M.ASc. Thesis, Department of Computer Science, University of Toronto.

Region-Oriented Main Memory Management in Shared-Memory NUMA Multiprocessors

by

Benjamin Gamsa

TR-92-10-01

Department of Computer Science
University of Toronto
Toronto, Ontario, Canada, M5S 1A4
email: ben@cs.toronto.edu

Abstract

The need to achieve higher performance through greater degrees of parallelism necessitates distributing the memory throughout a multiprocessor system to reduce contention and increase scalability. Unfortunately, such Non-Uniform Memory Access time (NUMA) multiprocessors introduce complications for the programmers, who must now be concerned with the physical distribution of their data in order to extract good performance from the system.

The impact of remote memory accesses can be reduced through replication and migration, either in processor caches or in main memory. Unfortunately, the effectiveness of caches is limited for large data sets due to capacity misses, while dynamic virtual memory page management suffers from a mismatch between the pages being replicated and the data structures in programs.

In this thesis we propose that data be partitioned into Shared Regions reflecting the granularity of data sharing in programs, and that special synchronization calls be added to enforce proper ordering of accesses to the shared data, as well as to manage replication and consistency transparently to the programmer. A number of different memory coherence protocols, replication strategies, and data distribution schemes are proposed, implemented, and evaluated on a 16-processor NUMA multiprocessor. A model is also developed to help estimate performance on larger systems of 64 and 128 processors.

Results from the measurements and the model indicate that the Shared Regions approach is successful in obtaining the necessary locality critical to performance, while incurring only minimal overhead. Data distribution methods are also observed to have a significant impact on the performance of the system, especially in the larger multiprocessors modeled. However, it is possible for excessive replication to occur and result in performance degradation. This suggests that a choice of replication and coherence policies is required in order to correctly match the policies to the expected access behaviour of the program.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

When seeking performance and programmability in parallel computing, shared memory multiprocessors are enticing because of their promise of high performance and a familiar programming environment.

Unfortunately, it has been discovered that it is difficult to obtain both. In particular, shared memory machines with a single global memory, often known as Uniform Memory Access (UMA) multiprocessors, have failed to scale beyond a few dozen processors, so most newer machines have a global *logical* memory space, but physically distribute the memory modules around the system. As a result, the time to access memory depends on the relative position of the memory module and the accessing processor, creating what is called a Non-Uniform Memory Access (NUMA) multiprocessor. Although this now allows the system to scale well, it creates complications for the programming environment, for although it is possible to treat memory uniformly, ignoring its physical distribution generally leads to poor performance.

A common approach to this problem, used initially in UMA multiprocessors, is the use of processor caches. Unfortunately, keeping the caches consistent generally requires extensive hardware support, and it is an open question whether such systems are truly scalable. Alternatively, caches may be kept consistent by a layer of software between the operating system and the application. This removes many of the existing limitations of hardware cache-coherent systems, but the effectiveness of such techniques remains to be shown.

Although caches can help alleviate the cost of remote memory accesses by reducing their frequency, when cache misses do occur they can still be expensive. For many problems, the size of the cache is insufficient to hold the working set, resulting in large numbers of capacity misses. In such cases, locality in main memory becomes critical, and arranging data to be near the processors that most frequently access it can improve performance substantially.

This last point is the main focus of this thesis. We are concerned with methods for improving locality in a way that adapts to the dynamic needs of the application. Although techniques based on dynamic page-based data migration and replication have proven somewhat effective, they are limited by their reliance on the memory management unit to help maintain consistency, which can only provide protection at the granularity of an entire page. Fixed-sized pages rarely match the data abstractions of the application, resulting in a mismatch between coherence of the desired objects in the application and the page-level coherence provided by the operating system. This conflict makes page-based techniques unsuitable for small grained or irregular data structures and access patterns.

The approach taken in this thesis is to move a minimum amount of the burden of coherence management onto the programmer, where knowledge about the data structures and the sharing patterns among the parallel activities exists, and spread the rest of the work between the run-time system and the operating system.

More specifically, the user partitions the data according to the natural units of sharing in the algorithm. These units are called *shared-regions*. The user then uses special synchronization calls to control access to shared-region. Equipped with knowledge of the granularity of sharing and the points in the program where such sharing implicitly occur, the system can manage locality on arbitrary units of data and can better respond to the sharing patterns exhibited by the applications.

The intent is to provide a balance of locality and flexibility, for both main memory and caches, using a combination of programmer and system software support.

1.1 Region-Oriented Memory Management: A Preview

For many parallel applications, locality in main memory can provide significant performance advantages. Not only does it reduce the time for memory accesses because of the shortened distance to memory, but it also reduces these costs by lowering the contention in the network and at the memory modules.

