows, which are distributed among the processors. Communication occurs across the boundary rows between bands and is synchronized with barriers.

**Volrend** renders three-dimensional volume data into an image using a ray casting method. Volrend's sharing patterns are very similar to those of Raytrace both in task queues and in the image plane, only its working set of read-only scene data is much smaller, and the accesses to scene data are less scattered.

**Water-Nsquared** simulates a system of water molecules in liquid state using an $O(n^2)$ brute force method with a cutoff radius. The water molecules are allocated contiguously in an array of $n$ molecules, and are partitioned among processors in contiguous pieces of $n/p$ molecules each. Updates are accumulated locally between iterations and are performed once at the end of each iteration. In this latter phase, a processor updates its own $n/p$ molecules and the following $(n/2 - n/p)$ molecules of other processors in the array using per-partition locks to protect the updates. Water-Nsquared exhibits a migratory sharing pattern in the updates but incurs little or no false sharing or fragmentation if the array partition size is a multiple of the page size and the partitions are page-aligned.

**Water-Spatial** solves the same problem as Water-Nsquared using a spatial directory rather than a brute-force $O(n^2)$ method; this technique is particularly beneficial for large problems. The 3-d physical space is broken up into cells, and every processor is assigned a contiguous cubical partition of cells together with the linked lists of molecules in those cells. A processor reads data from the processors that own cells on the boundary of its partition and may write the linked lists of these processors if molecules move across partitions. For large enough problems, this turns out to be mostly a 1P-1C inherent pattern with multiple consumers for some cells. For smaller problems it is mostly 1P-MC. Migration of molecules between cells requires locking of linked lists, but this migration is infrequent and therefore has little impact on performance.

## 3.4 Overall Simulation Results

Tables 11 and 12 present the speedups for the ten applications using the basic architectural model with a large one-level cache. For 16-processors executions, the tables show the speedups for each of the CC-NUMA, AURC and LRC schemes, with two problem sizes, and with two sets of bus speeds for AURC and LRC. The last two columns in Table 12 present results for 32 processors for the larger problem size and the fast bus only.

Table 13 shows the small and large problem sizes for each application. Due to simulation resource constraints I simulated only one problem size for Volrend and Raytrace. Also, for

| Application | Fast Bus | | | Slow Bus | |
|---|---|---|---|---|---|
| | CC-NUMA | AURC | LRC | AURC | LRC |
| Barnes | 13 | 9.1 | 6.2 | 6.7 | 5.7 |
| Cholesky | 14.8 | 4.5 | 2.9 | 2.7 | 2.4 |
| FFT | 14.3 | 4.6 | 4.6 | 3 | 2.9 |
| LU | 10.4 | 10.3 | 9.3 | 10.2 | 9 |
| Ocean | 5.6 | 5.2 | 3.1 | 3.4 | 2.2 |
| Radix | 20 | 1.4 | 1 | 1 | 0.8 |
| Water-Nsquared | 11.6 | 8.9 | 7.6 | 7.8 | 6.8 |

Table 11: Speedup Table for Small Problem Size, 16 Processors

| Application | 16 Processors | | | | | 32 Processors | |
|---|---|---|---|---|---|---|---|
| | Fast Bus | | | Slow Bus | | Fast Bus | |
| | CC-NUMA | AURC | LRC | AURC | LRC | AURC | LRC |
| Barnes | 14.2 | 10.3 | 6.4 | 8 | 6 | 13.4 | 8.2 |
| Cholesky | 15.1 | 6 | 4.4 | 3.7 | 3.8 | 4.3 | 3.5 |
| FFT | 14.8 | 11 | 10.2 | 8.3 | 7.6 | 14 | 11 |
| LU | 14.3 | 13.3 | 11.8 | 13.2 | 11.7 | 22 | 19 |
| Ocean | 13.2 | 11.4 | 5.3 | 8.2 | 4.1 | 17 | 8.7 |
| Radix | 16 | 2.9 | 2.7 | 2.4 | 2.1 | 3.6 | 2.8 |
| Raytrace | 14.7 | 6.6 | 7 | 6 | 6 | 0.37 | 0.28 |
| Volrend | 14.1 | 7.1 | 7.1 | 5.3 | 5.5 | 9.5 | 9 |
| Water-Nsquared | 14.4 | 11.5 | 10.2 | 10.2 | 9 | 14 | 11 |
| Water-Spatial | 15.1 | 13 | 12 | 11.6 | 10.1 | 19 | 16 |

Table 12: Speedup Table for Large Problem Size, 16 and 32 Processors

Water-Spatial I do not report the results for the small problem size used in Water-Nsquared (343 molecules) since it is too small a problem for that algorithm and causes severe load imbalance.

The hardware-coherent speedups in Table 11 and 12 are excellent, even for the small problem sizes, on almost all applications. It is also clear that the performance of both SVM schemes is more variable across applications than that of CC-NUMA. For the smaller problems, speedups are quite poor except for the regular applications that have no adverse induced communication effect, like LU or Water-Nsquared. Barnes and Ocean also deliver reasonable performance for AURC with the fast bus combination. As expected, the fast

| Application | Small Problem | Large Problem |
|---|---|---|
| Barnes | 8K particles | 16K particles |
| Cholesky | 0.3 M | 1.7 M |
| FFT | 64K points | 1 M points |
| LU | 512-by-512 | 1024-by-1024 |
| Ocean | 128-by-128 | 512-by-512 |
| Radix | 1M keys | 4M keys |
| Raytrace | - | balls4 |
| Volrend | - | $256^3$ head |
| Water-Nsquared | 343 molecules | 512 molecules |
| Water-Spatial | - | 512 molecules |

Table 13: Applications and Problem Sizes

memory bus tends to improve performance more for AURC than for LRC protocol. This is because the automatic updates stress bus performance more than diff computation.

For the applications that perform poorly for the small problems under SVM, the natural question is whether this is due to the problem sizes being too small, or whether the applications are inherently not well-suited for SVM. Using large problems helps many of the applications tremendously. For example, FFT gets a huge boost, and the performance of Water-Nsquared, Water-Spatial, and LU, as well as Barnes and Ocean for AURC, gets closer to the performance of hardware coherence on 16 processors. Cholesky achieves reasonable performance with the large problem in the fast bus case. Radix improves but continues to perform poorly.

The speedups for 32 processors are reported in the last two columns of Table 12. In some cases, such as Raytrace, Cholesky and Radix, the speedup does not improve much or even drops, although hardware-coherent machines continue to scale very well to 32 processors and beyond. In other cases, the speedups continue to increase although the parallel efficiency drops substantially. The parallel efficiency is defined as the speedup divided by the number of processors. The speedup increase for LU, Water-spatial and Ocean is substantial, while for FFT, Barnes, Water-Nsquared and Volrend it is significant but much smaller.

From these results it appears that several of the applications have the potential to run well on 32-processor SVMs with large problem sizes. What I call "large" problems are not anywhere near the high end of the sizes these applications would run in practice. To study this further, and to examine if the loss in parallel efficiency relative to hardware coherence in going from 16 to 32 processors can also be overcome by increasing problem sizes, I took a few applications with different performance behaviors and ran them on 32 processors with even larger problems (see Table 14). For Water-Spatial, FFT, Cholesky, the speedup improved substantially. However, Radix does not improve, and Barnes improves very slowly as the problem size increases. Ocean improves only a little. The detailed discussion of application performance in the next section will explain these effects. In conclusion, for several applications we can get quite good performance from SVM systems but usually by using significantly larger problems than needed to get good performance from hardware cache-coherent systems. However, for applications like Radix the performance under SVM systems appears intrinsically poor.

| Application | Problem Size | AURC | LRC |
|---|---|---|---|
| Barnes | 32K particles | 19 | 11 |
| Cholesky | 4M | 10 | 10 |
| FFT | 4M points | 20 | 19 |
| Water-Nsquared | 4096 molecules | 26 | 24 |

Table 14: Speedup table for Larger Problems, 32 Processors, Fast Bus

As for a comparison of AURC and LRC at this level, although the use of a very large one-level cache biases the results toward LRC by not letting its diffs pollute the cache much, AURC performs better than LRC. This is both because of the advantages of a home-based approach, which were discussed in Chapter 2, and because of the lack of diffs due to automatic update support. Compared with LRC, AURC benefits more from the use of faster buses, improves more quickly as larger problems are used, and preserves its speedups better for larger processor counts. The difference among the schemes varies across applications and increases with the number of processors, particularly in multiple-writer applications. In Section 3.5.5, I show that the impact of using a more realistic two-level cache hierarchy can be substantial, hurting speedups significantly for LRC but not affecting the speedups for AURC.

Finally, to capture the effect of realistic situations in which the working set of an application does not fit in the cache, I ran Ocean and FFT with smaller cache sizes as recommended in [WOT+95b]. The speedups for FFT were not much different than with the large cache, while those for Ocean exhibited a superlinear effect due to the partitions fitting in the cache as more processors were used.

The absolute performance of the SVM protocols, as well as their relative performance, varies much more widely across the applications than the performance of the hardware-coherent machine. It is also much more sensitive to issues like problem and machine size. Studies that evaluate SVM with a small set of applications and without looking at the effects of problem size can therefore be misleading. To truly understand the high-level results described above, the potential for SVM systems, and the value of the home-based approach combined with automatic update hardware support in AURC for different applications, it is very important to interpret and explain the results in light of individual application characteristics.

## 3.5 Performance Analysis

In this section, I analyze the performance of the two SVM schemes in the context of the application classification described earlier, identify the key performance bottlenecks, and explain the high-level results and trends observed in the previous section. For this discussion applications are divided into four classes. All the figures we use show the components of execution time in the SVM cases normalized to the execution time of the hardware CC-NUMA system. In all figures **H** stands for hardware CC-NUMA, **A** for AURC and **L** for LRC. The legend for the components is the same for all graphs and is shown only in Figure 10. *Computation* is the time spent executing useful instructions; *Communication*, *Synchronization* and *Stalling* are the times spent waiting at page fetches, synchronization operations, and local accesses, respectively; and *Protocol* is the time spent executing software protocol handlers, including their memory accesses.

### 3.5.1 Applications without Adverse Granularity Effects

Three of our applications are not affected by the large granularity of sharing in SVM and retain their inherent sharing patters. These are LU (Figure 8), Water-Nsquared (Figure 9), and Water-Spatial (Figure 10). In all three, the performance for both SVM protocols is fairly close to those of CC-NUMA, with AURC being at most 20% worse than CC-NUMA for 16 processors.

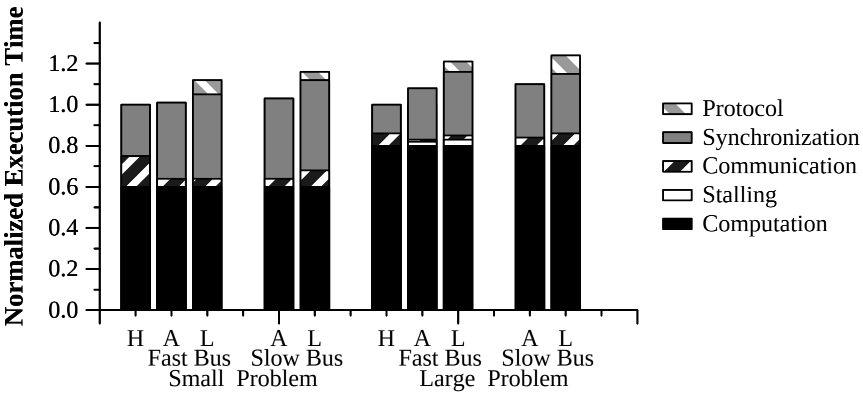



Figure 8: Results for LU. 16 Processors

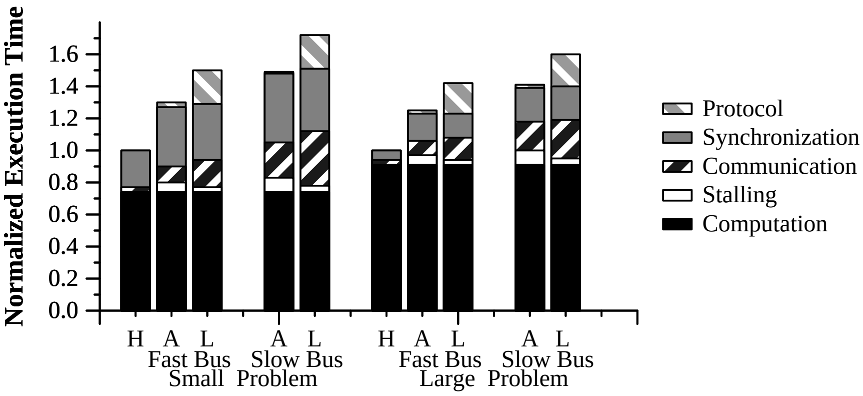



Figure 9: Results for Water-Nsquared. 16 Processors

For LU, the use of page rather than cache-line granularity in the SVM protocols provides a beneficial prefetching effect that compensates for the software protocol overheads and makes communication time lower than in CC-NUMA. This results in AURC delivering the same performance as CC-NUMA for 16 processors; however, synchronization time is always larger for the SVM protocols.

The main performance bottleneck in LU is load imbalance. While this persists for all systems, synchronization time is larger for the SVM protocols due to the protocol overhead. Of the two SVM protocols, LRC is slightly more affected because the computation of diffs at barriers increases the imbalance. For 32 processors the speedup for the SVM systems is still good although the efficiency drops due to an increased load imbalance.

In Water-Nsquared, the remote accesses are again regular and are to contiguous data, which makes communication at page grain very efficient. The inherent communication to

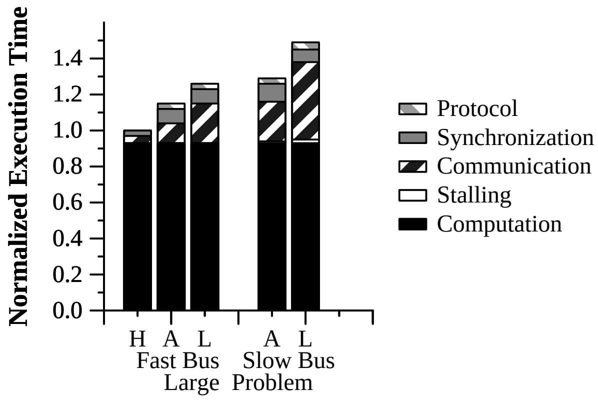



Figure 10: Results for Water-Spatial. 16 Processors

computation ratio is low. The frequency of lock-based synchronization in the interesting force accumulation phase is kept low by using temporary data structures as partial sums, but the remaining locks protect large critical sections and are quite highly contended. There are two reasons why locks are generally much more expensive in SVM systems than in hardware CC-NUMA. First, software protocol activity occurs at the synchronization points. Second, the communication (page faults) that occurs within the critical sections is more expensive which results in a large dilation of the time spent in the critical section and hence increases contention for the lock (see Figure 11). This exaggerated contention in lock-based synchronization accounts for most of the difference in synchronization cost between SVM protocols and hardware CC-NUMA; moreover, the lock serialization effect introduces an execution imbalance which increases the barrier waiting time. Thus, the expensive communication in SVM dilates the critical section causing a double impact on performance by increasing both the lock and the barrier waiting time.

The synchronization component increases at higher processor counts, limiting SVM speedup even more, but reduces relative to computation when the problem size is increased. In Water-Nsquared, diffs are large and accumulate due to the migratory data accessed at medium-coarse granularity. In this case the communication traffic actually ends up higher in LRC. While bus contention due to write-through has a visible effect for AURC, especially with the slow bus, the effect is negligible compared to other costs.

Although Water-Spatial makes accesses at the granularity of molecules, it behaves like a coarse-grained program because each molecule is allocated on a different page. The communication is again larger in LRC because diff fetching generates larger communication

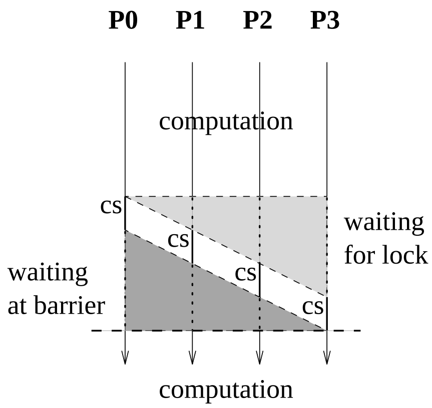



Figure 11: Critical Section (cs) Serialization

traffic than page fetching in AURC, due to diff accumulation. The speedup is quite good even at 32 processors, but the parallel efficiency drops mainly because the contention for locks increases. Increasing the problem size much further helps performance greatly (see Table 14).

Even for this class of regular applications, there is a difference between AURC and traditional LRC which is mainly caused by the inability of the traditional LRC to catch the single-writer case and avoid diffing. Of course, this would not have happened if an adaptive LRC protocol had been used instead of the traditional LRC. Once generated, the difference in communication time is reflected and amplified in the synchronization costs thereby damaging the affected protocol even more. However, for both LU and Water-Spatial the differences between AURC and LRC in the total execution times are small. This is because the communication to computation ratio is low, and the main bottleneck which is either load balance (LU) or contention for synchronization, affects both protocols similarly. Also, in both cases the induced 1P-MC pattern allows LRC to amortize the cost of computing the diffs over multiple consumer requests.

### 3.5.2 Regular Applications with Induced Fragmentation

Both FFT (Figure 12) and Ocean (Figure 13) fall in this category since they have fine- or medium-grained remote read accesses. Thus, the efficiency of communication is low in both cases if performed at page granularity.

In FFT, the amount of fragmentation depends on the problem size, so increasing the problem size improves performance dramatically. The high communication volume in FFT

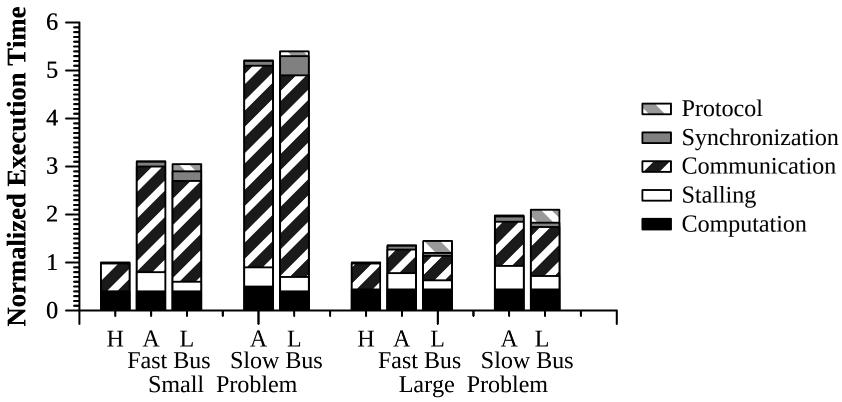



Figure 12: Results for FFT. 16 Processors

causes the bus speed to have a large impact too. Fragmentation increases when using more processors, so the speedup at 32 processors is not much higher than at 16. However, continuing to increase the problem size (Table 14) improves the speedups greatly even at 32 processors. A large enough problem would eliminate the fragmentation, at which point speedup should be comparable with CC-NUMA. AURC and LRC have similar performance. While there is additional protocol overhead due to diff computation in LRC bandwidth needs are high and overhead is compensated for by larger memory bus contention due to write-through traffic in AURC. Communication time is slightly larger in LRC due to diff application overhead.

Ocean should have similar behavior, with the difference that the fragmentation does not disappear by increasing the problem size. This is exactly what we observe for LRC although communication to computation ratio decreases substantially. However, Ocean turns out to be a performance showcase for AURC because the 1P-1C nearest-neighbor communication pattern allows the Copyset-2 optimization to avoid most of the page fetches (see Section 2.4).

This specialized protocol also allows AURC to perform dramatically better than LRC, and causes performance to scale well up to 32 processors. The problem for LRC in Ocean is that although only the boundary elements need to be communicated the diffs constitute the whole page that they fall on since the whole page, not just the boundary element, has been written by the producing processor. Thus, not only does the whole page have to be diff'd unnecessarily, but it ends up being communicated as well. The extra bus bandwidth

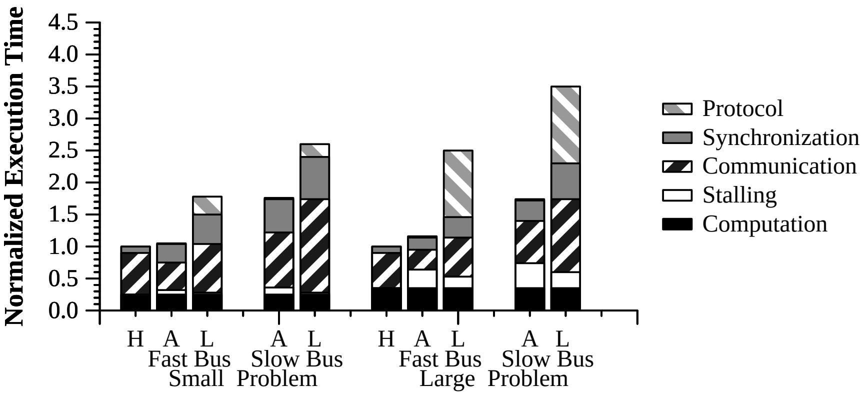



Figure 13: Results for Ocean. 16 Processors

needs of AURC are substantial for Ocean due to write-through caches for shared pages, but are overshadowed by these costs in LRC.

### 3.5.3 Irregular Applications with False Sharing and Fragmentation

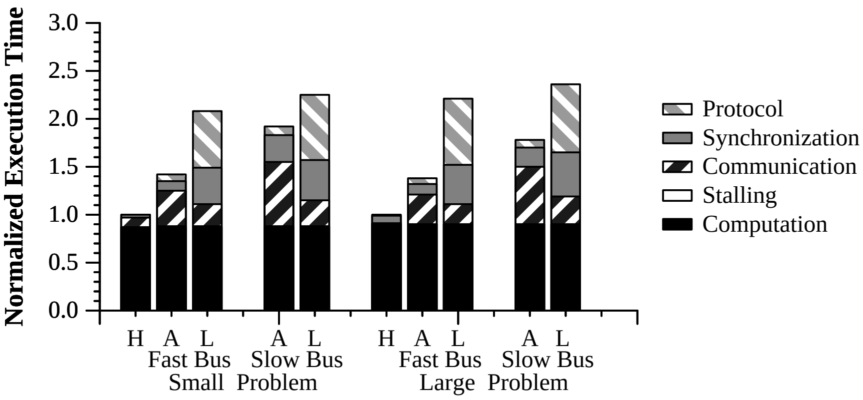



Figure 14: Results for Barnes. 16 Processors

Barnes, Cholesky, Raytrace and Volrend (Figures 14 to 17) are all irregular applications that have the undesirable MP-MC sharing pattern induced by their fine-grained read and write accesses. Therefore, all suffer substantially from false sharing as well as fragmentation. Synchronization overhead is also high because most of the synchronization is through a large number of locks. The dilating effect that SVM systems have on critical sections, discussed earlier, causes the cost and contention for these locks to be very high, even though the same locks incur very low overhead in hardware CC-NUMA. Imbalances across processors in

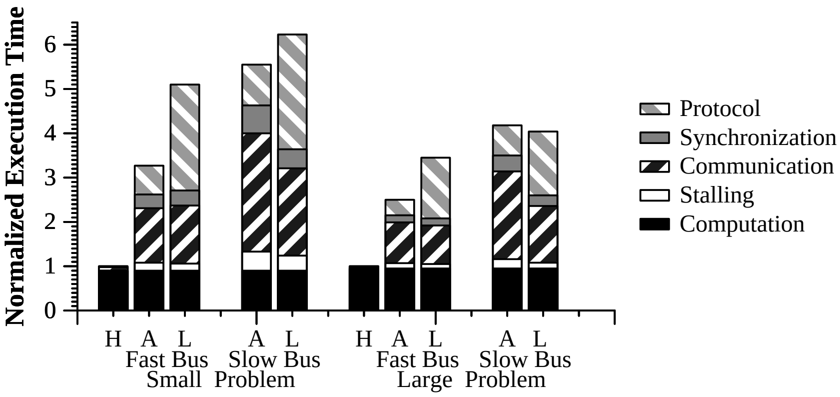



Figure 15: Results for Cholesky. 16 Processors

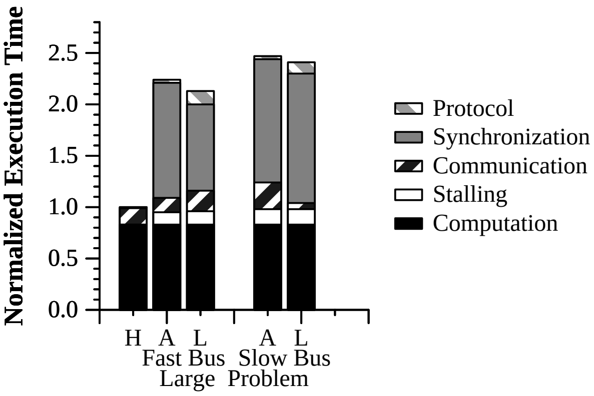



Figure 16: Results for Raytrace. 16 Processors

communication, contention and synchronization cost also amplify the overall load imbalance in these applications for SVM systems. These effects increase with the number of processors. The result is that performance can be quite limited for these irregular applications.

Speedups can be increased in some cases by increasing the problem size, but not as effectively as for regular applications. This is because in the irregular applications the accesses remain scattered when the problem size increases [WOT$^+$95b, ISL96b], and no thresholds are crossed that reduce fragmentation or false sharing. The inherent communication to computation ratio does not change so quickly either. For the MP-MC patterns the tradeoffs between the multiple-write mechanisms of LRC and AURC vary across applications.

In Barnes, the tree building phase exhibits the big difference between CC-NUMA and the SVM protocols. Unlike in hardware CC-NUMA, where it takes a small fraction of the execution time, the tree building phase becomes expensive under software shared

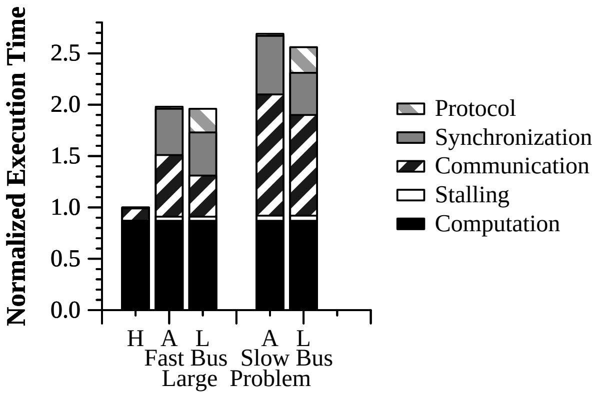



Figure 17: Results for Volrend. 16 Processors

memory (this characteristic holds for many of the irregular applications). The tree building involves a lot of very fine-grained communication protected by locks, and produces large communication and protocol overhead for very little computation. The synchronization wait time in Barnes is large mainly because of imbalances in communication and contention among processors (not in computation) caused by the accesses to the tree. This too is a much larger effect than in hardware CC-NUMA. Synchronization time is larger in LRC due to the diff overhead, which increases the cost of synchronization operations and amplifies the imbalance. Increasing the number of processors does not increase the speedup much because the relative frequency and cost of the expensive synchronization increases. Larger problem sizes help, but only slowly.

In Barnes, AURC transfers twice as much data as LRC due to the effectiveness of diffs for fine-grained sharing, but in fewer messages. In addition, the diff computation is much more expensive than the savings in bandwidth. Hence, Barnes performs much better with AURC particularly due to the tree-build phase.

In Cholesky, because accesses are largely medium-grained (half a page) rather than fine-grained, diffs are less effective in reducing traffic. The data communicated is about the same for AURC and LRC. On the other hand, the cost of diff computation is amortized better in Cholesky since the inherent pattern for the blocks is 1P-MC (similar with LU). This is why the performance gap between AURC and LRC is significant only for the fast bus case. Overall, performance is quite poor for Cholesky compared to CC-NUMA for small problems, and even degrades at 32 processors compared to 16 processors. Performance is made reasonable at 32 processors only by using much larger problems (Table 14), but is still much worse than that of hardware CC-NUMA.

For the graphics applications (Volrend and Raytrace), SVM performance is modest for two reasons. First, write-sharing is very fine-grained. Second, there is intensive lock synchronization corresponding to the task queue accesses which separates these writes. The performance of the two SVM protocols is similar. In Volrend, it is the standard tradeoff between protocol overhead and multiple messages in LRC, and extra communication in AURC. In Raytrace, synchronization and contention dominate for both systems. This is why while in Volrend the speedup increases from about 7 to about 9 when going from 16 to 32 processors, in Raytrace there is in fact an absolute slowdown at 32 processors. I was unable to run larger problems for these applications due to simulation constraints.

### 3.5.4 Applications Dominated by False Sharing

The Radix program (Figure 18) is interesting in that its induced pattern appears to be MP-1C: different processors write keys to portions of the array that only one processor will read. However, due to write-write false sharing induced among the writers after synchronization, the sharing pattern becomes MP-MC. This occurs in Radix where the producers write to the same addresses again in the next iteration. In this case, write-write false sharing turns the writers into involuntary consumers since they have to fetch the updates from the other writers even though they don't need them. This false sharing, together with an inherently large communication to computation ratio and a lot of uneven contention due to page granularity, hurts performance dramatically leading to almost no speedup.

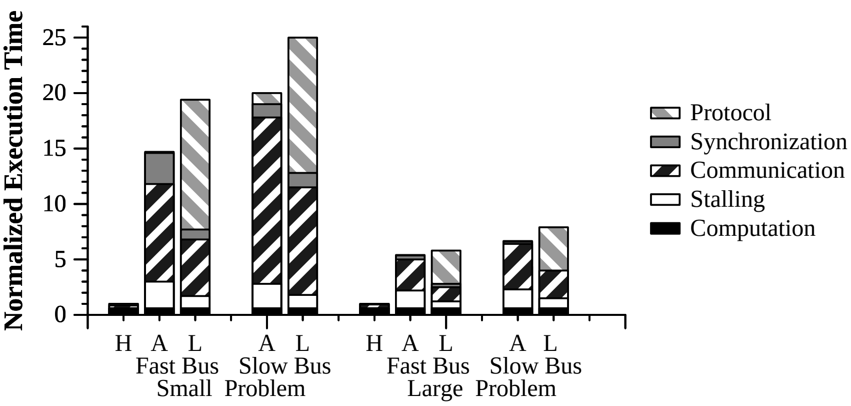



Figure 18: Results for Radix. 16 Processors

Apparently, AURC should perform much better than LRC since it supports fine-grained remote updates and automatically propagates the writes to the single consumer site. However, the above involuntary consumer effect comes into play here as well and causes many page faults. Using larger problem sizes does not help much in Radix since it only tends to increase contention due to larger traffic volume without changing the inherent communication to computation ratio. Performance is only likely to improve when the problem is large enough that the number of keys per radix value per processor ($\frac{n}{r*p}$) is much larger than a page, so the writes by a processor can be considered contiguous at page granularity and there is no false-sharing. However, the bandwidth requirements and contention are high enough in Radix to prevent it to achieve good performance under SVM.

One way to avoid the involuntary consumer effect caused by the write-write false sharing is to extend the protocol to support *write-only* pages. This extension is discussed in Section 4.2.9.

### 3.5.5 A More Realistic Cache Hierarchy

All the performance results presented so far are obtained from simulating a large one level cache. As discussed in Section 3.1, this is likely to bias the results in favor of LRC by not letting its diffs pollute the cache much. To study the behavior of the SVM protocols under more realistic assumptions, I also simulated a two level cache hierarchy with a 16K direct mapped first level cache (L1) and a 1MB two-way set associative second level cache (L2) with a 10 cycle hit cost. I perform all these experiments with the fast bus setup and the large problems from Table 12. Table 15 presents the speedups for both cache hierarchies. LRC speedups are reduced quite substantially, while the AURC speedups either don't change or increase slightly. This is because the damage to uniprocessor performance is greater than to the multiprocessor case. AURC now outperforms LRC substantially in all cases. In four out of 10 cases (Barnes, Cholesky, Ocean, Raytrace), the speedups with AURC are double of those with LRC.

### 3.5.6 Sensitivity and Robustness

In this section, I briefly examine some interesting variations in the architecture and programs that can affect our results.

**Write Buffer Size**. In all these simulations, the write buffer was 8 cache lines deep. Since deep cache buffers favor a write-through scheme like AURC, I also examine some of

| Application | L1 Cache | | L1 & L2 Cache | |
|---|---|---|---|---|
| | AURC | LRC | AURC | LRC |
| Barnes | 10.3 | 6.4 | 10.3 | 4.1 |
| Cholesky | 6 | 4.4 | 7.3 | 3.2 |
| FFT | 11 | 10.2 | 11 | 9.8 |
| LU | 13.3 | 11.8 | 13.5 | 11.3 |
| Ocean | 11.4 | 5.3 | 12.9 | 5.4 |
| Radix | 2.9 | 2.7 | 3.4 | 2.7 |
| Raytrace | 6.6 | 7 | 8.7 | 4.4 |
| Volrend | 7.1 | 7.1 | 7.4 | 5.5 |
| Water-Nsquared | 11.5 | 10.2 | 11.6 | 8.6 |
| Water-Spatial | 13 | 12 | 13.5 | 12 |

Table 15: Effect of Two Level Caches. Speedup Table for 16 Processors, Fast Bus

the applications that have a significant write stall time (Barnes, Cholesky, FFT and Ocean) using a 4 cache line deep write buffer both under LRC and AURC. While the write buffer size does affect the number of processor stalls which become 2-3x larger, the impact on total elapsed time is negligible, ranging from 2-4%.

**Contiguous versus Non-Contiguous Block Allocation**. Two of the applications, Ocean and LU, have less optimized versions provided for them in the Splash-2 suite. These "non-contiguous" versions use a more natural 2-d array representation of the 2-d grids and matrices, which does not allow us the benefit of contiguous access patterns or effective data placement across main memories, and results in substantial false sharing. These programs thus provide a good test of one form of the performance robustness of the systems. The results are catastrophic for both schemes due to false sharing, particularly for the LU program. Neither scheme is very robust to the lack of the data structuring optimization, but AURC is less robust than LRC because it is more likely to create hot spots in a home-based write collection scheme.

**Data Placement**. LRC does not benefit from the placement of pages in home memories since updates are collected in a distributed fashion everywhere. In AURC good home placement may save page faults and may also reduce data traffic if a page is assigned to a heavy writer. AURC, as all our home-based protocols, employs a first-touch policy for home assignment. This allows programmer's hints about page placement to be simple memory accesses.For AURC a sound home placement is therefore critical in reducing communication and contention. However, since data placement is also important for hardware CC-NUMA, shared address space programs often already have the necessary hints. Moreover, it is conceivable to develop an adaptive AURC which can perform dynamic home remaping, assuming the cost of memory mapping operations is not excessive. Such an adaptive AURC will greatly benefit from the automatic update traffic information which either the network interface or a specialized performance monitor card can provide [LJI+98].

## 3.6 How to Write a SVM Application

I learned a number of lessons while trying to understand and improve the performance of the applications for SVM protocols which may translate to useful guidelines for programming for SVM systems [ISL96b].

1. **Fine-Grained Allocation**. Many irregular applications have collections of logically small data structures allocated in large shared arrays; e.g particles in Barnes-Hut, rays, scene elements and task queues in Raytrace. These units are accessed in a spatially haphazard manner by processors, which leads to substantial fragmentation and false sharing. A simple way to alleviate these granularity effects in SVM systems is to allocate the small data structures separately in separate small *malloc*'s, whenever possible. This makes the units fall on separate pages, but it also allows the system to communicate the data and process protocol overhead for the data at the granularity given by the allocation size rather than the whole page. In most programs, this optimization can be easily done for many of the important and frequently written shared data structures, and I found it to be possible in almost all the applications I studied. Task queues are obvious and important candidates: instead of keeping all the queue pointers for different queues contiguously in an array while accesses and modifications are performed under distinct locks, I allocated each queue pointer separately. This simple and effective technique led up to 20% performance improvement for our applications. The potential problem with the above approach is the wasted memory since each little malloc is aligned to a page.

2. **Fine-grained Allocation versus Padding**. An alternative solution for false-sharing that is often used in hardware-coherent systems is padding, which also keeps unrelated data on distinct coherence units. However, padding is not so good in SVM. Padding to

a page does reduce false sharing, but it can cause useless data to be either communicated in AURC, or compared to compute diffs in LRC. Both padding and fine-grained allocation waste memory, but fine-grained allocation is preferable for shared virtual memory.

3. **Careful Synchronization**. Synchronization is a much more significant issue for shared virtual memory than for hardware shared memory. Since SPLASH-2 applications were written for hardware shared memory multiprocessors synchronization was employed without restriction and at relatively fine-grain. However, synchronization is much more expensive in shared virtual memory systems. It invokes the software protocol, performs consistency operations, and generates significant protocol overhead and subsequent communication. For locks, the subsequent page faults tend to greatly expand the duration of seemingly small critical sections, and hence substantially magnify serialization. Since synchronization is what causes consistency to be maintained in invalidation-based SVM protocols, it is both critical and expensive. In hardware-coherent machines synchronization operations are supported in inexpensive hardware. Programs with many locks (Radiosity, Raytrace) often work well in hardware-coherent machines but may perform poorly in SVM.

   There are many simple ways to reduce the frequency of synchronization and the contention it incurs. These techniques are often not needed in hardware-coherent machines, but are crucial in SVM. One technique is to use private local accumulators for partial per-process accumulation, and make locked updates to global variables as infrequently as possible. Reducing/eliminating critical sections that are not needed for the correct execution, like those that protect global statistics variables, is another technique. In general, algorithms that reduce locking and other synchronization, even at the expense of some extra computation, are likely to be very beneficial for SVM.

4. **Distribution of Pages to Home Memories**. In AURC, a page's home serves as the collector of automatic updates for the page, and whole pages are fetched from home on a page fault. By default, homes are selected by the protocol based on the first touch, perhaps with hints from the programmer. In LRC the home is not important since diffs are fetched from the last writers on a page fault. It does not matter where the first allocated home is . If the distribution of pages among home memories is not good in AURC, popular homes can become hot spots and contention can occur. This

can also result in major imbalances among processors and high synchronization times such as in Raytrace or Volrend. Because of the page granularity and the fact that all misses go to the home, the contention can be much greater than in hardware-coherent machines.

For regular applications, it is often relatively easy to produce a good distribution of homes for pages. But for irregular applications with fine-grained data sharing in the virtual space, this is often extremely difficult. The best which can be done in many of these cases is to hope that a round-robin distribution of pages works well or adjust the distribution when contention is detected.

5. **Modifying Rather Than Rebuilding Distributed Data Structures**. Many irregular programs use a distributed global data structure such as a tree, list, or queue. These structures are dynamic and must be adjusted periodically. The tree in Barnes-Hut is an example. Hardware-coherent programs often rebuild the data structure because tree-building is not an expensive phase for moderate-scale machines. However, this rebuilding usually requires a lot of fine-grained communication and synchronization for very little computation, and therefore quickly becomes very expensive in SVM. In many cases the rebuilding can be replaced by incremental modification of data structures. In hardware cache-coherent systems the overhead of writing incremental modification algorithms is often not worthwhile but it may be very helpful in SVM since it reduces the number of locks used. This issue was further investigated in [JSS97].

## 3.7 Summary

In this chapter, I introduce an application classification which serves as a framework to understand the performance of shared virtual memory. Based on this classification framework I analyze the simulation results of AURC and traditional LRC protocols.

The simulations indicate that SVM protocols can perform surprisingly well for several classes of applications, both regular and irregular, even up to 32-processor systems. However, SVM needs significantly larger problem sizes to obtain good parallel efficiency. Besides improving communication to computation ratio in regular applications, large problems often overcome thresholds in relationship with the coherence and communication granularity. The problem sizes used in the simulations are by no means large. The performance potential of SVM may increase for large practical problems.

The simulation results indicate that for certain regular applications AURC performs comparably with CC-NUMA for small-scale systems and outperforms the traditional LRC protocol– particularly if a realistic two-level cache hierarchy is considered. The basic tradeoff between the two schemes is that AURC potentially consumes more bandwidth, due to its write-through nature for shared data, while LRC has much greater end-point communication overhead due to diff processing. The results also suggest that the bandwidth needs are likely to be met more easily in modern systems and will be a less significant bottleneck than the diff processing overhead.

Because of its high software protocol overheads, SVM is sensitive to how data structures and access patterns interact with the page granularity. The sharing patterns induced by this granularity can be very different than the inherent sharing patterns in the application, which can have a substantial impact on performance. However, certain techniques like contiguous allocation of related data or separate memory allocations for logical objects can be used to effectively preserve simple sharing patterns like 1P-MC. I also found lock-based synchronization to be much more expensive and contentious in SVM than in CC-NUMA, particularly in the irregular applications, since the page faults that occurs within the critical sections greatly dilate the critical sections. In general, contention is a much more substantial issue, and induces high communication costs and load imbalances in many applications.

The simulation results indicate that it is possible to substantially improve the performance of SVM by following some additional programming guidelines and techniques, several of which I have identified. More research will likely identify more such techniques which may require substantial program changes from the way it is written adequately for cache-line based hardware coherence, i.e. assuming small coherence units, and relatively cheap fine-grained communication and synchronization. Further research may establish SVM as a good platform for a wider range of irregular applications, together with a set of programming guidelines, and proceed further towards identifying the classes of emerging irregular applications for which this attractive, inexpensive approach is appropriate.

The simulation-based evaluation of AURC and traditional LRC protocols leaves several questions unanswered: (i) how do these protocols perform on real systems; (ii) how do they scale beyond 32 processors; (iii) how much is the contribution of home-based schemes; and (iv) how-much does automatic update contribute to the performance of AURC. To answer these questions we implemented these protocols on real systems and will present these experiments in the next chapter.

# Chapter 4

# Implementation and Experiments

In this section, I evaluate the performance potential of the home-based approach on real systems, from two angles: scalability, and the impact of memory-mapped communication and automatic update.