Unfortunately, for problems with unpredictable sharing patterns and small units of coherence (perhaps only a few hundred bytes), traditional page-based migration schemes have proven to be ineffective.

To deal with such limitations, we use a technique first proposed by Sandhu, Gamsa, and Zhou [SGZ92], originally for managing cache coherence. Shared data in an application is partitioned into *shared-regions*, according to the natural unit of sharing as dictated by the parallel structure of the algorithm. Each set of accesses to each shared region is then bracketed by shared-region calls that perform synchronization and maintain coherence for that particular region. Because both the synchronization points and the data being accessed are linked together, full flexibility for managing coherence and locality is available to the system.

We discuss the general requirements for coherence, and present two coherence algorithms. The first algorithm ensures consistency by updating all known copies of a region when the region is released. Since all processes wishing to access the region must first acquire permission to do so, all copies can be updated before permission is granted, thus guaranteeing full coherence. The second algorithm validates the local copy of the region before allowing the process to acquire access to it. If the region is stale, a fresh copy is retrieved. All the information needed to maintain coherence is stored in a region descriptor associated with the data.

An implementation, based on the 16 processor NUMA multiprocessor Hector and its operating system Hurricane, is presented for these two algorithms. Since Hector does not have hardware cache coherence, the software cache coherence schemes discussed in [SGZ92] are used [1]. To take advantage of the underlying page-based Memory Manager which provides transparent replication at the page-level only, some restrictions on the shared-region coherence options are required, although the basics of the two algorithms are still implemented.

[1]We expect that the techniques presented in this thesis would be effective on systems both with and without hardware cache-coherence.

Further work is still required to obtain a better integration of shared-region coherence and operating system memory management.

The shared-region implementation is evaluated on Hector using some synthetic programs patterned after real applications. The results show increased performance when the reuse rate is high enough to amortize the cost of the coherence overhead. The use of small regions or regions with little reuse show degraded performance using shared-region coherence. Additionally, in cases where the processor cache is effective in hiding memory access costs, shared-region replication tends to reduce performance due to the unnecessary memory copying. The non-coherence-related overheads of shared-regions show little impact on performance. Further experimentation on a modified version of Hector designed to simulate a larger system shows improved benefits from locality and the use of shared-region memory coherence. However, increased sensitivity to the application's access patterns is also observed.

Finally, a model of the software and hardware system is developed to help evaluate the performance of shared-regions for larger problems and larger systems. The results indicate that the need for locality grows considerably with system size, and that more sophisticated techniques are required, both in the shared-region implementation and in the application, to obtain good performance on larger systems.

1.2 Thesis Organization

We begin with a discussion of related work in Chapter 2. This serves both to provide some background in the field and to motivate the work that follows. We attempt to cover a broad spectrum of ideas related to reducing the impact of communication in shared-memory systems, in order to make clear some of the other options that are available.

In Chapter 3, a brief overview of the hardware and software environment used to conduct our experiments is presented. This provides a context for the algorithms and implementations discussed in later chapters, and provides some motivation for the direction of our research.

Chapter 4 presents the algorithms and implementations of shared-regions, and discusses their relationship to the existing hardware and system software.

Chapter 5 provides results from our measurement experiments. Low-level hardware and system software measurement results are provided, followed by performance evaluations of some synthetic applications using shared-regions.

A modified version of the multiprocessor Hector, used to simulate a larger system is used in Chapter 6 to evaluate the impact of higher communication costs on locality and shared-regions. This is followed by a model of the hardware and software which is then used to further evaluate the effects of larger systems on shared-region performance.

Finally, in Chapter 7 the main results of the thesis are summarized, and future extensions and alternative directions for investigation are proposed.

Chapter 2

Related Work

This thesis is concerned with the benefits of main memory locality (achieved through dynamic replication of non-page-based regions) for the purpose of reducing average memory latency and thus improving shared-memory multiprocessor performance. As such, we are interested in the different approaches taken by researchers working to reduce the impact of memory latency. Although not all techniques are directly relevant to our work, they provide an important framework of alternative approaches essential to a proper evaluation of the techniques we introduce.

Researchers in parallel computing typically take one of two approaches: they either start with a programming paradigm and build a machine that can best implement that paradigm; or they start with a machine and find the best paradigm for programming it. The first group generally consists of hardware engineers looking to build an improved multiprocessor by attacking some portion of the scalability problem. The second group of researchers typically already have a machine, and want to determine how to get the highest performance with the least effort from the system. In addition, there is a third group that seeks to best understand the relationship between future hardware and software systems by analyzing the fundamental costs of hardware designs and software algorithms.

This chapter explores the work performed by these three groups in dealing with the problem of communication in a shared-memory environment. Three basic techniques are studied:

1. Reducing the cost of communication.

2. Hiding the cost of communication.

3. Reducing the frequency of communication.

2.1 Hardware

In the area of hardware design of shared-memory multiprocessors, all three of the previously mentioned techniques for reducing the impact of communication are actively being pursued.

2.1.1 Reducing the Cost of Communication

There is ongoing research in a large number of areas related to improving the speed of communication. Faster and wider network links, more redundant paths (to reduce network contention), more buffers at the network junctures (to reduce the impact of contention), and higher bandwidth and faster memory subsystems (interleaved and cached memories), are all popular techniques for reducing the cost of communication and memory accesses in a shared-memory environment. However, this is a very broad field, and only tangentially related to this thesis, in that the only the costs and benefits of replication and remote memory accesses are altered. Therefore, nothing more will be said on this topic.

2.1.2 Hiding the Cost of Communication

There are many techniques for hiding the cost of communication in the hardware design of a shared-memory multiprocessor, but they all come down to some method of overlapping communication with computation.

A popular technique in recent years has been the use of pipelining and buffering of memory accesses. This allows the latency of memory accesses to overlap each other as well as overlapping other independent instructions. However, because these techniques may violate sequential consistency [Lam79], several models of weaker consistency have been proposed [GLL+90, AH90, DS90] that allow these optimizations, provided that the programmer adhere to a stricter model of programming. These new contracts between the software and the hardware require that all accesses to shared data be controlled by hardware-visible locks that allow the hardware to enforce the stricter consistency requirements only at those points where it is needed (only where there is a chance that the data may be shared).

Another method gaining favour is pre-fetching of data. The two most common approaches are binding prefetching, where the compiler must ensure that the data will not change between the prefetch and its use [GGV90], and non-binding prefetching, where the data is brought into the cache, but may still be invalidated under the normal cache coherence protocol thereby ensuring correctness even if the data is changed before being used [MG91]. Key issues include methods to automatically prefetch data (compiler and hardware technology), ensuring that the data prefetched will actually be used, and ensuring that the extra traffic due to prefetching does not adversely affect the performance of the rest of the system.

Two other interesting techniques for hiding communication are multi-threaded CPUs and dynamically scheduled CPUs. In multi-threaded CPUs, multiple threads may be active in the processor concurrently, taking turns either at regular intervals [Smi81, ACC+90], or on cache-misses [ABHKK90]. In dynamically scheduled CPUs, micro-parallelism in a single thread is used to hide the latency of memory operations [GGH92].

A good overview of the above techniques and their relative impact on performance can be found in [GGH91].

2.1.3 Reducing the Frequency of Communication

The most popular method for reducing memory access costs is through caching. This brings the data closer to where it is needed so that future accesses will not pay the high cost of remote accesses. Caching also helps by indirectly prefetching data when using large cache blocks.

For small-scale bus-based systems, a number of protocols (snoopy cache protocols) take advantage of the broadcast nature of the bus [KEWP85, AB86, TS87]. Various permutations on these protocols have been developed, but indications are that there is no single optimal point [EK88].

Because of the limited scalability of bus, and of any cache coherence protocol that depends on broadcasts, a number of protocols that depend only on point-to-point links have been developed. Directory protocols are a class of such protocols that keep a list of referencing caches for every block of memory [Tan76, CF78, LLG+90]. Because these can require large amounts of memory for large systems, follow-on work has focused on reducing the space cost of directories [CKA91, ON90]. Another alternative, used by the IEEE standard Scalable Coherent Interface (SCI) protocol [JLGS90], is to dynamically link together the cache lines holding the same memory block. A survey of the different caching strategies is presented in [Ste90].

One of the problems that affects all caching strategies (hardware and software) is false sharing. This occurs when a cache line appears to be actively write-shared by two or more processors, but in actual fact, the data each processor is referencing is a disjoint subset of the cache line. This has been studied in a number of papers [EK88, EK89, WG89], and shown to be responsible for a large percentage of the coherence traffic. Additionally, with the trend towards larger cache line sizes to amortize the cost of the communication and protocol overheads, the problem is likely to become worse. There have been proposals for dynamically adaptable cache line sizes [DL92], but the complexity is quite high. Other work has focused on possible restructuring techniques that can be performed by the compiler to help eliminate false sharing [TLH90, EJ91]. The issue is still open, however.