The results confirm that HLRC scales significantly better than the traditional LRC in terms of overall performance, communication and memory overhead. They also show that memory mapped communication is an efficient support for home-based SVM protocols. Moreover, when it is catalyzed by an adequate protocol design, the automatic update support can significantly accelerate a home-based shared virtual memory system, particularly for applications with write-write false sharing.

## 4.1 Scalability

Due to its simplicity in design, low memory, communication and protocol overhead, the home-based approach promises to scale better than the traditional LRC approach. To verify this assumption, we evaluated the HLRC and traditional LRC protocols, plus the corresponding overlapped protocols (OLRC and OHLRC) on a 64-node Intel Paragon using a subset of Splash-2 benchmark programs.

### 4.1.1 The Intel Paragon Multicomputer

The Intel Paragon multicomputer used in our implementation consists of 64 nodes for computation. Each node has one compute processor and a communication co-processor, sharing 64 Mbytes of local memory. Both processors are 50 MHz i860 microprocessors with

16 Kbytes of data cache and 16 Kbytes of instruction cache [Int91a]. The data caches are coherent between the two processors. The memory bus provides a peak bandwidth of 400 MBytes/sec. The nodes are interconnected with a wormhole routed 2-D mesh network whose peak bandwidth is 200 Mbytes/sec per link [TD92].

The operating system is OSF/1 with multicomputer extensions for a parallel programming model and the NX/2 message passing primitives. The co-processor runs exclusively in kernel mode, and it is dedicated to communication. The one-way message-passing latency of a 4-byte NX/2 message on the Paragon is about 50 $\mu$sec [PR94]. The transfer bandwidth for large messages depends on data alignment. When data are aligned properly, the peak achievable bandwidth at the user level is 175 Mbytes/sec. Without proper alignment, the peak bandwidth is about 45 Mbytes/sec.

The operating system uses an 8 Kbyte page size for its virtual memory though the hardware virtual memory page size is 4 Kbytes. All implementations use the `vm_protect` Mach system call to set access protection for shared pages. Access faults can be handled by either the external memory manager or the exception handling mechanism. We used the Mach exception handling mechanism for efficiency reasons.

Both LRC and HLRC protocols are implemented exclusively at user-level using the NX/2 message library.

### 4.1.2 Protocol Implementation

Synchronization handling and related coherence operations are implemented at user level using NX/2 messages. Each lock has a manager which is assigned in a round-robin fashion among the nodes. The manager keeps track of which processor has most recently requested the lock. All lock acquire requests are sent to the manager unless the node itself holds the lock. The manager forwards the lock request to the processor that last requested the lock. The request message contains the current maximal vector timestamp of the acquiring processor. When the lock arrives, it contains the releaser's knowledge of all time intervals for the requester to update its timestamp vectors.

Barriers are also implemented using a centralized manager algorithm. When a barrier is entered, each node sends the barrier manager the write notices for all intervals that the manager has not seen. The barrier manager collects the intervals from all other nodes, computes the maximal timestamp, and selectively forwards the missing write notices to each node.

| Operation | Time in microseconds |
|---|---|
| Message Latency | 50 |
| Page Transfer | 92 |
| Receive Interrupt | 690 |
| Twin Copy | 120 |
| Diff Creation | 380-560 |
| Diff Application | 0-430 |
| Page Fault | 290 |
| Page Invalidation | 200 |
| Page Protection | 50 |

Table 16: Basic Operation Costs on the Intel Paragon

Similar to the approach used in TreadMarks [KCDZ94], barrier synchronization in LRC triggers garbage collection of protocol data structures when the free memory is below a certain threshold. Garbage collection is quite complex because it needs to collect all "live" diffs which are distributed on various nodes. All last writers for each individual shared page need to validate the page by requesting all the missing diffs from other nodes. The non-last writers can simply invalidate the page and modify the copyset for that page. After this phase, the collection algorithm can clean up the heaps and data structures.

For the HLRC there is no need to perform garbage collection since no diffs or write notices are ever stored beyond a release or barrier.

### 4.1.3 Basic Operation Costs

Table 16 shows the costs of basic operations on the Intel Paragon.

Using the basic operation costs we can determine the minimum cost for a page miss and a lock acquire assuming no contention. In HLRC, a page miss takes at least 1,172 microseconds for a full page transfer. Similarly, a page miss in LRC takes at least 1,130 microseconds. A remote acquire request, which is intermediated by the home of the lock, costs at least 1,550 microseconds.

### 4.1.4 Applications

To evaluate the performance of our implementations, we used two kernels (LU and SOR) and three applications (Water-Nsquared, Water-Spatial and Raytrace), four of which (LU, Water-Nsquared, Water-Spatial and Raytrace) are from the Splash-2 suite.

| Application | Problem Size | Sequential Execution Time (secs) |
|---|---|---|
| LU | $2048 \times 2048$ | 1,028 |
| SOR | $1024 \times 4096$ | 1,360 |
| Water-Nsquared | 4096 molecules | 1,130 |
| Water-Spatial | 4096 molecules | 1,080 |
| Raytrace | Balls4.env(256x256) | 956 |

Table 17: Applications, problem sizes, and sequential execution times.

Table 17 shows the problem sizes and their sequential execution times. For all applications we chose relatively large problem sizes, each requiring approximately 20 minutes of sequential execution. Problem sizes were determined by the capabilities of our four prototypes: although the home-based protocols can run larger problems, we chose the largest problems runnable under all protocols and all machine sizes for the sake of comparison.

| | 8 nodes | | 32 nodes | | 64 nodes | |
|---|---|---|---|---|---|---|
| Application | LRC | HLRC | LRC | HLRC | LRC | HLRC |
| LU | 4.4 | 5.2 | 11.5 | 13.9 | 15.5 | 27.0 |
| SOR | 4.3 | 6.4 | 13.0 | 22.7 | 12.3 | 35.7 |
| Water-Nsquared | 6.7 | 7.0 | 11.7 | 18.9 | | 19.2 |
| Water-Spatial | 7.4 | 7.4 | 14.1 | 20 | 10.6 | 22.6 |
| Raytrace | 6.9 | 7.6 | 10.6 | 26.8 | 7.4 | 40.6 |

Table 18: Speedups for LRC and HLRC on the Intel Paragon with 8, 32 and 64 nodes.

### 4.1.5 Overall Performance

Table 18 summarizes the speedups for LRC and HLRC on 8, 32 and 64 nodes. There are two key observations to be made here. First, the home-based LRC protocol clearly outperforms

| | 8 nodes | | 32 nodes | | 64 nodes | |
|---|---|---|---|---|---|---|
| Application | OLRC | OHLRC | OLRC | OHLRC | OLRC | OHLRC |
| LU | 4.7 | 6.0 | 13.5 | 16.6 | 16.8 | 29.1 |
| SOR | 4.8 | 6.5 | 13.6 | 23.6 | 12.6 | 36.9 |
| Water-Nsquared | 7.0 | 7.1 | 14.0 | 21.0 | – | 20.6 |
| Water-Spatial | 7.5 | 7.7 | 17.1 | 23.5 | 11.5 | 26.4 |
| Raytrace | 7.1 | 7.7 | 10.6 | 28.0 | 7.4 | 43.0 |

Table 19: Speedups for OLRC and OHLRC on the Intel Paragon with 8, 32 and 64 nodes.

the no-home or "homeless" traditional LRC counterpart except for Water-Spatial on 8-node where speedups are comparable. Second, the performance gap between home and no-home protocols increases dramatically for 32- and 64-node configurations. This result, which is consistent across all applications, reveals a significant difference in scalability between the two classes of protocols. For instance, for 64 nodes HLRC outperforms LRC by a factor of 1.7 for LU, a factor of 2 for Water Spatial, a factor of 3 for SOR and a factor of almost 6 for Raytrace. For Water Spatial and Raytrace the speedups under the LRC protocol actually drop when going from 32 to 64 nodes. Obviously, such insights would have been impossible to guess from the 8-processor runs where the performance of the home-based and no-home protocols are very close.

The overlapped protocols (Table 19) provide modest improvements over the non-overlapped ones for the applications we ran. The range of speedup improvements varies among applications, from as little as 2-3% to as much as 30%.

Summarizing, given the limitations of the Paragon architecture in terms of message latency and interrupt cost all five real Splash-2 applications perform quite well under the home-based protocols, with more than 50% parallel efficiency on 32 nodes, and between 30% and 66% on 64 nodes.

### 4.1.6 Home Effect

Table 20 shows the frequency of page faults and diff related operations for HLRC and LRC on 8 and 64 nodes. There are several "home effects" revealed by this table. First, since the home's copy of the page is eagerly updated, the home node has no page fault nor diff creation. This can lead to substantial protocol savings, particularly if there is a one-writer, multiple-reader sharing pattern and the writer is chosen as the home. This explains why

| Application | Number of nodes | Read misses | | Diffs Created | | Diffs Applied | |
|---|---|---|---|---|---|---|---|
| | | LRC | HLRC | LRC | HLRC | LRC | HLRC |
| LU | 8 | 1,050 | 1,050 | 5,704 | 0 | 0 | 0 |
| | 64 | 452 | 452 | 1,396 | 0 | 0 | 0 |
| SOR | 8 | 343 | 343 | 25,088 | 0 | 171 | 0 |
| | 64 | 385 | 385 | 3136 | 0 | 342 | 0 |
| Water Nsquared | 8 | 938 | 811 | 1,266 | 810 | 4,809 | 810 |
| | 32 | 893 | 859 | 956 | 815 | 8,742 | 815 |
| Water Spatial | 8 | 1,361 | 1,356 | 466 | 10 | 74 | 10 |
| | 64 | 918 | 878 | 122 | 66 | 3,948 | 66 |
| Raytrace | 8 | 339 | 209 | 551 | 733 | 3,591 | 733 |
| | 64 | 108 | 87 | 83 | 103 | 2,368 | 103 |

Table 20: Average number of operations on each node.

LU and SOR have created no diffs. The other three applications have reduced page faults and diffs due to the home effect.

Finally, the HLRC protocol has fewer diff applications than the traditional LRC protocol because the HLRC protocol applies diffs eagerly but only once whereas the LRC protocol may apply diffs multiple times.

### 4.1.7 Detailed Performance Analysis

To better understand where the time goes, the protocols were instrumented to collect the average execution time breakdowns per node. Figure 19 shows the breakdowns, including the computation time, data transfer time, garbage collection time, synchronization time including locks and barriers, and protocol overhead. Protocol overhead includes diff and twin creation, diff application, write-notice handling, and remote request service. Table 21 shows the number of synchronization operations for each application.

We use Water-Nsquared (Figure 19(c)) as an example to introduce the time breakdown analysis. The time breakdowns show that the differences in performance between HLRC and LRC are due to the reduction of synchronization time, data transfer time, and protocol overhead.

| Application | Number of nodes | Lock Acquires | Barriers |
|---|---|---|---|
| LU | 8 | 0 | 64 |
| | 64 | 0 | 128 |
| SOR | 8 | 0 | 98 |
| | 64 | 0 | 98 |
| Water Nsquared | 8 | 652 | 12 |
| | 32 | 556 | 12 |
| Water Spatial | 8 | 11 | 10 |
| | 64 | 11 | 10 |
| Raytrace | 8 | 340 | 1 |
| | 64 | 94 | 1 |

Table 21: Number of synchronization operations on each node

Synchronization cost dominates the total overhead. To identify the sources of the execution imbalance among nodes, we instrumented all applications to collect per processor execution time breakdowns between two consecutive barriers. Figure 20 shows per-processor breakdowns of the execution time for Water-Nsquared between barriers 9 and 10 for both LRC and HLRC protocols on 8 and 64 nodes. For 8 nodes the imbalance is small and mostly due to computation imbalance. For 64 nodes the computation time is evenly distributed among nodes; almost all the imbalance is due to lock contention and communication imbalance.

Lock waiting time is significant because page misses dilate the critical sections (see Section 2.5). Therefore variations in the data transfer time are reflected in the lock waiting time. For instance, in Water-Nsquared the lock waiting time is larger under LRC than under HLRC because the data transfer time is larger as well. Lock waiting time can cause execution imbalance due to serialization of lock acquisitions when lock contention occurs. Water-Nsquared is a regular application; lock contention occurs when an imbalance in data transfer time occurs. This explains why there is an imbalance in the lock waiting time for LRC but not for HLRC. For irregular applications, such as Water Spatial, imbalance in the lock waiting time occurs even when data transfer time is balanced.

The second dominating overhead is data transfer time. The data transfer times of HLRC are smaller than those of LRC for two reasons. For irregular applications, HLRC sends fewer

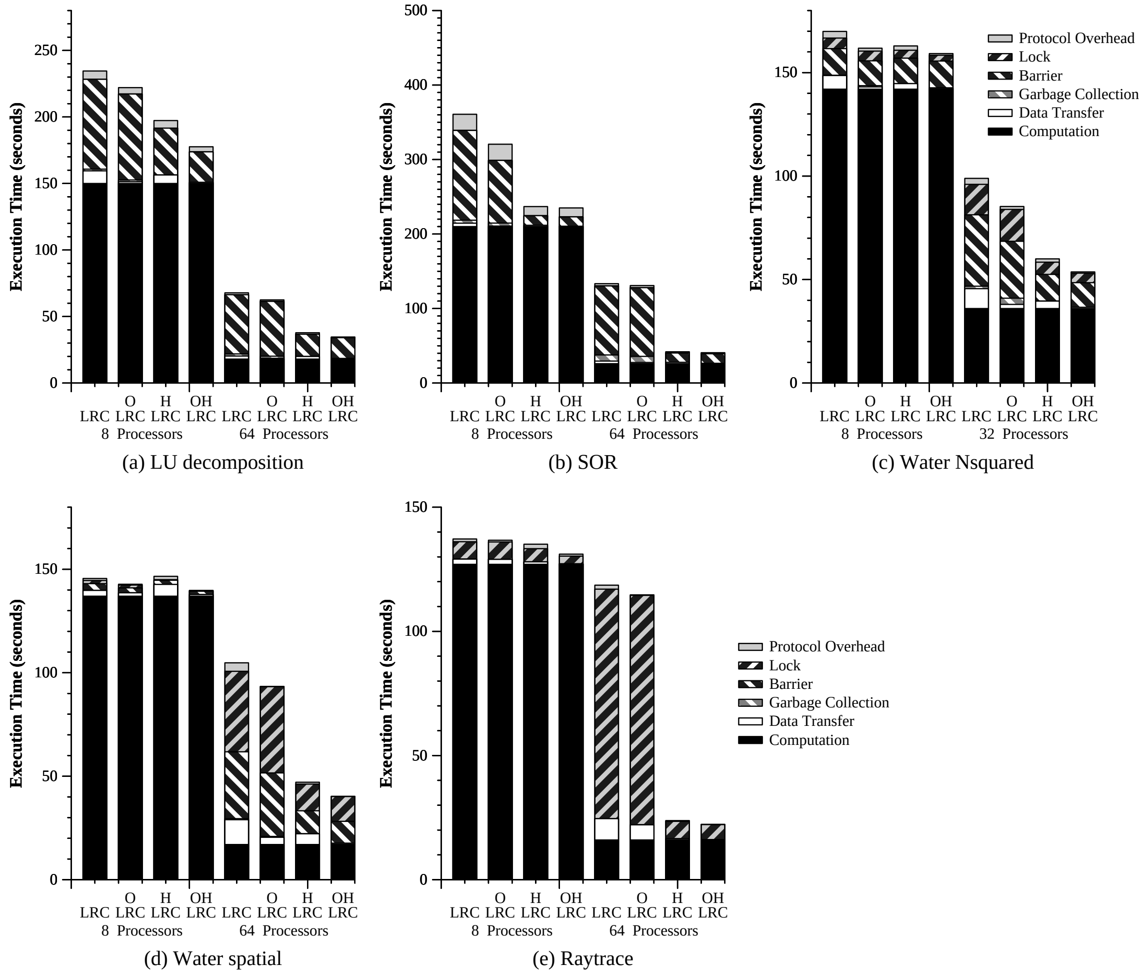



Figure 19: Time breakdowns of LU, SOR, Water-Nsquared,Water-Spatial and Raytrace.

messages than LRC and the difference increases with the number of nodes. On a system like Intel Paragon where message overhead is large, the difference in the number of messages translates into a significant difference in the data transfer time. For regular applications with coarse-grain sharing and migratory data patterns [ISL96a], such as Water-Nsquared the data traffic to service page misses is higher for the LRC protocol than for the HLRC protocol. This is counter-intuitive since LRC transfers diffs whereas HLRC always transfers full pages. However, in LRC migratory data patterns can cause diff accumulation with aggregate sizes significantly larger than a page.

By offloading some of the protocol overhead from the critical path to the communication processors, the overlapped protocols reduce data transfer time, protocol overhead, and

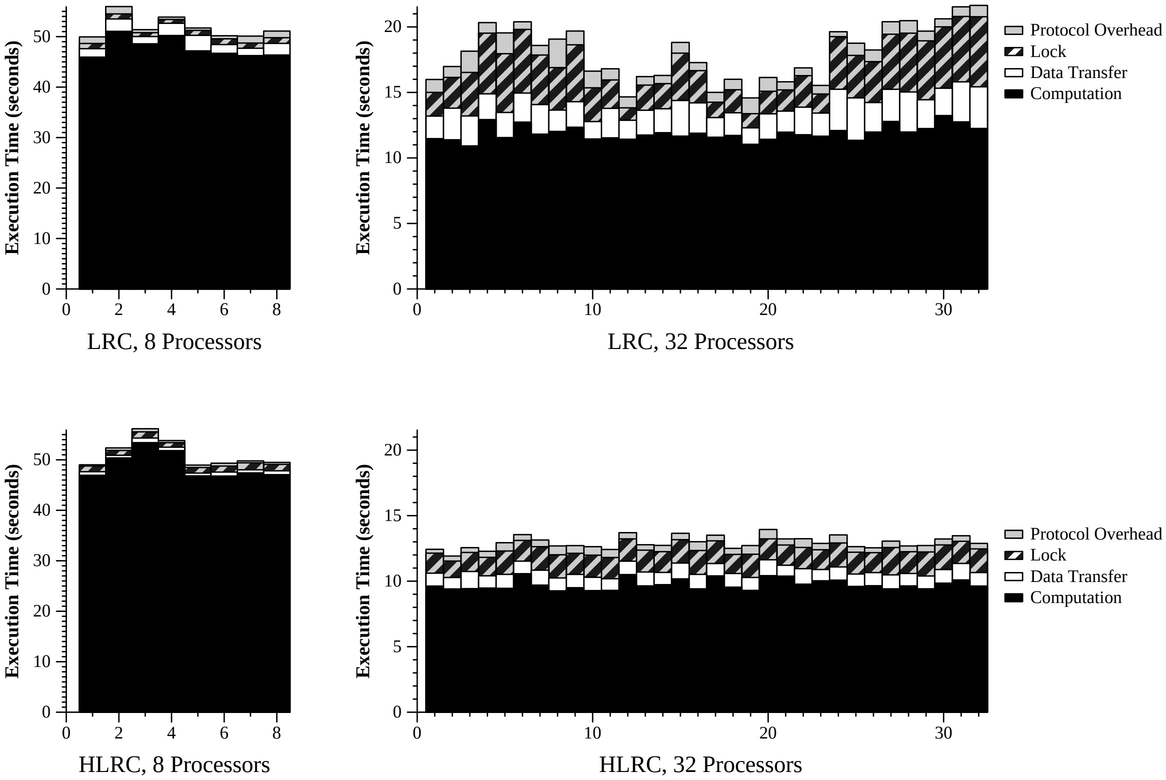



Figure 20: Time breakdowns of Water-Nsquared between barriers 9 and 10.

synchronization time. Data transfer time is reduced because the remote fetch requests are handled on the communication processor. Synchronization cost is reduced slightly because overlapping does not change the execution imbalance among the nodes. Protocol time is always reduced with overlapping, but its relative contribution to the total execution time is small.

The performance analysis for Water-Nsquared holds for the other applications as well. The 8-processor run of Water-Spatial reveals a case in which HLRC performs slightly worse than LRC. Although Water-Spatial induces a multiple-writer sharing pattern at page level [ISL96a], for a small number of steps (two in our case) most pages are single-writer, so LRC and HLRC are similar in terms of data traffic. However, the message-size limitation of the Paragon forces HLRC to send more physical messages thus increasing the data transfer time compared to LRC.

For all applications, the synchronization cost is the main component that limits the performance efficiency for large configurations. For all five applications we have evaluated in this paper, the home-based protocols scale better on Paragon than their no-home counterparts because of a lower communication cost. Other overheads such as communication traffic and memory requirements also affect the scalability of shared virtual memory systems.

| Application | Nodes | Update traffic | | | | Protocol traffic | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | Number of messages | | Message traffic (Mbytes) | | Number of messages | | Message traffic (Mbytes) | |
| | | LRC | HLRC | LRC | HLRC | LRC | HLRC | LRC | HLRC |
| LU | 8 | 1,050 | 1,050 | 8.6 | 8.6 | 1,178 | 1,178 | 0.28 | 0.16 |
| | 64 | 452 | 452 | 3.7 | 3.7 | 708 | 708 | 1.7 | 0.6 |
| SOR | 8 | 343 | 343 | 4.2 | 2.8 | 538 | 538 | 0.95 | 0.76 |
| | 64 | 385 | 385 | 6.0 | 3.2 | 581 | 581 | 2.6 | 0.9 |
| Water Nsquared | 8 | 1,102 | 1,621 | 16.1 | 6.8 | 2,784 | 2,531 | 1.8 | 0.8 |
| | 32 | 1,421 | 1,674 | 11.5 | 7.1 | 3,093 | 2,493 | 6.7 | 1.2 |
| Water Spatial | 8 | 1,360 | 1,366 | 7.3 | 7.3 | 1,411 | 1,403 | 0.06 | 0.03 |
| | 64 | 2,581 | 944 | 16.4 | 4.7 | 2,628 | 925 | 0.42 | 0.08 |
| Raytrace | 8 | 1,069 | 942 | 1.46 | 1.43 | 1,446 | 586 | 0.44 | 0.22 |
| | 64 | 2,368 | 190 | 0.10 | 0.61 | 403 | 313 | 0.52 | 0.20 |

Table 22: Average communication traffic on each node.

### 4.1.8 Communication Traffic

Table 22 shows the communication traffic of the LRC and HLRC prototypes for all applications. The traffic information gives us several insights that substantiate our time breakdown analysis. Since the overlapped protocols have approximately the same communication traffic as the non-overlapped ones, we compare only HLRC with LRC.

For each created diff, HLRC always sends one message to update the home. In addition, one roundtrip is always enough in HLRC to satisfy any page miss. On the other hand, a no-home LRC protocol requires at least one message per page miss. It requires exactly one when there is only one writer per page or when the page contains migratory data and all the diffs are found at the last writer. If there is only one writer per page, as in SOR, the two protocols send about the same number of messages unless LRC is penalized by the garbage collection process, such as in LU, and requires additional messages to fetch pages. In a regular migratory data application, such as Water-Nsquared, HLRC ends up sending more messages than LRC because of the additional messages sent to update the homes eagerly. In the common case, which corresponds to multiple reader and writer pages such as in Raytrace, LRC sends more messages than HLRC because it requires more than one diff message to satisfy a page miss.

For applications with predominantly single-writer pages, such as LU and Water-Spatial, the amount of update-related traffic in HLRC and LRC is similar because diff accumulation does not occur in LRC. Although SOR also exposes single-writer pages, LRC has more data traffic than HLRC due to garbage collection. For applications with multiple-writer pages, such as Water-Nsquared or Water-Spatial the data traffic on 64 nodes under HLRC is substantially less than under LRC.

For applications with fine-grain sharing, such as Raytrace, the comparison moves towards what we expected to be the typical difference between HLRC and LRC: more messages in LRC and more traffic in HLRC. This leads to a latency vs. bandwidth tradeoff in the comparison of the two protocols with respect to data transfer time. For instance, systems, like the Paragon, which have relatively high message latency and high bandwidth are likely to benefit more from the home-based protocols.

Finally, for the protocol traffic, the HLRC approach is consistently cheaper than the traditional LRC protocol. The LRC protocol sends more protocol related data than HLRC, especially for large number of nodes where write notices can become substantially larger due to full vector timestamp inclusion.

In conclusion, the HLRC protocol scales better than the traditional LRC protocol in terms of protocol traffic.

### 4.1.9 Memory Requirements

Protocol memory requirement is an important criterion for scalability. It directly affects application performance in two ways: the limitation on the problem sizes and the frequency of garbage collection of protocol related data.

Table 23 reveals some interesting facts. If a garbage collection is triggered only at barriers, then the memory requirement of the LRC protocol can be even larger than the application memory (by a factor of 8 in Water-Nsquared). On the other hand, the memory used by the HLRC protocol is only a small percentage of the application memory (10% to 25%). As we increase the number of nodes, the memory required for protocol data does not change much in HLRC, but in LRC it can increase dramatically (Water-Spatial and

| Application | Number of nodes | Application Shared Data (Mbytes) | Protocol Data (Mbytes) | |
|---|---|---|---|---|
| | | | LRC | HLRC |
| LU | 8 | 32 | 10.0 | 2.8 |
| | 64 | 32 | 7.8 | 2.8 |
| SOR | 8 | 33 | 14.3 | 2.6 |
| | 64 | 33 | 9.8 | 2.6 |
| Water Nsquared | 8 | 2.5 | 20.0 | 0.54 |
| | 32 | 2.5 | 13.6 | 0.45 |
| Water Spatial | 8 | 4.8 | 9.5 | 0.9 |
| | 64 | 4.8 | 13.0 | 0.5 |
| Raytrace | 8 | 10.2 | 2.1 | 1.1 |
| | 64 | 10.2 | 15.2 | 0.9 |

Table 23: Memory requirements for application and protocol on each node.

Raytrace). Although it is possible to reorganize the data structures to reduce the memory consumption, I do not expect the overall picture to change dramatically.

Out of the various protocol data structures in Table 24, the major memory consumers are the diffs and the write notices. In HLRC, diffs are discarded almost immediately after they are created or applied, while in LRC they have to be kept until the next garbage collection time in most cases. Write notices can consume a large amount of memory. For instance, the LRC protocol consumes about 5 Mbytes of memory on each of the 32 nodes in order to store write notices in the Water-Nsquared application. Since the write-notice data structure includes the full vector timestamps in the LRC protocol, the storage requirement increases proportionally with the number of nodes.

In short, the HLRC protocol scales much better than the LRC protocol with respect to memory requirements.

### 4.1.10 Limitations

An interesting question is whether the HLRC protocol is always better than a traditional LRC protocol. The performance evaluation we have conducted on the Paragon provides limited answers towards a complete understanding of the tradeoffs between HLRC and LRC protocols. The Paragon architecture is characterized by message latency, page fault,

| Application | Number of nodes | Protocol data structure memory used (Mbytes) | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | Twins | | Diffs | | Page table | | Write notices | |
| | | LRC | HLRC | LRC | HLRC | LRC | HLRC | LRC | HLRC |
| LU | 8 | 4.2 | 0 | 4.4 | 0 | 1.3 | 2.6 | 0.1 | 0.1 |
| | 64 | 2.5 | 0 | 2.7 | 0 | 1.3 | 2.6 | 1.3 | 0.2 |
| SOR | 8 | 4.2 | 0 | 8.4 | 0 | 1.3 | 2.6 | 0.13 | 0.008 |
| | 64 | 2.3 | 0 | 5.2 | 0 | 1.3 | 2.6 | 0.9 | 0.009 |
| Water Nsquared | 8 | 7.3 | 0.3 | 10.8 | 0 | 0.1 | 0.2 | 1.8 | 0.04 |
| | 32 | 3.9 | 0.1 | 4.2 | 0 | 0.1 | 0.2 | 5.4 | 0.15 |
| Water Spatial | 8 | 2.8 | 0.5 | 6.4 | 0 | 0.2 | 0.4 | 0.12 | 0.003 |
| | 64 | 0.4 | 0.1 | 12.0 | 0 | 0.2 | 0.4 | 0.4 | 0.01 |
| Raytrace | 8 | 0.4 | 0.4 | 0.6 | 0 | 0.3 | 0.6 | 0.8 | 0.1 |
| | 64 | 0.7 | 0.1 | 13 | 0 | 0.3 | 0.7 | 1.3 | 0.1 |

Table 24: Breakdown of memory requirements for protocol data on each node.

and interrupt times that are relatively large compared with memory and network bandwidth (Table 16). As a consequence, a roundtrip communication for either a page or lock transfer is, at best, on the order of a millisecond. Current network technologies [BLA+94, BDFL96, GCP96, BCF+95], as well as aggressive software for fast interrupts, exceptions [TL94] and virtual memory mapped communication [DIFL96, FAB+96] have brought such latencies down significantly, to the neighborhood of a couple of microseconds. An interesting question is to what extent our results are specific to the Paragon architecture and how they would be affected by different architectural parameters. Fast interrupts and low latency messages make the roundtrips shorter and since LRC usually has more messages than HLRC it is likely that the LRC protocol will benefit more from these architectural improvements. Therefore, the performance gap between the HLRC and the LRC protocols would probably be smaller on such architectures.

This performance evaluation shows that the HLRC protocol performs much better than the traditional LRC for three regular applications with coarse-grain sharing and for two irregular applications with small communication to computation ratios. The traffic ratio for the two classes of protocols will probably be different if fine-grain sharing dominates. To help answer this question we ran SOR with all elements 0, except at the edges. Under this initialization, the interior elements of the matrix do not change for many iterations. Consequently, these pages have no updates during those intervals, and thus do not produce diffs. Even when diffs are produced later in the computation, there is only a single writer per page, a single diff produced per interval, and the diff size increases gradually. Consequently, the conditions of this experiment favor LRC over HLRC since HLRC must transfer full pages. Despite these factors, the experimental results showed that HLRC is still better than LRC.

In addition to the Paragon communication parameters there are several specific limitations in these experiments. The virtual memory page size used in the OSF-1 operating system is 8 Kbytes, although the hardware allows 4 Kbyte page size. We have not been able to modify the virtual memory system to change the page size in order to conduct more experiments. Asynchronous receives are not interruptible. The Paragon NX/2 message layer cannot support message packets larger than the page size. This limitation can affect both HLRC and LRC. HLRC has to always send two messages in reply to a page fetch request: one with the page and the other with the timestamps of the page. In LRC when the aggregate diff size exceeds the page size, the number of actual messages sent is larger than the number of logical messages.

Several possible optimizations are still unexplored. Synchronization operations have been implemented in a straightforward way using NX/2 messages and centralized algorithms. Protocol overlapping using the communication processor was not employed for coherence and synchronization control. Finally, we have implemented a traditional LRC protocol with no advanced optimizations which were later proposed [ACDZ97, Kel98], and we evaluated only five applications for a single problem size.

## 4.2 Memory-Mapped Communication

Virtual-memory mapped communication(VMMC) [DIFL96] is a communication model that allows protected, user-level data transfers with latency and bandwidth close to those supported by the underlying hardware. Under VMMC, protection control is separated from data transfer and the operating system involvement is eliminated in the common case. Communication is protected because it may take place only after the receiver gives the sender permission to transfer data to a given area of the receiver's address space. The receiving process expresses this permission by *exporting* areas of its address space as receive buffers where it is willing to accept incoming data. A sending process must *import* remote buffers which it will use as destinations for transferred data. An exporter can restrict possible importers of a buffer; VMMC enforces the restrictions when an import is attempted. After a successful import, the sender can transfer data from its virtual memory into the imported receive buffer. VMMC makes sure that transferred data do not overwrite receiver's addresses outside the destination receive buffer.

VMMC supports two data transfer modes: *deliberate update* and *automatic update*. Deliberate update is an explicitly initiated transfer of a contiguous block of data from any (readable) virtual address in the caller's process to a previously imported receive buffer. VMMC guarantees in-order, reliable delivery of deliberate update messages.

Automatic update is performed implicitly by the system on each write to local memory. To use automatic update, a sender must first create an *automatic update mapping*. This designates a region of the sender's virtual memory as an automatic update area and "maps" it to a remote receive buffer. Once the automatic update mapping exists, all writes by the sender to its automatic update area are automatically propagated to the remote receive buffer. Since writes are propagated automatically, there is no way to explicitly specify a destination for each write. The VMMC subsystem infers the destination from the local address written to. As a result, there may be only one automatic update destination associated with a given local address. VMMC guarantees in-order, reliable delivery of automatic update messages.

When a message sent by either transfer mode arrives at its destination, it is transferred directly into the memory of the receiving process, without interrupting the receiver's CPU. Thus there is no explicit receive operation in VMMC.

In VMMC, buffer management becomes the responsibility of the application since no kernel buffers are involved. Zero-copy protocols are possible because a direct application-to-application transfer of data can occur. The CPU overhead to send data can be very small: one local write for automatic update, or a few user-level instructions for deliberate update.

The VMMC model can be implemented in various ways depending on the amount of hardware and/or software support which can be added to the system. One approach is a custom-designed network interface which supports both user-level DMA transfer and fine-grain remote writes. An alternative approach is to implement VMMC using commodity but programmable network interfaces. In the Scalable, High Performance Really Inexpensive Multicomputer (SHRIMP) project at Princeton [BLA+94, BDFL96, FAB+96, BAB+98], aiming to build a PC-based parallel system, we took both approaches. First, a custom network interface was designed. The SHRIMP network interface [BLA+94] is a custom-designed network interface. It can snoop local writes off the memory bus and generate automatic updates to a remote virtual memory. In addition, the network interface support deliberate updates, which are explicit user memory to user memory DMA transfers. Second, we used Myrinet network interface to implement VMMC [DIFL96].

In this section, I investigate the potential of VMMC in building efficient home-based protocols. One idea is to take advantage of memory-mapped communication to eliminate memory-to-memory copy operations in a home-based protocol. Further, the automatic update (AU) mechanism can be used to accelerate a home-based protocol. Finally, an AU-based write collection scheme is used in the Automatic Update Release Consistency (AURC) protocol. Unlike HLRC, in which updates are detected and propagated in software, in AURC automatic updates can be used to transparently propagate writes performed to the local copy of the page to the home copy.

### 4.2.1 The SHRIMP Multicomputer

The SHRIMP multicomputer [BAB+98] system consists of 16 Pentium PC nodes connected by an Intel Paragon routing network [TD92, Int91b]. Each PC uses an Intel Pentium Xpress motherboard [Int93] that holds a 60 MHz Pentium CPU, 256 Kbytes of level 2 cache, and 40 Mbytes of DRAM memory. Peripherals are connected to the system through the EISA expansion bus [BCP92]. Main memory data can be cached by the CPU as write-through or write-back on a per-virtual-page basis, as specified in the process page tables. The caches snoop DMA transactions and automatically invalidate the corresponding cache lines, thereby keeping the caches consistent with *all* main memory updates, including those from EISA bus masters.

The custom network interface is the key system component of the SHRIMP system: it connects each PC node to the routing backplane and implements hardware support for virtual memory-mapped communication (VMMC) [BLA+94]. The single-word update latency from one node's virtual memory to that of another node is under 5 microseconds. The data transfer bandwidth achieved is about 25 Mbytes/sec, which is close to the PC's I/O bus bandwidth. The basic architecture of a SHRIMP node is shown in Figure 21.

The network interface supports the VMMC abstraction in two ways [FAB+96]. First, it interface supports *deliberate update* (DU) operations. An explicit send instruction is

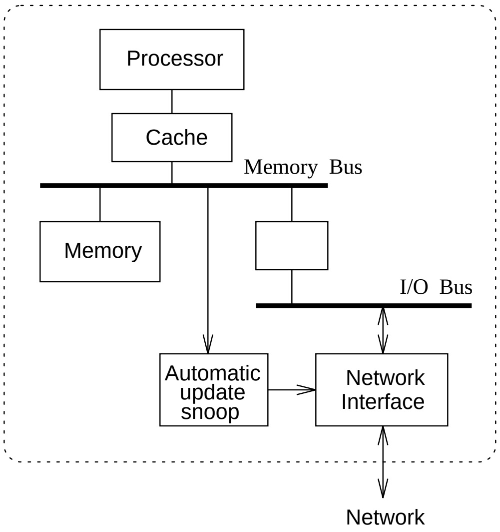



Figure 21: Architecture of SHRIMP Node.

required for DU, but a user-level DMA mechanism [BLA+94] reduces the send overhead to a few instructions.

Second, the SHRIMP network interface provides an automatic update mechanism that allows memory writes snooped from the Xpress memory bus to be propagated to the physical memory of a remote node, according to an import-export mapping established by the sender and receiver. Pages with outgoing automatic update mappings use write-through caching, so that every write to such a page causes a transaction on the memory bus. On seeing a write, the network interface hardware checks to see whether the page being written has an automatic update mapping; if it does, the network interface automatically creates a network packet and sends it to the correct destination. The network interface also combines automatic updates to consecutive addresses into a single packet to reduce network traffic.

In addition, the network interface supports a notification mechanism that uses fast interrupts to notify a receiving process that data has arrived if the receiving process has requested notifications for a particular mapping.

### 4.2.2 Protocol Implementation

Home-based protocols are well-suited for VMMC for several reasons. The cost of page fault in home-based protocols can be significantly reduced by using VMMC because the home can transfer the page directly into the physical page to which the faulting page is mapped with no copies or additional overhead. Similarly, protocol data such as write-notices or flush timestamps can be transferred directly from the sender data structure to the receiver data structure without packing/unpacking.

Automatic update opens additional opportunities to improve the efficiency of a home-based SVM protocol [IBD+98]. One way to utilize hardware AU support in an HLRC protocol is to use the AU mechanism to map the page holding the diffs to the home page. As the protocol computes the diffs at a release point, the memory writes that create the diffs are propagated to the home page automatically. This method allows the protocol to avoid buffering diffs and sending them to the home in an explicit message. A more promising approach is to incorporate the AU support into the protocol design as it occurs in AURC. In addition, AU support is used in AURC to propagate the flush timestamps from the non-home to the home nodes at release.

The relative performance of AURC and HLRC represents a high-level tradeoff between potentially higher bandwidth needs in AURC, due to multiple writes to the same word being propagated multiple times, and higher end-point overhead in HLRC, due to diff computation and bursty propagation.

The performance evaluation consists of three parts. First, I examine the impact of memory-mapped communication which allows pages to be transferred without the copy-on-receive required in SVM implementations based on sockets or other standard message passing libraries.

Second, I evaluate the impact of using automatic update support simply to accelerate the communication in HLRC. For this purpose, I implemented an HLRC protocol that uses automatic updates to propagate diffs transparently as they are produced rather than buffering and sending them using a deliberate update message as in standard HLRC. This eliminates the memory overhead of buffering and sending the diff.

Third, I take full advantage of AU support at the protocol level and use AURC which eliminates diffs altogether and propagates writes to shared data to the remote home immediately rather than waiting for the next release. I compare AURC with HLRC.

### 4.2.3 Basic Operation Costs

Table 25 shows costs of basic operations on the SHRIMP system.

| Operation | Time in microseconds |
|---|---|
| AU Message Latency | 4.75 |
| DU Message Latency | 7.6 |
| 4K Page Transfer | 180 |
| Message Notification | 53 |
| 4K Page Copy | 64 |
| Page Fault | 20 |
| Page Protection | 22 |

Table 25: Basic operations on SHRIMP

### 4.2.4 Applications

I evaluated the performance of the VMMC-based HLRC and AURC protocols on 8 SHRIMP nodes, using a subset of the SPLASH-2 applications and kernels [SWG92] that exhibit both regular and irregular sharing patterns.

| Application | Problem Size | Sequential Execution Time (secs) |
|---|---|---|
| Barnes | 8K bodies | 55 |
| FFT | 256K | 23 |
| LU | $512 \times 512$ | 26 |
| Ocean | $130 \times 130$ | 13 |
| Radix | 1 M keys | 7 |
| Water-Nsquared | 512 molecules | 64 |
| Water-Spatial | 512 molecules | 119 |

Table 26: Applications, problem sizes, and sequential execution times.

Table 26 shows the problem sizes and the sequential execution times for all applications. Problem sizes are small due to the current limitations of the memory mapping size in SHRIMP.

### 4.2.5 Effect of Zero-copy Communication

Figure 22 presents the execution time breakdowns and speedups obtained on an eight-processor SHRIMP using two implementations of the all-software protocol: HLRC-copy

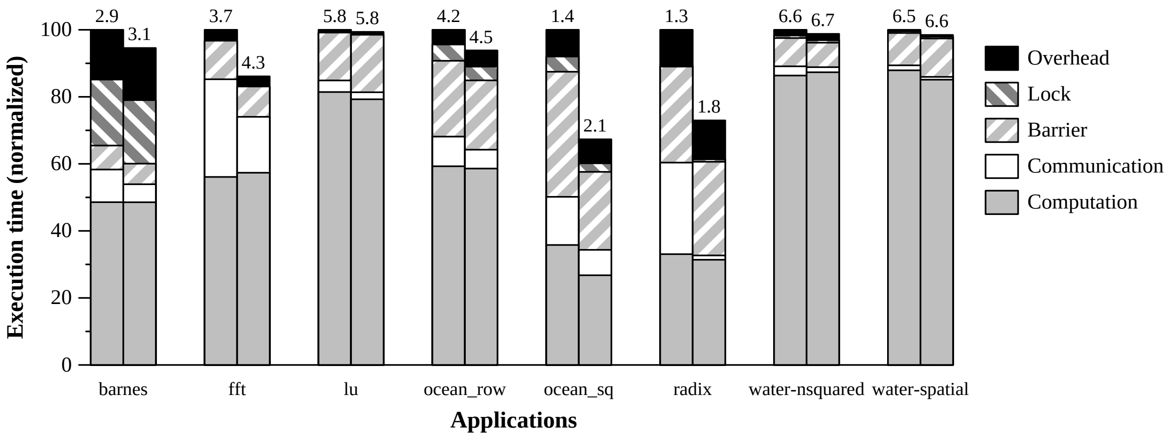



Figure 22: Comparative performance: HLRC-copy(left) vs. HLRC (right). 8 Processors.

and HLRC. The number on top of each bar represents the speedup relative to a sequential run. The difference between these two implementations is in how a page fault is handled at the requester. In HLRC, when a page is sent from the home in response to a page request, it is transferred directly into the physical page frame of the local copy of that page using memory mapped communication. HLRC-copy performs one copy on receive as is done in SVM implementations based on sockets or some other message passing systems. In HLRC-copy the node receives the page in a special buffer and then copies the page from the special buffer into the local copy of that page.