One other approach to improving the effectiveness of caching is to treat all of main memory as a large, slow cache, and use a limited broadcast method to locate memory blocks as needed [BIFKR92, WH88]. This eliminates the problem of limited cache size and capacity misses and can be cheaper by using slower memory, but, because of the high costs of misses, its benefits with respect to the above techniques are unclear [SJG92].

Finally, it should be noted that all of the above techniques are complex and expensive, and their scalability to machines with hundreds of processors is unproven.

2.1.4 Interaction with Software

A general trend appearing in many of the newer hardware designs is a closer cooperation between the hardware and the software. This growing relationship is an attempt to increase the semantic content of the program that is visible to the hardware, thus allowing more leeway in hardware optimizations and the ability to act more directly according to the needs of the program.

Exposing synchronization is important for several optimization as we have already seen. With minor modifications to the hardware, it may be possible for the software to more directly participate in cache coherence with potentially improved performance at lower cost [CKM88, HLRW92]. Additionally, scheduling of threads in multi-threaded architectures may benefit from closer ties with the compiler [BR92].

2.2 Software

From the software perspective, there are many ways of dealing with the impact of communication when using the shared-memory paradigm (the one that we are interested in here). For the most part these options are constrained by the hardware available and the method chosen for programming these systems.

The two hardware platforms that have the biggest impact on the choice of software techniques are:

- message passing systems (multiprocessors or networked workstations), which imply some form of distributed shared memory (DSM), and
- shared-memory multiprocessors (primarily those without cache coherence, or in some cases without caches altogether).

Of the three techniques previously described for reducing the impact of communication (reducing the cost, hiding the cost, and reducing the frequency), only the last two apply to software solutions, and in the realm of shared-memory programming the primary emphasis is on reducing the frequency of communication.

Because of the strong impact of the different hardware architectures on the software solutions, the breakdown for this section will be based on the message passing hardware versus shared-memory hardware.

2.2.1 Shared Memory in Message Passing Systems

As already mentioned, shared-memory programming on message passing systems generally takes the form of Distributed Shared Memory (DSM, also known as Virtual Shared Memory, VSM, or Distributed Virtual Shared Memory, DVSM). The technique has been used both on networked workstations [LH86] and message passing multiprocessors such as hypercubes [LS89]. It has many similarities to hardware cache coherence: main memory is treated as a cache, each page is managed like a cache line, and the memory management unit is used to trap misses and writes.

The problems that need to be addressed are similar to those in hardware cache-coherent systems: reducing the number of messages, dealing with false sharing, and reducing the impact of cache misses (coherence page faults in DSM systems). However, the relative size and speed of the various components, as well as the flexibility of using software, has a strong influence on the applicability of various solution techniques.

Following the initial work to prove the viability of DSM, a number of different protocols were examined, based on their overhead and the number of messages required in order to perform certain common operations (finding a page, invalidating or updating copies), as well as the load placed on the network. Stumm and Zhou [SZ90] examined several different techniques, but none were found to be superior in all cases.

The relationship between data coherence and communication, and the weaker forms of coherence that can be built on this understanding (as previously discussed with respect to hardware cache coherence) has been used by a number of groups in Distributed Shared Memory to attack both the false sharing problem as well as the high cost of communication.

In Clouds [RAK89], a distributed object-oriented system, synchronization and coherence are combined into a single protocol, with data being both locked and retrieved coherently with a single message. This reduces the overall communication requirements of the applications, improving performance.

Munin [CBZ91] takes this one step further by supporting a number of different protocols, including a variant of release consistency that allows multiple processors to write to a page provided they are writing to disjoint data. They accomplish this by saving the original copy on each processor that is writing, and sending deltas when they are done. System visible synchronization primitives are used to mark the points where the various copies must be merged. The choice of coherence protocol is made by the programmer by annotating the data with extra type information. The type information reflects the expected sharing pattern. Types that are supported include read-only, migratory, producer-consumer, reduction, concurrent shared writers, and result. Each of these has a specially tuned coherence protocol to support it. For instance, for the producer-consumer type, the system dynamically determines who the producer and consumers are, and automatically sends data that has been written by the producer to the consumers, before the data is actually referenced by them. This flexibility helps eliminate many of the delays incurred when handling coherence faults, and reduces the number of messages required.

A distributed object system is presented in [FL92], that attempts to address the problems DSM systems have when dealing with domain decomposition problems. They propose three components to solve the problem: immutable versioned objects that allow a writer and a reader to concurrently access an object by using a version number to distinguish between the different copies; object fragments that provide a finer granularity of coherence; and a unification of synchronization, data, and coherence to reduce communication and allow coherence to operate on a finer granularity. The use of this technique, however, requires more work on the part of the programmer to perform the partitioning and explicitly select the correct object version. Benefits are similar to the previously mentioned techniques: reduced cost of synchronization, eager sending in producer-consumer relationships, and elimination of the false sharing problem.