The HLRC protocol achieves speedups over sequential execution in the range of 22%-84% parallel efficiency. Parallel efficiency is defined as the speedup divided by the number of processors. The speedups are good for Water-Nsquared, Water-Spatial, LU, FFT and Ocean-row, and reasonable for Barnes given the small problem sizes I was able to run. Breakdown of execution times show that compared to HLRC, HLRC-copy spends twice as much time in communication, indicating that the extra copy operation halves the effective page transfer bandwidth. This result is consistent with the results reported in [FAB$^+$96]. Overall, HLRC performs significantly better than HLRC-copy for the applications with high communication-to-computation ratios (FFT, Radix and Ocean-square), a little better for those with medium communication-to-computation ratios (Barnes and Ocean-row), and similarly for those with very low communication-to-computation ratios (Water and LU). To conclude, a communication architecture that provides zero-copy data transfer, such as memory mapped communication, is very important to the efficiency of an SVM protocol.

### 4.2.6 Automatic Update as a Communication Accelerator

In contrast to the benefit of removing the copy, using AU to transfer diffs via fine-grained, overlapped writes (HLRC-AU), rather than buffering them together and sending them with DU in standard HLRC, has negligible effects on performance for the applications I have studied.

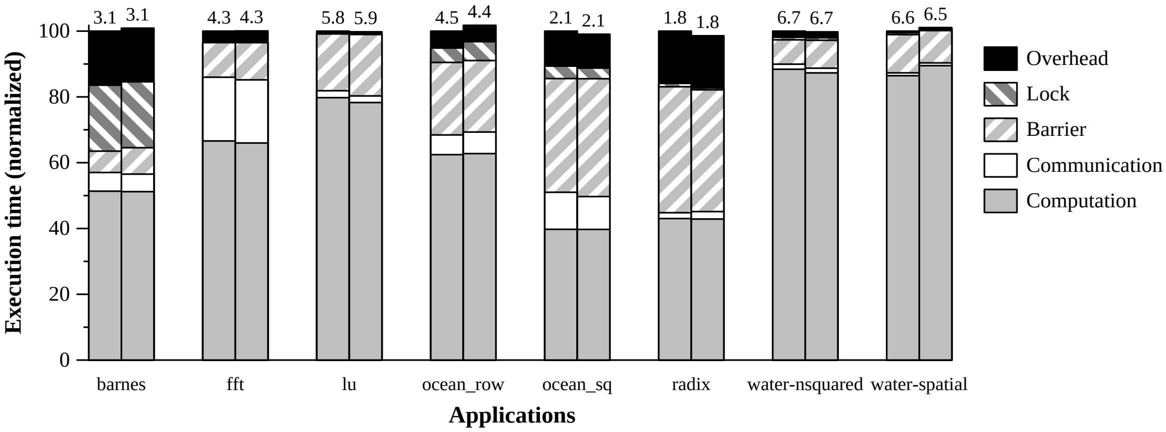



Figure 23: Comparative performance: HLRC(left) vs. HLRC-AU(right). 8 Processors.

Figure 23 shows that HLRC-AU either slightly improves performance for some applications (e.g. Radix and Ocean-square) or slightly hurts the performance of others (e.g. Ocean-row, Barnes and Water-spatial). This is because while AU reduces overhead at the sender compared to explicit transfer, it increases the occupancy of the sender's and receiver's memory buses, possibly reducing the ability of nodes to respond quickly to remote requests. Nonetheless, the performance effects of this simple use of AU are small.

### 4.2.7 Automatic Update as a Protocol Optimization

Figure 24 shows the performance comparison between HLRC and AURC using a 8-node SHRIMP. The results confirm our expectations for AURC.

AURC significantly outperforms HLRC for applications that exhibit significant write-write false sharing: by 15%, 20%, 52%, and 78% for Ocean-row, Barnes, Ocean-square, and Radix, respectively. The communication in Water-Nsquared and Water-Spatial also benefits from AURC, but overall performance does not benefit significantly due to low communication-to-computation ratio in these applications. Radix, an application known for its poor performance on SVM systems [ISL96a], achieves a substantial performance improvement. This is because Radix performs irregularly scattered writes to remotely allocated data in its communication phase which leads to a lot of diff-ing in HLRC but is quite well suited to the automatic fine-grained propagation of writes in AURC. Using AURC, five applications improve their parallel efficiency compared to HLRC, exceeding 50% with the exception of Barnes, Ocean-square and Radix which come between at 20% and 40%.

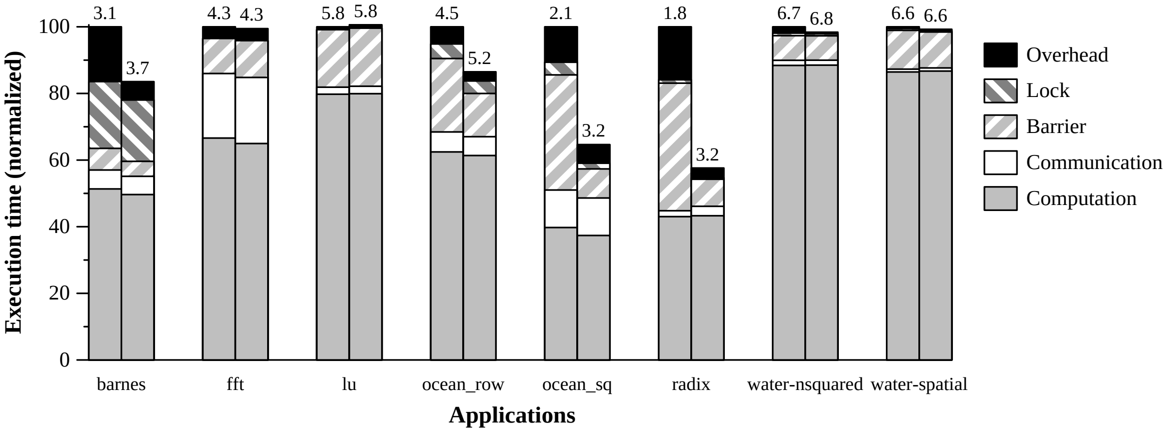



Figure 24: HLRC(left) vs. AURC(right). 8 Processors.

On the other hand, HLRC and AURC perform similarly for applications that exhibit little or no multiple-writer false sharing. These applications are single-writer applications even at page granularity, and if the pages a processor writes are allocated in its local memory then there is no need for the propagation of updates to remote homes. Thus, there is no need for diffs in HLRC, and the AU mechanism is hardly used at all.

Detailed analysis of execution time and protocol cost breakdowns shows that AURC does its job of eliminating diff related overhead very well (see Table 27) without noticeably increasing computation time or network contention due to write-through page mapping and AU traffic. This is true even in Radix, which tends to stress communication bandwidth. The effect on overall performance depends on the characteristics of the application and how much time is spent in computing diffs and its side-effects. The savings in protocol overhead are greater than a factor of 4 in Radix, 3 in Barnes, and 2 in the two Ocean applications.

As a side effect of reducing protocol overhead, the synchronization time of locks and barriers is reduced. This is because the diff computation at release points dilates the critical

| Application | Page misses | | Diffs Created | | Diffs Applied | | Lock Acquires | Barriers |
|---|---|---|---|---|---|---|---|---|
| | HLRC | AURC | HLRC | AURC | HLRC | AURC | | |
| Barnes | 2,517 | 2,498 | 3,398 | 0 | 3,398 | 0 | 20,468 | 8 |
| FFT | 2,240 | 2,240 | 0 | 0 | 0 | 0 | 1 | 10 |
| LU | 234 | 234 | 0 | 0 | 0 | 0 | 1 | 65 |
| Ocean-row | 432 | 443 | 365 | 0 | 365 | 0 | 92 | 399 |
| Ocean-sq | 1,822 | 1,740 | 1,785 | 0 | 1,785 | 0 | 92 | 399 |
| Radix | 166 | 166 | 1,941 | 0 | 1,941 | 0 | 25 | 10 |
| Water Nsquared | 391 | 387 | 222 | 0 | 222 | 0 | 64 | 22 |
| Water Spatial | 631 | 641 | 111 | 0 | 111 | 0 | 22 | 18 |

Table 27: Average number of operations on each node.

sections protected by lock, and hence increases the synchronization and average wait time for a lock. This serialization and the diff computation itself may also magnify load imbalances among processors and increase the time waiting at barriers.

Table 28 shows that by replacing diffs with AU, AURC achieves significant savings in the number of explicit messages. In the Radix case the difference in the number of messages between AURC and HLRC is within an order of magnitude. In terms of data traffic, the gain of AURC due to diff elimination is smaller than the gain in number of messages. The dominator of communication traffic in HLRC is full page transfers. In the current system the AU traffic cannot be measured. However, if we assume that the AU traffic in AURC is comparable to the diff traffic in HLRC, we can conclude that the AU traffic does not contribute much to the total traffic in AURC. The result implies that selecting homes whose writes do not generate AU can be helpful. This result agrees with the previous simulations [IDFL96], and is further confirmed by the absence of visible effects of contention in the computation time of AURC compared with HLRC.

### SVM Performance on 16-node SHRIMP

I evaluated the performance of the home-based protocols on the full-size SHRIMP using those applications for which there was a performance gap between AURC and HLRC on 8 processors. For Barnes and Radix I doubled the problem size used on 8 nodes, while for

| Application | Update traffic | | | | Protocol traffic | | | |
|---|---|---|---|---|---|---|---|---|
| | Number of messages | | Message traffic (Mbytes) | | Number of messages | | Message traffic (Mbytes) | |
| | HLRC | AURC | HLRC | AURC | HLRC | AURC | HLRC | AURC |
| Barnes | 5,915 | 2,498 | 10.4 | 10.1 | 39,600 | 32,785 | 2.2 | 2.16 |
| FFT | 2,240 | 2,240 | 9.2 | 9.2 | 4,600 | 4,600 | 0.4 | 0.4 |
| LU | 234 | 234 | 0.9 | 0.9 | 990 | 990 | 0.04 | 0.04 |
| Ocean-row | 797 | 443 | 2.0 | 1.4 | 6,782 | 6,063 | 0.15 | 0.13 |
| Ocean-sq | 3,607 | 1,740 | 9.1 | 7.0 | 12,130 | 8,478 | 0.4 | 0.32 |
| Radix | 2,107 | 166 | 2.1 | 0.7 | 4,471 | 589 | 0.15 | 0.09 |
| Water Nsquared | 613 | 387 | 1.6 | 1.3 | 1,813 | 1,365 | 0.09 | 0.08 |
| Water Spatial | 742 | 641 | 1.8 | 1.7 | 1,880 | 1,668 | 0.1 | 0.1 |

Table 28: Average communication traffic on each node.

Ocean, I ran the same problem size due to memory constraints. Overall, the results are qualitatively similar to those obtained on the 8-node configuration. They confirm that the diff overhead in HLRC can contaminate the synchronization cost and is responsible for a longer execution time than for AURC (see Figure 25).

### 4.2.8 Write-Only Pages

Radix application captures an interesting effect of the coherence protocol, which is due to its inability to distinguish write-write false sharing from read-write false sharing. This inability stems from the fact that the TLB entries allow read-only but not *write-only* permission to be set. If virtual-memory hardware supported write-only access permission then the SVM protocol could read-protect a page while still allowing writes. This would allow concurrent writers to write to their local copies without generating page misses until a read occurs.

In the absence of such a capability the SVM protocol can turn writers into involuntary readers, thus generating unnecessary page transfer traffic. In Radix this situation occurs at each step during the key permutation phase. The writer of the output array does not itself need to see the data that it writes into the remotely allocated portions of the array. We would like to label these pages of the output array for a permutation as write-only, so the writes to them do not cause page faults. To evaluate the effect of supporting write-only

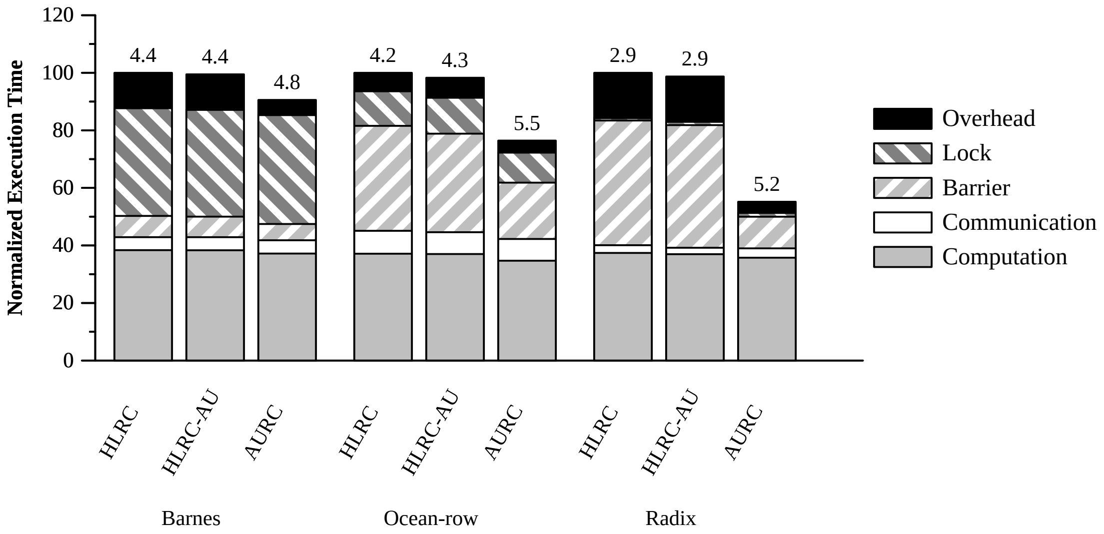



Figure 25: Comparative Performance of HLRC, HLRC-AU and AURC on 16 Processors

pages for HLRC and AURC, I used explicit annotations in the application to indicate write-only pages. The protocol trusts the application and does not invalidate write-only pages, thus emulating a write-only permission control.

Figure 26 shows the effect of this optimization on both HLRC and AURC protocols. In both cases when write-only annotations are used, the number of page misses is dramatically reduced from 166 to less than 10, which in turn considerably reduces communication overhead. But, given that the synchronization cost dominates, the impact of write-only optimization on the performance of Radix remains small overall.

### 4.2.9 Limitations

Three implementation artifacts reduced the conclusiveness of this evaluation. First, the cost of export/import calls is significantly higher than the cost of VMMC communication, due to the current implementation of VMMC model which uses sockets for communication among daemons. As a result, the cost of these operations was factored out by bringing them in the initialization phase of the application which is usually eliminated as a cold-start effect.

Second, the current implementation of the network interface limits the amount of memory that can be mapped for automatic-update. This limits the problem size to the size of physical memory per node rather than the total memory per cluster.

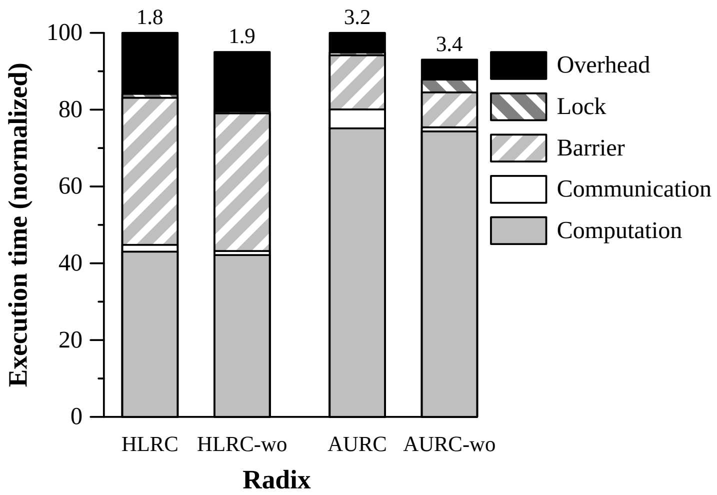



Figure 26: Effect of Write-Only pages on HLRC and AURC. 8 Processors.

## 4.3 Summary

In this chapter, I have investigated the scalability of the home-based protocols and the performance benefit of memory-mapped communication and automatic update on home-based protocols implemented on real hardware platforms.

Using a 64-node Intel Paragon multicomputer, and five applications, we show that the HLRC protocol performs and scales substantially better than the LRC protocol. The HLRC protocol outperforms the standard LRC protocol for several reasons:

- Synchronization cost is higher for the LRC protocol. This is because the waiting time depends to a large extent on the data transfer time, which is both larger and possibly imbalanced in the LRC protocol. For all applications, the synchronization cost is the main component limiting performance for large configurations.

- Data transfer time in the LRC protocol is also larger than in HLRC. Diff accumulation in LRC may produce messages larger than the page-size messages transfered in HLRC. Also, the number of messages is usually larger in LRC.

- An HLRC protocol is simpler than a LRC protocol. As a consequence, HLRC produces less protocol overhead, generates less protocol traffic, and requires substantially less memory than LRC. Garbage collection is not required in HLRC because diffs have a very short lifetime in a home-based protocol.

We have also found that diff overlapping using the communication processor provides performance improvement for multiple-writer applications such as Water-Nsquared and Water-Spatial.

An interesting question is whether home-based protocols are always better than no-homes ones. While our study provides only limited answers, it suggests that home-based protocols are likely to perform robustly for a large number of applications on a large scale machine.

Using SHRIMP, the only platform which supports automatic update exclusively in hardware, I implemented and evaluated three home-based protocols on a subset of SPLASH-2 applications. The results confirmed the expectations set by earlier simulation studies [IDFL96, ISL96a].

First, I found that virtual memory-mapped communication helps home-based LRC protocols because these protocols can easily take advantage of the zero-copy communication allowed by the memory mapping mechanism.

Second, I showed that by taking advantage of the automatic update mechanism in the LRC protocol, one can further improve performance. For applications with write-write false sharing the AURC protocol can significantly outperform the HLRC that uses diffs. For applications without much write-write false sharing, the two protocols perform similarly.

Finally, in my experiments I did not see any effect on the execution time due to write-through caching and AU traffic. This indicates that the AU mechanism is indeed implemented in a way that accelerates shared virtual memory without much hidden costs, and that the expectations from AU in such protocols are confirmed.

# Chapter 5

# Scope Consistency

Systems that preserve coherence at a large granularity, such as shared virtual memory systems, tend to suffer from false sharing and extra communication. Several relaxed memory consistency models [GLL+90, CBZ91, KCZ92, BZ91] have been proposed to alleviate the performance in shared virtual memory systems. However, models with increasing effectiveness also increase programming complexity. The challenge is to define models that maximize the performance benefits of relaxed consistency while minimizing the additional programming requirements.

Protocols based on Release Consistency (RC) [GLL+90] are accepted to offer a reasonable tradeoff between performance and programming complexity for shared virtual memory [CBZ95]. RC guarantees memory consistency only at synchronization points which are marked as *acquire* or *release* operations. Modifications to shared data are globally performed [GLL+90] no later than the next release operation. Lazy Release Consistency (LRC) [KCZ92] is a relaxed implementation of RC in which coherence actions are postponed from the release to the next acquire operation.

The consistency model can be further relaxed by exploiting the association of shared data objects with synchronization variables such as locks or barriers. Entry consistency (EC) [BZ91] lets the acquire operation obtain modifications only to the data that have been explicitly associated with the acquired synchronization variable — not all the shared data that have been modified by other processors — thus reducing unnecessary traffic compared to LRC or AURC. The EC model, however, requires that the coherence protocol be given the associations between data structures and synchronization variables. Thus, while EC can improve performance, the programming burden of explicitly associating synchronization with data can be substantial (since compilers cannot do this automatically today) and EC has not gained wide acceptance.

In this section, I proposes a new memory consistency model called *Scope Consistency* (ScC). ScC offers most of the potential performance advantages of EC without requiring explicit association of data to synchronization variables. ScC introduces the concept of *consistency scope* which effectively establishes the association dynamically and transparently. In most cases, programs that follow the programming discipline of LRC can run with ScC without modifications — their synchronization events imply the scopes. However, not all LRC programs are correct under ScC, and additional scopes may have to be defined in some cases. Nonetheless, even in these cases the extended interface for ScC is simpler to use than the EC programming interface since scopes are associated with sections of code rather than data.

As a consistency model, ScC is independent of implementation and can be used with any shared virtual memory protocol. In this dissertation, I describe a home-based ScC protocol in which modifications are eagerly propagated to the home node for a page, whether supported by hardware automatic update (AU) or in software. Recently the ScC model has been implemented using a no-home based multiple-writer scheme similar to that used in TreadMarks[KCDZ94], in the Brazos prototype [SB97]. This implementation relies heavily on the broadcast support of the interconnect for a hybrid update-invalidate protocol, but it corroborates the promise of our simulation results (Table 2).

To understand the performance implications of ScC I implemented an AU-based ScC and an AURC protocol within the TangoLite simulation framework [Her94], I conducted detailed simulation studies with five Splash-2 [WOT$^+$95a] applications, as well as a synthetic benchmark designed to emphasize a communication pattern that illustrates the benefits of ScC. ScC reduces the number of page faults in the synthetic benchmark by 45%. For three of the five real applications, ScC improves the speedup achieved by AURC by about 10%. For the other two applications, ScC did as well as AURC. One ScC-improved application and both the unimproved ones were already correctly written for scope consistency, while the other two applications needed program changes to comply with ScC.

| **P0** | **P1** |
|---|---|
| Acquire (L0) | |
| X0 = 1 | |
| Release (L0) | |
| | Acquire (L0) |
| | a = X0 |
| | Release (L0) |

Figure 27: Programming in Release Consistency

## 5.1 Relaxed Consistency Models

A consistency model is essentially a contract between the memory system and the parallel program which specifies the order in which memory accesses can be executed by the system [AH90]. Relaxed consistency models define a set of sufficient conditions or rules such that if the program obeys these rules, the system guarantees sequentially consistent memory behavior. The goal of consistency relaxation is to provide the system a certain degree of freedom in ordering memory operations, thus reducing the impact of data access latency and improving the overall performance. This is particularly important in systems that preserve coherence in software and at large granularity. Unfortunately, relaxed consistency models increase programming complexity, leading to a tradeoff with performance. In the following subsections I revisit release consistency [GLL+90] and entry consistency [BZ91], the two main consistency models which are supported in software shared memory systems.

### 5.1.1 Release Consistency

The *Release Consistency(RC)* model was initially introduced for hardware cache-coherent multiprocessors [GLL+90]. It has since become the cornerstone of software shared virtual memory systems [CBZ91, KCZ92, KCDZ94, ISL96a, KS96b].

Release consistency distinguishes between ordinary memory accesses and synchronization accesses defined as either *acquire* or *release*. To make this possible the program must be *properly labeled* [GLL+90] (Figure 27). Only the synchronization accesses are strictly ordered (either sequentially [Lam79] or processor consistent [GW88]). Ordinary memory accesses to different locations are ordered only with respect to synchronization accesses and can be completely reordered between them. The following two definitions are reformulated after [DSB86, SD87, GLL+90]. A complete set of definitions for terms used in consistency models can be found in [AH90].

- A write is *performed with respect to a process P* when a read issued to the same address by P will return the value stored by that write (or a subsequent write to the same location).

- A read is *performed with respect to a process P* when the issuing of a write to the same address by P cannot affect the value returned by the read.

The consistency conditions for release consistency [GLL+90] can be formulated as follows:

1. Before an ordinary memory access is allowed to perform with respect to any other process, all previous acquire operations must have been completed successfully.

2. Before a release operation is allowed to perform with respect to any other process, all previous ordinary accesses must have been completed.

3. Acquire and Release operations must be processor consistent with respect to one another.

The combined effect of these rules is that all memory accesses preceding the release of a lock are guaranteed to be performed before all memory accesses which follow the acquire of the lock. This is the only order among memory accesses which is guaranteed in release consistency.

### 5.1.2 Entry Consistency

The *Entry Consistency(EC)* model was proposed for the Midway software shared memory system [BZ91]. Midway is a variable- or region-based rather than a page-based system. Entry consistency requires each ordinary shared variable to be explicitly associated with some synchronization variable such as a lock or barrier which protects access to that variable (Figure 28). When a lock is acquired, only the variables associated with that lock are updated. The explicit association of data to synchronization further reduces the effect of false sharing but at the same time adds substantial programming complexity compared with release consistency.

| P0 | P1 |
|---|---|
| Lock_Bind(L0, X0) | Lock_Bind(L0, X0) |
| Acquire_Exclusive(L0) | |
| X0 = 1 | |
| Release (L0) | |
| | Acquire_Non_Exclusive (L0) |
| | a = X0 |
| | Release (L0) |

Figure 28: Programming in Entry Consistency

To allow more concurrency, entry consistency distinguishes between exclusive and non-exclusive accesses to a lock. To write the variables associated with a given lock, the process must own the lock, i.e. must acquire the corresponding lock in exclusive mode. In non-exclusive mode, multiple processes can acquire the same lock but they can only read the associated variables. The consistency rules for entry consistency are the following [BZ91]:

1. Before an acquire is allowed to perform, all updates to the guarded shared data must be performed with respect to the process.

2. When a lock is acquired in exclusive mode, no other process may acquire that lock, not even in non-exclusive mode.

3. After a lock is acquired in exclusive mode, the next non-exclusive acquire of that lock performed by any other process is allowed to be performed only after it is performed with respect to the owner of that lock.

## 5.2 Scope Consistency

*Scope Consistency(ScC)* uses a concept called *consistency scope* to implicitly establish the relationship between data and synchronization events. The main attraction of this approach is that in most cases consistency scopes can be derived implicitly from the synchronization already present in programs that conform to release consistency, and can deliver performance improvement similar to entry consistency. When required, additional explicit scopes can be introduced through annotations. The annotations are much easier to use than in EC because the binding is defined in terms of code sections and not variables.

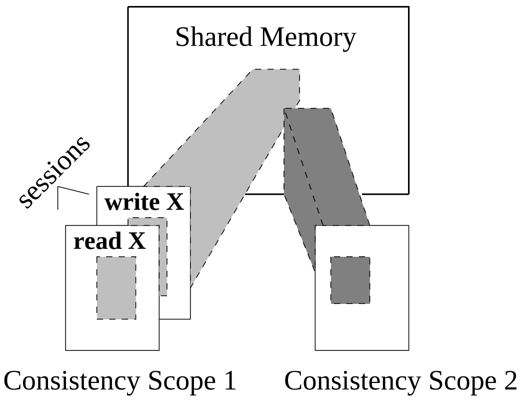



Figure 29: Consistency Scopes

### 5.2.1 Consistency Scope

A *consistency scope* is a limited view of memory with respect to which memory references are performed. That is, modifications to data performed within a scope are only guaranteed to be visible within that scope. One can think of a consistency scope as consisting of all critical sections protected by the same lock. Additionally, barriers define a global consistency scope which includes the entire program. For each consistency scope there is an *open* operation that opens the scope and a *close* operation that closes the scope. Examples are given in Sections 5.2.2 and 5.2.3. The interval during which a consistency scope is open at a given process is called a *session*. Any modifications made within a consistency scope session become visible to processes that then enter new sessions of that scope. Modifications made outside the scope session are not guaranteed to be visible. Figure 29 illustrates consistency scopes. Memory is accessed through different scopes that are independent of one another. Each such perspective or scope is built out of sessions which occur on different processes. Sessions belonging to different scopes can interleave in time or overlap in the code or memory. In addition to the individual consistency scopes whose sessions may be delineated by locks and unlocks, there is also a global consistency scope with regard to which all memory references are performed. The sessions of the global scope are typically delineated by barriers.

To define the model we need an additional definition to distinguish a reference being *performed with respect to a consistency scope* from a reference being *performed with respect to a process*:

| **P0** | **P1** | |
|---|---|---|
| Acquire (L0) | . . . . . . . . . . . . . . . . . . | *--- open scope L0 ---* |
| X0 = 1 | | |
| Acquire (L1) | . . . . . . . . . . . . . . . . . . | *--- open scope L1 ---* |
| X1 = 1 | | |
| Release (L1) | . . . . . . . . . . . . . . . . . . | *--- close scope L1 ---* |
| Release (L0) | . . . . . . . . . . . . . . . . . . | *--- close scope L0 ---* |
| | Acquire (L1) . . . . . . . | *--- open scope L1 ---* |
| | a = X0 | |
| | b = X1 | |
| | Release (L1) . . . . . . . | *--- close scope L1 ---* |

Figure 30: Programming with Lock-based Scopes

- A write that occurs in a consistency scope is *performed with respect to that scope* when the current session of that scope closes.

The consistency rules for Scope Consistency are:

1. Before a new session of a consistency scope is allowed to open at process P, any write previously performed with respect to that consistency scope must be performed with respect to P.

2. A memory access is allowed to perform with respect to a process P only after all consistency scope sessions previously entered by P (in program order) have been successfully opened.

The first rule establishes that after a consistency scope is opened at a process, the updates previously performed *with respect to that consistency scope* are guaranteed to be performed *with respect to that process.* In other words, all the updates which occurred in all previously closed sessions of a scope are guaranteed to be performed in the memory local of the process which opens a new session of that scope.

By forcing open operations to complete before the following memory accesses are allowed to perform, the second rule guarantees that these accesses see a view of memory that is updated according to the first rule.

### 5.2.2 Lock-based Consistency Scopes

As mentioned earlier, in most programs ordinary synchronization events, such as locks and barriers, delimit consistency scopes, and no additional annotations are needed. Critical sections protected by the same lock define a consistency scope, and a particular critical section protected by that lock and entered at runtime delimits a scope session. The lock acquire opens the scope session and the lock release closes it. Barriers define a global consistency scope. The global scope is initially open at all processes. Each process entering a barrier closes the global consistency scope session and opens a new one on barrier exit. When a process gets past a barrier, it is guaranteed to see all changes, either within or outside critical sections, made by any process before that barrier, just as in RC. Any benefits from ScC therefore occur through more specific synchronization within the region between two barriers. For flag-based synchronization an explicit *open_scope* operation can be used in conjunction with a flag set operation to delimit a consistency scope.

To reformulate the above general rules of ScC in terms of lock-based scopes we use the term *performed with respect to a lock* for *performed with respect to a lock-based scope*:

1. A write is *performed with respect to a lock or barrier* when that lock or barrier is released.

This definition implies that multiple processes can simultaneously perform writes with respect to a barrier but only one process at a time can perform writes with respect to a lock. The second definition is as before. The scope consistency rules can be formulated for locks and barriers as following:

1. Before a lock (or barrier) acquire is allowed to perform by process P, all writes performed with respect to that lock (or barrier) must be also performed with respect to P.

2. A memory access is allowed to perform with respect to a process P only after all previous acquire events at P have succeeded.

If scope consistency semantics are attached to both locks and barriers, RC is upgraded to ScC in a very simple way: every memory reference occurs at least in the global consistency scope which is simultaneously open at all processes. In addition, a memory reference is also considered to occur in all consistency scopes corresponding to the locks which have been acquired but not yet released.

**Example I**

Figure 30 illustrates scope consistency with an example. When Process P0 acquires lock L0 it enters the scope defined by that lock. Process P0 writes X0 while in scope L0 and writes X1 while in scopes L0 and L1. Next, process P1 acquires lock L1, thus opening scope L1, and reads both X0 and X1. Under scope consistency, P1 is guaranteed to see P0's write to X1 assuming P1 acquires L1 after P0 releases it. However there is no guarantee that P1 will see P0's write to X0 because the scope L0 was not opened at P1 when it read X0; thus, P0 need not propagate the modification of X0 to P1. Under RC or LRC, this latter guarantee would exist since the consistency model does not distinguish among synchronization variables, and the modification would have to be propagated. However, if P1 had acquired L0 instead of L1 and if that had occurred after P0 had released L0, then both writes to X0 and X1 performed by P0 would have become visible at P1.

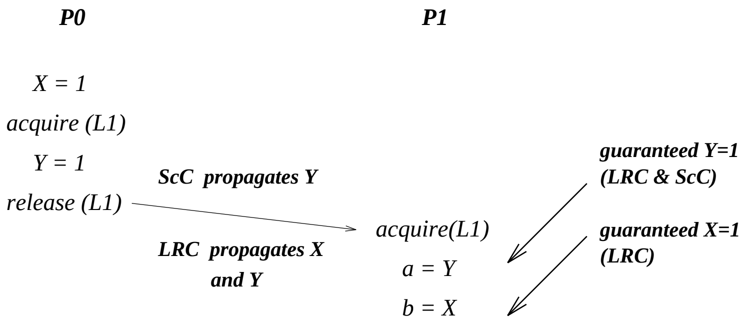



Figure 31: Scope Consistency versus Release Consistency.

Not all programs that follow the programming discipline of RC are *ScC-correct* without modifications. A program is called ScC-correct if it observes a sequential consistency memory when run on a system that supports ScC. The following are sufficient conditions for an RC program to be ScC-correct:

1. Lock-protected modifications to shared data are not expected to be visible at a process before at least one of the protecting locks is acquired by that process.

2. Modifications to shared data that are not protected by a lock are not expected to be visible at a process before the next barrier.

**Example II**

Figure 31 provides an example showing the type of programming problems which can be encountered in switching from RC to ScC. RC is represented by the LRC protocol. $P_0$ writes $X$ first and then $Y$; $Y$ is written in the critical section guarded by lock $L_1$ while $X$ is written outside it. $P_1$ acquires $L_1$ after $P_0$ has released it.

Under LRC [KCZ92], after acquiring $L_1$, $P_1$ is guaranteed to see all writes which occurred at $P_0$ before the release of $L_1$, so the new value of both $X$ and $Y$ are guaranteed to be visible. On the other hand, ScC guarantees only that $P_1$ sees the new value of $Y$, because the release event ensured memory consistency over only one consistency scope, namely the one defined by all critical sections guarded by the same lock $L_1$. To guarantee the visibility of both $X$ and $Y$, the write to $X$ by $P_0$ must be performed inside the same critical section, which makes sense if the writes are semantically related. Otherwise, the write to $X$ will be guaranteed to be visible at $P_1$ only at the next barrier.

Some programs which use task queues to distribute the computation exemplify this situation: the critical section protects the task queue modification and not the writes to data performed by that task. However, the programmer assumes that when a task is selected from the task queue, the updates by previously completed tasks have been performed and are visible. This assumption is true under RC but not necessarily under ScC if only the task queue updates are protected by locks.

Thus, precautions may have to be taken to ensure that programs are ScC-correct. One solution, as we shall see for the Barnes-Hut application, is to make critical sections large enough to contain all modifications which are intended to be performed at the time when that lock is acquired. In some cases, this approach may not be effective since it can increase serialization and hence hurt performance. An alternative solution is to add new critical sections to protect the unprotected modifications or, better yet, to use explicit scopes in addition to the existing lock-based ones.

### 5.2.3 Explicit Scopes

Scopes can be created dynamically using explicit annotations. Figure 32 illustrates a typical example of a task-queue based program which uses explicit scopes in addition to lock-based scopes. New scopes are created to enclose the writes performed by tasks and are attached to the tasks themselves. The primitive *open_scope* with no argument creates a new scope

| **P0** | **P1** | |
|---|---|---|
| S = open_scope() | | --- *create a new scope* --- |
| X0 = 1 | | |
| close_scope (S) | | --- *close the new scope* --- |
| Acquire (L) | | --- *open scope L* --- |
| task. scope = S | | --- *save the scope id* --- |
| Release (L) | | --- *close scope L* --- |
| | Acquire (L) | --- *open scope L* --- |
| | T = task.scope | --- *get the scope id in T* --- |
| | Release (L) | --- *close scope L* --- |
| | open_scope (T) | --- *open scope T* --- |
| | a = X0 | |

Figure 32: Programming with Explicit Scopes

and returns its *scope id*. Within that scope $X0$ is updated. Next, the scope is closed and its id is stored in the per-task data. When the task is scheduled at P1, the corresponding scope is explicitly opened with *open_scope*, thus ensuring the visibility of the update performed at P0. Cholesky was modified using explicit scopes to make it ScC-correct.

Although the approach using *open-scope* and *close-scope* appears conceptually similar to the use of non-exclusive lock acquire in EC, there are significant differences both in terms of programming model as well as protocol implementation. Unlike in EC, in ScC explicit scopes do not require explicit binding of data to locks, thus making them much easier to use. Scopes are attached to code sections, not to data. The "bindings" to data are dynamically created based on the memory accesses issued between *open_scope* and *close_scope* while a session of the scope is open. Adding more locks is, however, an alternative solution to adding explicit scopes in ScC, and its performance benefit would be very similar to the one in EC.

## 5.3 Comparison with RC and EC

Scope consistency is a relaxed consistency model situated between RC [GLL+90] and EC [BZ91]. In what follows I compare ScC with RC and EC using an LRC protocol to represent RC.

### 5.3.1 ScC versus RC

Both ScC and RC do not require explicit binding of data to synchronization. However, ScC assumes an implicit binding of memory accesses to scope determined from the program structure. ScC reduces to RC for applications that use only global synchronization, such as barriers, because then there is only one global scope.

**Reduced False Sharing.** A large coherence granularity is a source of false sharing. By postponing memory coherence from release to acquire and supporting multiple writers within the same page, LRC eliminates the effects of certain false-sharing cases compared with an eager) RC-based shared virtual memory. But to comply with the RC model, the acquiring processor in LRC must see all the data modifications seen so far by the releasing processor.

ScC goes further in eliminating the effects of false sharing. By supporting multiple memory scopes, ScC can postpone seeing memory updates until the *corresponding* scope is open. This allows the acquiring processor to perform only the modifications which occurred in the same scope.

Figure 33 illustrates the benefit of ScC over LRC in reducing the impact of false sharing. Assume that an array of $N$ particles $X$ is shared by $p$ processes. Each particle $X_i$ is protected by a different lock $L_i$. Assume $X_0$ and $X_1$ lie on page *0* while $X_2$ and $X_3$ lie on page *1*.

$P_0$ computes an interaction between particles $X_0$ and $X_2$, then $P_1$ computes the interaction between $X_2$ and $X_1$. To accomplish this, $P_0$ modifies first $X_0$ and then $X_2$ by acquiring $L_0$ and $L_2$ locks. When $P_1$ acquires lock $L_2$ from $P_0$, under the LRC model it has to see both modifications (to $X_0$ and $X_2$) since the acquire event is allowed to perform only after all previous memory accesses have been performed. Therefore, both pages (*0* and *1*) will be invalidated by $P_1$ in an invalidation-based protocol.

In fact, $P_1$ does not need to see the modification to $X_0$ since it does not acquire $L_0$, and $X_1$ may be already up-to-date. Despite this, the invalidation that occurred on page *0* when acquiring $L_2$ causes a page miss when accessing $X_1$, due to false sharing.

In ScC, when lock $L_2$ is acquired by $P_1$ only the modifications protected by $L_2$ are propagated. In this case, an invalidation and the subsequent miss will be performed on page *1* but not on page *0*. The program will behave correctly since the new value for $X_0$ is not needed in computing the interaction between $X_2$ and $X_1$.

**Programming Effort.** Like RC, ScC requires explicit synchronizations which must be labeled with system-supplied primitives (lock or barrier). However, ensuring the correctness of a RC program under ScC is not trivial since it requires checking whether dependencies between data updates assumed under RC are preserved under ScC. Program modifications are sometimes necessary to make an RC-correct program ScC-correct.

### 5.3.2 ScC versus EC

Both EC and ScC relax the memory consistency model by taking advantage of the relationship between data and synchronization. Both schemes are particularly effective for applications that use a lot of point to point or mutually exclusive synchronization. However, the solutions proposed by the two schemes are quite different.

**Binding.** In EC [BZ91], the binding between *data objects* and *synchronization objects* is explicitly specified by the programmer. In fact, the solution proposed by EC is for variable-based software shared memory systems, close to object-based or region-based systems, not for transparent page-based systems. EC requires explicit associations of data with synchronizations because the protocol relies on this information to identify which modifications must be propagated and when. This is why EC can be implemented very effectively with an update-based protocol. In both EC and ScC, binding can change during the program lifetime, but EC requires the programmer to explicitly rebind a lock to the new data it protects.

In ScC rebinding occurs automatically since it is detected by the system based on the modifications which occur in the sessions of that scope. EC binds data to locks explicitly while in ScC the binding is implicit and dynamically created by the memory accesses.

**Global Synchronization.** The explicit binding in EC has difficulty including global synchronization in the model [ACD+96]. The alternatives are: (1) not to bind data to barriers at all; (2) to have explicit binding as for locks; (3) to bind the entire address space. The first approach substantially limits the intuition of shared memory since barriers do not cause modifications to become visible. It requires the programmer to employ read-only locks to cause modifications to be visible after a barrier. This is the scheme used in Midway [BZ91]. The second approach is difficult to program because it may require multiple bindings if the barriers are used to separate different phases of the computation. The third approach requires high communication in a update-based protocol like EC uses.

ScC introduces a global scope which is a simple intuition for global synchronization. In this scope, ScC guarantees the visibility of all modifications. This is the same as the third approach discussed above. The global scope is particularly useful in ScC to avoid

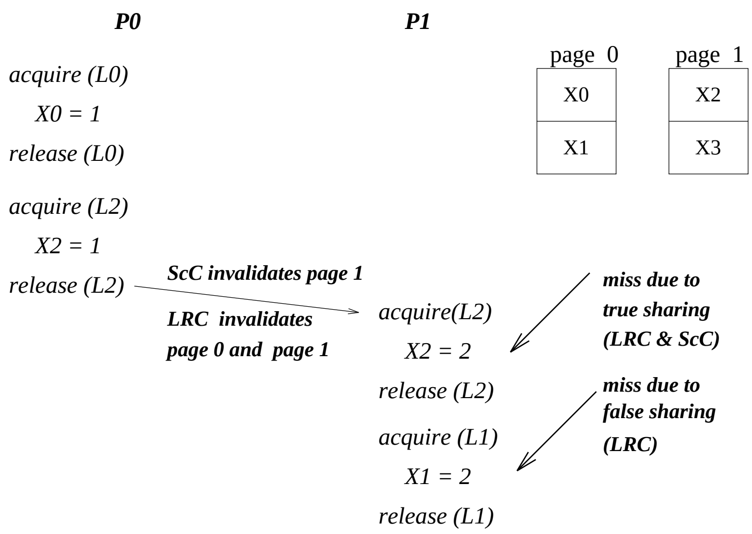



Figure 33: False Sharing in ScC versus LRC

reasoning about update propagation in terms of transitivity of lock-based scopes. The model is simple for the programmer: after each barrier the entire shared address space is guaranteed to be coherent. In addition, unlike in EC, the global scope solution is reasonable in ScC because ScC systems will typically employ an invalidation-based, rather than an update-based protocol.

**Granularity.** EC and ScC systems have a granularity of binding and a granularity of communication. In EC the binding granularity is of variable size because it corresponds to the *data objects* the programmer binds to *synchronization objects*. Since binding in EC must be explicitly specified, its granularity is an important factor in performance and programming complexity. Binding large contiguous pieces of memory may reintroduce the effects of false sharing and require additional lock rebindings. Small non-contiguous bindings are more efficient because they require less communication and/or less processing. But they are also more difficult to program since the programmer must establish many more bindings for each lock. Determining a good binding granularity is the task of the programmer. The communication granularity depends on how write collection is implemented. Midway [BZ91, ZSB94] uses software dirty bits so the amount of communication is strictly proportional to the size of the updated data.

In ScC the binding granularity is determined by the write detection granularity, which is of word size for both automatic update and all-software write detection schemes. Therefore any modification occurring in a given scope automatically binds the corresponding location to that scope. However, the communication granularity in ScC is typically of page size if ScC is implemented with a shared virtual memory protocol which means that in order to fetch a location which is bound to a scope, the entire page which contains that location is fetched. On the other hand, although the communication volume may be higher in ScC than in EC due to page fragmentation, the page size transfers allow ScC to benefit from potential prefetching [ACD+96].

**Programming Effort.** ScC is potentially easier to program than EC because in most cases it does not require any change to RC programs. When RC programs are not ScC, adding explicit scope annotations in ScC requires similar program understanding as for adding explicit binding in EC. However, adding annotations is easier in ScC since it is code-oriented rather than data-oriented.

## 5.4 ScC Protocols

Scope consistency can be implemented using page-based invalidation protocols similar to those used in traditional shared virtual memory systems. For such protocols ScC reduces the impact of false sharing caused by lock-based synchronization resulting in fewer page invalidations between barriers. Savings in page invalidations produce savings in communication cost only if the number of data fetching messages and the data traffic are proportional to the number of page faults and not to the number of modifications. In traditional diff-based systems like TreadMarks [KCDZ94], the data traffic consists of diffs; thus, it is proportional to the number of updates. A *diff* is a run-length encoding of the modifications performed by one process on a shared page. Since all the updates must be eventually performed, avoiding some page invalidations using ScC makes little difference compared with LRC because the total data traffic remains the same.

Home-based protocols are a family of protocols which can benefit from ScC because the traffic on page fault is proportional to the number of page faults, not the number of modifications to those pages. In home-based protocols, the updates performed on each local copy of a page are eagerly propagated to the home of the page. The home copy is thus kept up-to-date, so on a page fault the page is fetched from the home rather than diffs being fetched from all writers. Because each page fault requires a full-page transfer from the home regardless of the number of updates actually performed, the data traffic and the number of data fetch messages in home-based protocols are proportional to the number of page faults. Consequently, the impact of ScC on home-based protocols can be significant.

This section describes an AU-based scope consistency protocol which uses lock-based consistency scopes. Pages are versioned using vector timestamps like in AURC. In addition, ScC introduces an incarnation number for each lock which is incremented with each release of that lock. For each lock incarnation it produces, a processor keeps an *update list* of pages it updated during that incarnation. If a page is updated while several critical sections are open, the update will appear in the update list of all these locks for the corresponding incarnations. In particular, all pages are updated since the last barrier will be reported for the global scope at the next barrier event.

In what follows, I describe how the ScC protocol works for lock-based scopes. The actions are similar for general acquires and releases if explicit scopes are added. Each process remembers the last incarnation it produced for each lock it ever held. At acquire time, the acquiring process sends with the lock request the last incarnation number it produced for that lock. Based on this information, the releasing processor provides the acquirer with the update lists only for the missing lock incarnations. By contrast, in AURC the releasing processor would have provided the update list for all the missing intervals regardless of whether or not they have been produced by the acquired lock. The acquirer uses the update lists to make its address space scope consistent by selectively invalidating the obsolete pages.

The ScC model can also be implemented in an all-software home-based protocol following the same approach used to convert AURC to ScC.

## 5.5 Performance Evaluation

To evaluate the performance impact of scope consistency, I implemented both the AU-based ScC and AURC protocols on top of a shared virtual memory multiprocessor simulator. The simulator interfaces with the TangoLite execution-driven reference generator [Her94]. The architectural parameters (Table 29) are essentially those of the SHRIMP multicomputer. The simulator handles contention in detail both at the memory bus level as well as in the network interface between the incoming and outgoing traffic. Contention in the network itself is not simulated but this does not affect the comparison much since the amount of AU-based traffic is the same for both protocols.

| Parameter | Value |
|---|---|
| Processor clock | 60 MHz |
| Page size | 4 Kbytes |
| Data cache | 256 Kbytes |
| Cache line size | 32 bytes |
| Write buffer size | 4 words |
| Memory bus bandwidth | 80 Mbytes/sec |
| Page transfer bandwidth | 28 Mbytes/sec |
| Trap cost | 150 cycles |
| Message latency | 5 $\mu$sec |

Table 29: System Parameters Used in Simulation.

I evaluate the performance gained from ScC for different classes of sharing patterns in real applications. The programs for this evaluation were selected from the Splash-2 suite [WOT$^+$95a] to cover specific cases of interest. The problem sizes used are the default sizes indicated in Splash-2. I also wrote a simplified N-body kernel representing a simple pairwise force calculation phase [SHG92] for which we arranged data distribution to cause a lot of false sharing. This is intended to artificially showcase the potential benefits of ScC. The benefits obtained from ScC were measured by two means: reductions in the number of page misses, and increases in speedup (hence performance) obtained. The former measure is objective to a large extent because it is not directly dependent on simulation parameters. The latter is what we ultimately care about.

| Application | Problem Size | Number of page misses per processor | | Speedup | |
|---|---|---|---|---|---|
| | | AURC | ScC | AURC | ScC |
| Modified N-body kernel | - | 1823 | 1002 | - | - |
| LU | 512-by-512 | 398 | 398 | 9.4 | 9.4 |
| Water-Nsquared | 512 molecules | 1681 | 1651 | 11.3 | 12 |
| Barnes | 16K particles | 2700 | 2400 | 9.2 | 10.1 |
| Raytrace | balls4 | 9052 | 8107 | 7.3 | 8.3 |
| Cholesky | 1.7M | 3875 | 2866 | 5.8 | 6.5 |

Table 30: Simulations Results for ScC and AURC, 16 Processors

The simulation results are shown in Figure 30, and will be analyzed next. For applications which are improved by ScC (Barnes, Raytrace and Cholesky), the number of page misses is reduced from 10% up to 26%. For all three applications, the speedup is improved by 10-14%. LU and Water are insensitive to ScC. The artificial N-body kernel verifies that the savings in the number of page misses can be dramatic under certain circumstances.

The performance benefit of AU-ScC over AURC comes from two main sources: less communication due to a reduced false sharing effect, and lower protocol overhead due to simpler coherence handling (coherence bookkeeping is per-lock instead of per-process). In what follows, I analyze the sharing patterns of the applications to understand how and when the ScC model provides the savings in the communication traffic. I start with the artificial N-body kernel for illustration.

### 5.5.1 The Modified N-body kernel

The performance benefit of AU-ScC over AURC or other RC based protocols depends on the presence of false sharing effects. Our N-body kernel, a simplified force calculation from an $O(n)$ N-body algorithm using particles and space cells, displays such false sharing. It is based loosely on the Water-spatial application in Splash-2 [WOT+95a]. The computational domain is divided into cells which are distributed among processors. Each processor computes the interactions of the particles belonging to its cell with particles from nearby cells. As a result of each pairwise interaction between particles, particle properties are modified and particles may move between cells, causing cells to be updated as well.

Every particle and space cell is protected by a distinct lock. In this example, the data structure holding the particles is allocated as an array. Since particles move, the locations of particles in the array has little to do with their locations in physical space and hence in cells. Under these circumstances, force computation and particle/cell updates – particularly the synchronization for a global status variable – can lead to random false sharing which is difficult to trace. I deliberately choose an order of interactions which causes false sharing to occur once for each pair computation. For instance, assume that the $i$-th interaction at processor $p$ occurs between one of its particles, say $a$, and a particle $b$ which belongs to processor $q$, and the $i$+1-th interaction at processor $q$ occurs between its particle $c$ and another particle $d$. The kernel arranges that at the end of $i$-th interaction, $p$ and $q$ update the global status variables using locks in that order, and also that $b$ and $c$ are located on the same page. The aggressive false sharing that results under AURC is caused by the lock-based synchronization for global status variables that is unrelated to the particle/cell updates, and is hence perfect for illustrating the gains from ScC. ScC reduces page misses by 45% compared to AURC. Speedups are of course irrelevant for this kernel.

### 5.5.2 ScC-insensitive Applications

ScC does not provide any performance improvement for many types of applications. For example, applications that only use barriers and no locks are not helped. The **LU** factorization kernel is an example, and ScC performs exactly like AURC in this case. Another example is **Water-Nsquared**, an $O(n^2)$ molecular dynamics calculation that is another type of N-body simulation. It computes intermolecular interactions using some lock-based synchronization to update particle forces. However, the benefit of ScC is small in this case since the unrelated synchronization effect is not present. Also, accesses to the particle array are very regular and avoid false sharing. This is because there are no space cells, and a processor's assigned particles are allocated contiguously in the particle array.

### 5.5.3 ScC-sensitive Applications

The question of whether or not scope consistency can provide any performance benefit on real applications was addressed by choosing three important types of applications: Barnes-Hut, which uses a hierarchical method to solve the N-body problem; Raytrace, a rendering application from computer graphics using ray tracing; and Cholesky, which performs a blocked Cholesky factorization of a sparse positive definite matrix. These applications represent not only distinct domains but also distinct characteristics relevant to ScC.

In **Barnes-Hut** the computational domain is represented as an octree of space cells. The leaves of the octree contain particles, and the computation for both particles and space cells are assigned to processors based on their positions in space. While very different, the computation in some ways follows a similar pattern to that described for our artificial N-body kernel. At each step, the octree is rebuilt based on the current positions of the bodies. Each processor computes the forces for its particles by partially traversing the tree. Interactions with far-away particles are replaced by the interaction with their center of mass. At the end of each time-step the new positions for the bodies are computed.

False sharing occurs at page granularity because the properties of particles and cells are updated in an order that has little to do with their placement in the particle and cell arrays, and the placement in the array has little to do with the assignment of particles or cells to processes. The false sharing is particularly large in the tree-building phase where many of these updates are protected by per-cell locks, so unrelated lock-protected updates occur by different processes to the same page.

While Barnes-Hut shows performance improvement from ScC, I had to expand some critical sections in the tree-building and force calculation phases in order to include all the modifications which are supposed to be visible the next time the lock is acquired. This is the case, for instance, during the distributed bottom-up computation of the center of mass. Before computing the center of mass for a cell, a processor has to wait to have it computed for all of its subcells. When a subcell is ready, it signals this through a "done" flag. Before the flag is set, various fields of that subcell are updated. Accesses to this done flag occur in critical sections. Under AURC, the critical section needs to protect only the flag update: once the consumer acquires the lock and finds the flag set, it is guaranteed to see all the updates previously performed on that subcell. In ScC, this is not guaranteed unless the critical section is extended to protect the modifications to the cell data structure which are supposed to become visible at the consumer after the flag is set. This is easy to do in the program.

In **Raytrace**, we get the same benefit as in Barnes-Hut but without changing the program. In fact Raytrace is sensitive to ScC for very different reasons than Barnes-Hut. Raytrace works by creating tasks corresponding to rays that are traced through each pixel in the image plane. Task are dynamically distributed among processors using a system of distributed task queues (one per processor) with task stealing for load balancing. Each task queue is protected by a distinct lock, and the number of lock events is high. Based on the interaction with the read-only scene the color of each pixel of the image is computed. The image pixel updates are outside any critical section since each pixel is accessed only by the process to which it is assigned. Therefore, there is no need to make the local copies of the image coherent until the next barrier. However, since the image updating alternates with the critical sections for task queue access and management, under AURC false sharing of image data will induce a lot of unnecessary page invalidations on the image plane due to the unrelated intervening synchronization. In contrast, in ScC the updates to the image are not in the scope of the task queue locks but in the global scope, so they do not cause invalidations to occur to image pages when acquiring task queue locks. Updates to image data are collected at the home of the relevant page, and the local copies are invalidated only at the next barrier which closes the global scope. The reduction in false-sharing causes a significant increase in performance, even though most of the time in the program is spent on traversing the read-only scene tracing rays rather than updating the image.

**Cholesky** is an application which also uses task queues to dynamically distribute computation. The number of task queues is again equal to the number of processors. A task computes one block and produces new tasks corresponding to dependent blocks. These tasks are exported to the task queues of the processors owning those blocks. When a new task executes it assumes that blocks modified by the task that created it are visible. Task queues are accessed with mutual exclusion using locks. The critical sections are small and contain only the insertion and deletion operations on the task queues. However, since the modifications of the blocks themselves occur outside the scope of these critical sections, they are not visible to the new tasks and the program does not run correctly under ScC as it is. To ensure correct execution under ScC, the tasks themselves need to be contained in scopes related to the corresponding task queue operations. Explicit scope annotations were used to create scopes dynamically and store their identifiers in the task data structure as shown in the example in Figure 32. Introducing explicit scopes increases the protocol overhead due to the communication involved between the scope producer and the scope consumer. The effect is conceptually similar with the increased synchronization cost usually required by EC over LRC. The question is whether the additional cost pays off or not. It turns out that ScC substantially reduces the number of page faults for Cholesky (over 25%). The effects on overall speedup are smaller, about 12%, because of the increase in protocol overhead and communication due to explicit scopes.

### 5.5.4 Limitations

The main limitation of this evaluation is that it is based on simulations. Nevertheless, the false sharing reduction, expressed as the difference in the number of page faults, is accurate and does not depend on the simulation parameters. Moreover, the performance evaluation of Brazos, an implementation of ScC on a NT-based PC cluster, confirms the simulations at least for Raytrace. For the other applications, the authors of Brazos conclude that the major impact on performance came from the multicast and multithreaded support used in the protocol.

It is important for future research to formalize the definition of scope consistency and then prove the condition stated in sections 5.2.1 and 5.2.2 are indeed sufficient to define ScC-correct programs, similar to what has been done for other relaxed consistency models [AH90, GMG91].

## 5.6 Summary

In this chapter I propose and evaluate a new consistency model, called Scope Consistency, targeted at shared virtual memory systems. ScC provides the performance benefits of a model like entry consistency, but without the programming complexity of explicitly associating data with synchronization. In most cases, programs that run under RC also run under ScC without modifications, and can achieve performance improvements with no changes. Some programs need changes to comply with ScC, in order to ensure that semantically related writes are related by synchronization scopes. These changes include moving some synchronization operations to expand critical sections or adding explicit scope annotations. I claim that programming in ScC is simpler than in EC since scopes are associated with sections of code rather than with data.

I have also described and evaluated a scope consistency protocol using the automatic update mechanism provided by recent network interface technologies. I found performance benefits for real applications that suffer false sharing due to unrelated synchronizations, and no losses in performance in other cases. I have shown how the ScC model can be implemented entirely in software in the absence of automatic update support, by using a home-based protocol similar to HLRC. I expect performance benefits in this case as well. While scope consistency is not always completely transparent, it is nevertheless promising for systems built with the increasingly popular network interfaces that support automatic remote updates, as well as potentially for all-software shared virtual memory systems.

# Chapter 6

# Conclusions

In this dissertation, I investigated how to improve the performance of the shared virtual memory approach by looking at consistency models, protocols, hardware support and applications. This dissertation contributes to all these areas of SVM research: a new consistency model called scope consistency, a new family of LRC protocols called home-based LRC protocols, SVM protocols with automatic-update architectural support and application classification with respect to their behavior under SVM. The main conclusion of this research is that the performance of shared virtual memory can be significantly improved by the synergism of solutions in all these areas. On the other hand, it becomes obvious that there is still a performance gap between hardware shared memory and software shared memory approaches. More research is needed, especially in hardware support and application design, before the performance of the software approach can be considered satisfactory.

Although mature and "well-respected", the traditional lazy release consistency appears to suffer from large and not always motivated communication, memory and protocol overhead. Moreover, as the network technology shifts from expensive, kernel-controlled TCP/IP based protocol stack on local area networks to low-overhead, user-level memory mapped communication on system area networks, some of the tradeoff decisions made at the beginning of the 90's for LRC become inadequate.

In this dissertation, I propose a novel write collection scheme for lazy release consistency called *home-based LRC*. The basic idea is to propagate invalidations and data updates with different degrees of laziness. In the traditional LRC both the invalidations and the updates are propagated and applied as late as possible. While it is good for invalidation propagation which is done at the acquire time and in a single roundtrip message, maximum laziness is not good for update propagation because in the presence of false sharing, updates are produced concurrently by multiple writers and update collection can become extremely expensive. Diff accumulation for instance can significantly increase the network traffic, memory consumption and protocol complexity and overhead.

In the home-based LRC approach, invalidations are still propagated as lazily as possible. However, updates are propagated in two stages: eagerly to one selected node per page, called home, and lazily from home to other non-home nodes. This approach has multiple benefits that can impact performance. The coherence state is simple and more scalable than the traditional LRC approach, low diff and memory overheads. In this dissertation, I describe two home-based protocols. *Automatic Update Release Consistency* (AURC) propagates the updates to the home using hardware support for fine-grained remote writes. *Home-based Lazy Release Consistency* (HLRC) implements a home-based write collection entirely in software using the same diff mechanism as the traditional LRC.

As the protocol struggle to deliver better performance approaches saturation, the complexity of the coherence algorithm has steadily increased. Adaptive LRC protocols are in this sense the current extreme case. In this context the simplicity of the home-based protocols becomes an important feature. Both the AURC and HLRC protocols were encoded for VMMC library in less than 1,000 lines of C code. Implementing the protocols to extensively use the memory-mapped communication in order to eliminate every possible memory copy and buffer management was nevertheless a challenging programming task.

As the entire SVM research has entered a phase of relative maturity, performance scalability becomes a crucial metric in the protocol evaluation. The HLRC proposed in this dissertation has shown to scale better than the traditional LRC approach, mainly because both the communication and memory overhead per node remains constant as the number of nodes increase. In LRC, for irregular applications that expose write-write false sharing, the number of messages, diff and memory overhead per node increase linearly with the number of nodes. To evaluate the scalability of the home-based approach, both the HLRC and the traditional LRC protocols have been implemented on a 64-node Intel Paragon. Although, the absolute speedup is affected by the high costs of communication and interrupt handling of Paragon, the communication and memory statistics indicate a substantial gap in the frequency of messages and memory consumption which are independent of architectural parameters. Moreover, to our knowledge, this is the first implementation and evaluation of a SVM protocol on such a large system and getting parallel efficiencies closer or above 50% is in itself an encouraging result for SVM in general.

Memory-mapped communication is a promising user-level communication abstraction which eliminates most of the message, memory and protocol overhead that typically occurs in traditional message-passing environments. Home-based protocols take advantage of memory-mapped communication by trading potentially higher bandwidth requirement for a lower memory and protocol overheads. In order to evaluate the effect of memory-mapped communication on home-based SVM protocols, I implemented both AURC and HLRC protocols on SHRIMP, a PC-based multicomputer that supports user-level memory-mapped communication and automatic update through a custom-designed network interface. The protocol evaluations on the SHRIMP platform indicate that (i) zero-copy memory-mapped communication can significantly improve the efficiency of home-based protocols, and that (ii) the automatic update support is an effective SVM accelerator. Moreover, the AURC evaluation shows that the performance of a home-based protocol can be improved with as little as a hardware support for remote writes (which is the easiest feature to provide in a memory-mapped network interface beyond explicit transfers). This, in addition to the fact that in the experiments with AURC on SHRIMP, I did on not see any effect on the execution time from write-through caching and automatic update traffic, indicate that a simple hardware support for automatic update can be an effective solution to improve the performance of SVM.

This dissertation also introduces *scope consistency* (ScC), a consistency model similar to entry consistency, for which a home-based protocol is described and evaluated through simulations. Scope consistency exploits the implicit and dynamic association between data and synchronization defined by the memory accesses that occur inside scopes. A scope is delimited by the critical sections protected by the same lock. An attraction of the ScC model is that in most cases programs that follow the programming discipline of LRC can run with ScC without modifications — their synchronization events imply the scopes — and that scopes are associated with sections of code rather than data. However, not all LRC programs are correct under ScC, and checking the correctness of a parallel program for ScC is not as trivial as for LRC.

As a consistency model, ScC is independent of implementation and can be used with any shared virtual memory protocol. In this dissertation, I describe a home-based ScC protocol in which modifications are eagerly propagated to the home node for a page, whether supported by hardware automatic update (AU) or in software. A simulation-based comparative evaluation of ScC and AURC protocols shows that ScC reduces false sharing for irregular applications with lock-based synchronization. Expressing the simulation results in terms of frequency of page faults eliminates the artifacts of simulation and makes the conclusion significant.

Finally, to understand the performance of all these SVM protocols and to allow a comprehensive evaluation, I propose a classification of parallel applications that can serve as a framework for understanding the performance of shared virtual memory protocols. This classification is based on the sharing patterns that are induced by the interaction between the sharing patterns inherent in the algorithm and the coherence granularity imposed by the protocol implementation. My dissertation includes an extensive simulation-based evaluation of LRC and AURC protocols on the Splash-2 application suite using this classification framework for performance understanding. For many applications, the performance under SVM is poor because of the mismatch between the application sharing patterns and protocol coherence granularity. This research led to a new, application-driven, approach towards improving the SVM performance. Rather than taking the applications as they are and trying to adapt the protocols, this approach suggests to restructure the applications based on a good understanding of the protocols. The time of an integrated, multiple-direction research in software shared memory has come.

## 6.1 SVM: Quo Vadis ?

When the first SVM was proposed, no one, not even its author and my advisor, Kai Li, could have predicted that a decade would not be enough to exhaust all the research potential of this field. To be cynical and humorous at the same time, I would say that this is exactly because SVM did not yet succeed in achieving a satisfactory performance, and we should NOT do this if we want to keep the SVM research alive. Of course, this is not what we should do. There is, nevertheless, a privileged beauty of this field, a kind of "fata morgana" in the desert who seems close but then runs away when you try to come nearer.

There are several possibilities for improving the home-based protocols even further. An adaptive home-based approach combined with home migration is a conceivable follow-up inspired by the recently proposed adaptive LRC. The AURC protocol requires further performance evaluation which becomes possible as soon as the performance tools become available for SHRIMP. Feedback from a performance monitor can be exploited by the protocol to migrate homes and perform remapping. At this time, this is impossible because there is no way of measuring the outgoing or incoming automatic update traffic. The tradeoff eager versus lazy invalidation propagation, revived by the co-existence of AURC and Cashmere protocols, must be evaluated on a single but flexible hardware platform that allows selecting the amount of hardware support. In my opinion, for most applications, the performance gap between an eager and a lazy protocol is caused by the difference in the number of coherence control messages and related overhead and not by the difference in communication due to false sharing. This makes the presence (or the absence) of the broadcast support essential in a comparative evaluation of AURC and Cashmere protocols.

Employing multithreading and execution migration in conjunction with data migration has a great deal of potential to further improve the SVM performance. I do believe that new approaches in using scheduling to improve locality and to control memory coherence can reduce the communication and protocol overhead in SVM even further. As the computing platforms remain diversified, I also expect to see an increasing interest in using software shared memory to integrate heterogeneous systems into a coherent computing environment, idea which was anticipated long time ago [ZSLW92].

However, the most productive way to improve protocols and systems remains understanding applications and their protocol interactions, and performance bottlenecks. We should better understand how to structure applications to interact better with software shared memory systems, and to port well in performance across software and hardware shared memory. To achieve this we also need more research effort in the area of performance and debugging tools, parallel software engineering and programming interface design.

In the past, software shared memory was often viewed as a cheaper but lower-performance substitute for hardware-coherent machines. Today, fueled by major advances in system area networks, software shared memory is offered the role of extending the coherent shared address space programming model across sizeable hardware coherent multiprocessor nodes. Combined with adequate hardware and compiler support, application structuring, and performance debugging tools, software DSM has now a real chance to establish a competitive alternative to all-hardware shared-memory parallel platforms.

# Bibliography


[ACD+96] S. V. Adve, A. L. Cox, S. Dwarkadas, R. Rajamony, and W. Zwaenepoel. A Comparison of Entry Consistency and Lazy Release Consistency Implementation. In *Proceedings of the 2nd IEEE Symposium on High-Performance Computer Architecture*, February 1996.

[ACDZ97] C. Amza, A.L. Cox, S. Dwarkadas, and W. Zwaenepoel. Software DSM Protocols that Adapt between Single Writer and Multiple Writer. In *Proceedings of the 3rd IEEE Symposium on High-Performance Computer Architecture*, 1997.

[AH90] S.V. Adve and M.D. Hill. Weak Ordering–A New Definition. In *Proceedings of the 17th Annual Symposium on Computer Architecture*, pages 2–14, June 1990.

[BAB+98] M.A. Blumrich, R.D. Alpert, A. Bilas, Y. Chen, D.W. Clark, S. Damianakis, C. Dubnicki, E.W. Felten, L. Iftode, K. Li, M. Martonosi, and R.A. Shillner. Design Choices in the SHRIMP System: An Empirical Study. In *Proceedings of the 25th Annual Symposium on Computer Architecture*, June 1998.

[BCF+95] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29–36, February 1995.

[BCP92] BCPR Services Inc. *EISA Specification, Version 3.12*, 1992.

[BCZ90] J.K. Bennett, J.B. Carter, and W. Zwaenepoel. Adaptive Software Cache Management for Distributed Shared Memory Architectures. In *Proceedings of* *the 17th Annual Symposium on Computer Architecture*, pages 125–134, May 1990.

[BDFL96] Matthias Blumrich, Cezary Dubnick, Edward Felten, and Kai Li. Protected, User-Level DMA for the SHRIMP Network Interface. In *Proceedings of the 2nd IEEE Symposium on High-Performance Computer Architecture*, February 1996.

[BFRS+94] A.R. Lebeck B. Falsafi, S.K. Reinhart, I. Schoinas, M.D. Hill, J.R. Larus, A. Rogers, and D.A. Wood. Application-Specific Protocols for User-Level Shared Memory. In *Proceedings of Supercomputing '94*, 1994.

[BH90] L. Borrmann and M. Herdieckerhoff. A Coherency Model for Virtually Shared Memory. In *1990 International Conference on Parallel Processing*, August 1990.

[BIMS96] A. Bilas, L. Iftode, D. Martin, and J.P. Singh. Shared Virtual Memory Across SMP Nodes Using Automatic Update: Protocols and Performance. Technical Report TR-517-96, Princeton, NJ, March 1996.

[BIS98] Angelos Bilas, Liviu Iftode, and Jaswinder Pal Singh. Evaluation of Hardware Support for Shared Virtual Memory Clusters. In *Proceedings of the ACM International Conference on Supercomputing*, 1998.

[BISS98] Angelos Bilas, Liviu Iftode, Rudrajit Samanta, and Jaswinder Pal Singh. Supporting a coherent shared address space across SMP nodes: An application-driven investigation. In *IMA Volumes in Mathematics and its Applications, Volume 105, Algorithms for Parallel Processing*, 1998.

[BKP+96] R. Bianchini, L.I Kontothanassis, R. Pinto, M. De Maria, M. Abud, and C.L. Amorim. Hiding Communication Latency and Coherence Overhead in Software DSMs. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[BLA+94] M. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. Felten, and J. Sandberg. A Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer.

In *Proceedings of the 21st Annual Symposium on Computer Architecture*, pages 142–153, April 1994.

[BR90] R. Bisiani and M. Ravishankar. PLUS: A Distributed Shared-Memory System. In *Proceedings of the 17th Annual Symposium on Computer Architecture*, pages 115–124, May 1990.

[BS97] Angelos Bilas and Jaswinder Pal Singh. The Effects of communication Parameters on End Performance of Shared Virtual Memory Clusters. In *Proceedings of Supercomputing'97*, November 1997.

[BZ91] B.N. Bershad and M.J. Zekauskas. Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors. Technical Report CMU-CS-91-170, Carnegie Mellon University, September 1991.

[CBZ91] J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proceedings of the Thirteenth Symposium on Operating Systems Principles*, pages 152–164, October 1991.

[CBZ95] J. B. Carter, J. K. Bennett, and Willy Zwaenepoel. Techniques for Reducing Consistency-Related Communication in Distributed Shared-Memory Systems. *ACM Transactions on Computer Systems*, 13(3):205–244, August 1995.

[CDK+94] A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, and W. Zwaenepoel. Software Versus Hardware Shared-Memory Implementation: A Case Study. In *Proceedings of the 21st Annual Symposium on Computer Architecture*, pages 106–117, April 1994.

[CF89] A.L. Cox and R.J. Fowler. The Implementation of a Coherent Memory Abstraction on a NUMA Multiprocessor: Experiences with PLATINUM. In *Proceedings of the Twelfth Symposium on Operating Systems Principles*, pages 32–44, December 1989.

[DCZ96] S. Dwarkadas, A.L. Cox, and W. Zwaenepoel. An Integrated Compile-Time/Run-Time Software Distributed Shared Memory Systems. In *ASPLOS-VII*, 1996.

[DIFL96] C. Dubnicki, L. Iftode, E.W. Felten, and K. Li. Software Support for Virtual Memory-Mapped Communication. In *Proceedings of the 10th International Parallel Processing Symposium*, April 1996.

[DSB86] M. Dubois, C. Scheurich, and F. Briggs. Memory Access Buffering in Multiprocessors. In *Proceedings of the 13th Annual Symposium on Computer Architecture*, pages 434–442, June 1986.

[DWB+91] M. Dubois, J.C. Wang, L.A. Barroso, K. Lee, and Y-S Chen. Delayed Consistency and Its Effects on the Miss Rate of Parallel Programs. In *Proceedings of Supercomputing '91*, pages 197–206, 1991.

[ENCH96] A. Erlichson, N. Nuckolls, G. Chesson, and J. Hennessy. SoftFLASH: Analyzing the Performance of Clustered Distributed Virtual Shared Memory. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[FAB+96] E.W. Felten, R.D. Alpert, A. Bilas, M.A. Blumrich, D.W. Clark, S. Damianakis, C. Dubnicki, L. Iftode, and K. Li. Early Experience with Message-Passing on the SHRIMP Multicomputer. In *Proceedings of the 23rd Annual Symposium on Computer Architecture*, May 1996.

[GCP96] R. Gillett, M. Collins, and D. Pimm. Overview of Network Memory Channel for PCI. In *Proceedings of the IEEE Spring COMPCON '96*, February 1996.

[GLL+90] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 17th Annual Symposium on Computer Architecture*, pages 15–26, May 1990.

[GMG91] P.B. Gibbons, M. Merritt, and K. Gharachorloo. Proving Sequential Consistency of High-Performance Shared Memories. In *Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures*, July 1991.

[GW88] James R. Goodman and Philip J. Woest. The Wisconsin Multicube: A New Large-Scale Cache-Coherent Multiprocessor. In *Proceedings of the 15th Annual Symposium on Computer Architecture*, pages 422–431, June 1988.

[Her94] S.A. Herrod. *TangoLite; A Multiprocessor Simulation Environment.* Computer Systems Laboratory, Stanford University, 1994.

[HHS+95] C. Holt, M. Heinrich, J.P. Singh, E. Rothberg, and J. Hennessy. The Effects of Latency, Occupancy and Bandwidth in Distributed Shared Memory Multiprocessors. Technical Report CSL-TR-95-660, Stanford, January 1995.

[IBD+98] L. Iftode, M. Blumrich, C. Dubnicki, D.L. Oppenheimer, J.P. Singh, and K. Li. Shared Virtual Memory with Automatic Update Support. Technical Report TR-575-98, Princeton, NJ, February 1998.

[IDFL96] L. Iftode, C. Dubnicki, E. W. Felten, and Kai Li. Improving Release-Consistent Shared Virtual Memory using Automatic Update. In *Proceedings of the 2nd IEEE Symposium on High-Performance Computer Architecture*, February 1996.

[Int91a] $i860^{TM}$ XP Microprocessor. Programmer's Reference Manual, 1991.

[Int91b] Intel Corporation. *Paragon XP/S Product Overview*, 1991.

[Int93] Intel Corporation. *Express Platforms Technical Product Summary: System Overview*, April 1993.

[ISL96a] L. Iftode, J. P. Singh, and Kai Li. Understanding Application Performance on Shared Virtual Memory. In *Proceedings of the 23rd Annual Symposium on Computer Architecture*, May 1996.

[ISL96b] L. Iftode, J.P. Singh, and K. Li. Irregular Applications under Software Shared Memory. Technical Report TR-514-96, Princeton, NJ, February 1996.

[ISL96c] L. Iftode, J.P. Singh, and 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*, June 1996. To appear in Theory of Computing Systems Journal, 1998.

[JJT93] Andrew W. Wilson Jr. Richard P. LaRowe Jr. and Marc J. Teller. Hardware Assist for Distributed Shared Memory. In *Proceedings of 13th International Conference on Distributed Computing Systems*, pages 246–255, May 1993.

[JSS97] Dongming Jiang, Hongzhang Shan, and Jaswinder Pal Singh. Application Restructuring and Performance Portability Across Shared Virtual Memory and Hardware-Coherent Multiprocessors. In *Proceedings of the 6th ACM Symposium on Principles and Practice of Parallel Programming*, June 1997.

[KCDZ94] P. Keleher, A.L. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the Winter USENIX Conference*, pages 115–132, January 1994.

[KCDZ95] P. Keleher, A.L. Cox, S. Dwarkadas, and W. Zwaenepoel. An Evaluation of Software-Based Release Consistent Protocols. In *The Journal of Parallel and Distributed Computing*, volume 29, September 1995.

[KCZ92] P. Keleher, A.L. Cox, and W. Zwaenepoel. Lazy Consistency for Software Distributed Shared Memory. In *Proceedings of the 19th Annual Symposium on Computer Architecture*, pages 13–21, May 1992.

[Kel96] 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.

[Kel98] P. Keleher. On the Importance of Being Lazy. Technical Report UMIACS-TR-98-06, University of Maryland, College Park, 1998.

[KHS+97] L. Kontothanassis, G. Hunt, R. Stets, N. Hardavellas, M. Ciernak, S. Parthasarathy, W. Meira Jr., S. Dwarkadas, and M. Scott. VM-based Shared Memory on Low-Latency, Remote-Memory-Access Networks. In *Proceedings of the 24th Annual Symposium on Computer Architecture*, 1997.

[KS96a] M. Karlsson and P. Stenstrom. Performance Evaluation of Cluster-Based Multiprocessor Built from ATM Switches and Bus-Based Multiprocessor Servers. In *Proceedings of the 2nd IEEE Symposium on High-Performance Computer Architecture*, February 1996.

[KS96b] L. I. Kontothanassis and M. L. Scott. Using Memory-Mapped Network Interfaces to Improve the Performance of Distributed Shared Memory. In

*Proceedings of the 2nd IEEE Symposium on High-Performance Computer Architecture*, February 1996.

[Lam79] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocessor Programs. *IEEE Transactions on Computers*, C-28(9):690–691, 1979.

[LH86] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. In *Proceedings of the 5th Annual ACM Symposium on Principles of Distributed Computing*, pages 229–239, August 1986.

[Li86] K. Li. *Shared Virtual Memory on Loosely-coupled Multiprocessors*. PhD thesis, Yale University, October 1986. Tech Report YALEU-RR-492.

[LJI+98] C. Liao, D. Jiag, L. Iftode, M. Martonosi, and D. Clark. Monitoring Shared Virtual Memory Performance on a Myrinet-based PC Cluster. In *The ACM International Conference on Supercomputing*, 1998.

[LS89] K. Li and R. Schaefer. Shiva: An Operating System Transforming a Hypercube Into a Shared-Memory Machine. Technical Report CS-TR-217-89, Princeton University, April 1989.

[MCL98] T. Mowry, C. Chan, and A. Lo. Comparative Evaluation of Latency Tolerance Techniques for Software Distributed Shared Memory. In *Proceedings of the 4th IEEE Symposium on High-Performance Computer Architecture*, 1998.

[PL93] K. Petersen and K. Li. Cache Coherence for Shared Memory Multiprocessors Based on Virtual Memory Support. In *Proceedings of the IEEE 7th International Parallel Processing Symposium*, April 1993.

[PL94] K. Petersen and K. Li. An Evaluation of Multiprocessor Cache Coherence Based on Virtual Memory Support. In *Proceedings of the IEEE 8th International Parallel Processing Symposium*, April 1994.

[PR94] R. Pierce and G. Regnier. The Paragon Implementation of the NX Message Passing Interface. In *Proceedings of the Scalable High-Performance Computing Conference*, May 1994.

[SB97] E. Speight and J.K. Bennett. Brazos: A Third Generation DSM Systems. In *USENIX Workshop on Windows-NT*, 1997.

[SB98] E. Speight and J. Bennett. Using Multicast and Multithreading to Reduce Communication in Software DSM Systems. In *Proceedings of the 4th IEEE Symposium on High-Performance Computer Architecture*, 1998.

[SBIS98] 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 Symposium on High-Performance Computer Architecture*, 1998.

[SD87] C. Scheurich and M. Dubois. Correct memory operation of cache-based multiprocessors. In *Proceedings of the 14th Annual Symposium on Computer Architecture*, pages 234–243, June 1987.

[SDH+97] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, and M. Scott. Cashmere-2L: Software Coherent Shared Memory on a Clustered Remote-Write Network. In *Proceedings of the Sixteenth Symposium on Operating Systems Principles*, 1997.

[SFL+94] I. Schoinas, B. Falsafi, A.R. Lebeck, S.K. Reinhardt, J.R. Larus, and D.A. Wood. Fine-grain Access for Distributed Shared Memory. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 297–306, October 1994.

[SG97] D. Scales and K. Gharachorloo. Towards Transparent and Efficient Software Distributed Shared Memory. In *Proceedings of the Sixteenth Symposium on Operating Systems Principles*, 1997.

[SGT96] D.J. Scales, K. Gharachorloo, and C.A. Thekkath. Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[SHG92] J.P. Singh, J.L. Hennessy, and A. Gupta. Implications of Hierarchical N-Body Methods for Multiprocessor Architecture. Technical Report CSL-TR-92-506, Stanford University, 1992.

[SISL96] R. Samanta, L. Iftode, J.P. Singh, and Kai Li. Home-based SVM Protocols for SMPs. Technical report tr-535-96, Princeton University, Princeton, November 1996.

[SMV95] Savic S., Tomasevic M., and Milutinovic V. Improved RMS for the PC Environment. *Microprocessor Systems*, 19(10):609–619, December 1995.

[SWG92] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared Memory. *Computer Architecture News*, 20(1):5–44, 1992. Also Stanford University Technical Report No. CSL-TR-92-526, June 1992.

[TD92] Roger Traylor and Dave Dunning. Routing Chip Set for Intel Paragon Parallel Supercomputer. In *Proceedings of Hot Chips '92 Symposium*, August 1992.

[TL94] Ch. A. Thekkath and H.M. Levy. Hardware and Software Support for Efficient Exception Handling. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 110–121, October 1994.

[WG89] W. Weber and A. Gupta. Analysis of Cache Invalidation Patterns in Multiprocessors. In *The Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 243–256, April 1989.

[WOT+95a] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. Methodological Considerations and Characterization of the SPLASH-2 Parallel Application Suite. In *Proceedings of the 23rd Annual Symposium on Computer Architecture*, May 1995.

[WOT+95b] Steven Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, June 1995.

[YKA96] D. Yeung, J. Kubiatowicz, and A. Agarwal. MGS: A Multigrain Shared Memory System. In *Proceedings of the 23rd Annual Symposium on Computer Architecture*, May 1996.

[ZIL96] Y. Zhou, L. Iftode, and K. Li. Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Virtual Memory Systems. In *Proceedings of the Operating Systems Design and Implementation Symposium*, October 1996.

[ZIS+97] Y. Zhou, L. Iftode, J.P. Singh, K.Li, B.R. Toonen, I.Schoinas, M.D. Hill, and D.A. Wood. Relaxed Consistency and Coherence Granularity in DSM Systems: A Performance Evaluation. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1997.

[ZSB94] M.J. Zekauskas, W.A. Sawdon, , and B.N.Bershad. Software Write Detection for a Distributed Shared Memory. In *Proceedings of the Operating Systems Design and Implementation Symposium*, pages 87–100, November 1994.

[ZSLW92] S. Zhou, M. Stumm, K. Li, and D. Wortman. Heterogeneous Distributed Shared Memory: An Experimental Study. *IEEE Transactions on Parallel and Distributed Computing*, 3(5):540–554, May 1992.