A similar technique is described in [BCaCR92], where the data is partitioned, and coherence is maintained through explicit synchronization calls.

Finally, in [KCZ92], another form of consistency, termed "lazy consistency" is used to reduce even further the number of messages required to support coherence. With their technique, the synchronization calls are used to build up a dynamic causal graph that indicates which processors expects to see what changes from which other processors. The differences between the copies are propagated at each synchronization point along with the causal graph, allowing any processor to reconstruct the data as it should see it, based on the causal links formed by the synchronization that has occurred. The goal is to reduce the number of messages sent to only those strictly required by the synchronization relationship among the processors.

2.2.2 Shared Memory in NUMA multiprocessors

Many of the techniques that were applied to hardware cache-coherent systems and DSM systems can also be applied to NUMA multiprocessors. Because of the high cost of memory accesses due to long latencies and contention, replicating data to closer memory modules can significantly reduce the average cost of memory references. Additionally, synchronization and the potential benefits of prefetching also play a role in non-cache-coherent NUMA multiprocessors.

The primary difference, however, is that with main memory caching through replication and migration in NUMA multiprocessors, there is always the option of remotely accessing the data rather than copying it locally. This option avoids the cost of replication and coherence, and changes the balancing point between the benefits and costs of locality, when compared to the previous environments. Additionally, because communication is much less expensive in shared-memory multiprocessors than in message passing systems, it is harder to justify the software complexity present in many DSM systems. In many cases, it is best just to leave a page frozen in place and access it remotely, rather than expending the effort to bring it closer. As with the DSM systems, false sharing is a major concern, although the option of freezing a page lessens its impact (ping-pong-ing of pages can be avoided).

Cox and Fowler present a directory-based scheme for replicating and migrating pages to local memory modules [CF89]. A simple invalidation-based cache coherence protocol is used. The goal is to make the system appear to be a uniform memory access (UMA) system, despite the fact that the BBN Butterfly that they use has a remote-to-local access time ratio of approximately 15. In function, it is similar to the DSM systems, with the exception that pages that bounce around too frequently are frozen in a single spot. A simple timestamp per page is used to determine if faults on a page are occurring too close together, in which case the page is frozen. To avoid a page being frozen unnecessarily and slowing down the program, a daemon periodically defrosts the pages to allow the system to re-assess its earlier decision. They discuss the importance of the BBN block memory move operation, and suggest that such a feature is critical for a system such as theirs to be effective. They report good performance but provide no basis for comparison.

Similar work was conducted by Bolosky, Scott, and Fitzgerald on the IBM ACE multiprocessor [BFS89]. The architecture of the ACE, which has separate local and global memory, led to a slightly different policy to control page thrashing, but the results were similar. Pages are either replicated or migrated to local memory on access, or they are frozen in the global memory if the total number of migrations reaches a preset limit. Although they recognize the importance of false sharing as the next step towards improved performance, they only briefly discuss the possibilities of data segregation and access-pattern information to aid the system.

A joint follow-up paper by the two previous groups examined the impact of their different hardware platforms on the choice of coherence protocols [BSF+91]. The overall conclusion was that simple policies, chosen to match the architecture, could function near-optimally, but that ignoring these architectural features would result in very poor performance. Once again, the significance of false sharing as the most important problem facing these systems was restated.

LaRowe and Ellis present experimental results based on a memory management scheme similar to the above, using a wide variety of coherence protocols (all implementing sequential consistency) and a wide variety of applications [JE91], The coherence protocols vary in how quickly pages are frozen and defrosted, and the types of triggers used to cause replication and migration. The general conclusion is that once again, it is difficult to choose a "best" protocol, and that although many of the applications show significant improvement using these techniques, their performance does not match the versions specially tuned to the architecture.

In a subsequent paper, LaRowe, Ellis, and Kaplan report on their experience porting the memory management policies from the original BBN butterfly, to the BBN TC2000 [JEK91]. Despite the lower remote/local access ratio of the new machine and its lack of a special block-memory move operation, the results indicate that similar policies work well on both systems, contrary to the expectations of the authors, and although a parameterized policy that could be tuned to each application improved performance, its effects were only felt at the extremes of the parameter space.

Unrau, Stumm, and Krieger make a case for a clustered approach to memory management design and policy [USK92]. In particular, they note that for hierarchically structured multiprocessors, the replication and migration strategy should reflect the architectural hierarchy, by using different policies within clusters and across clusters. For instance, replication may be used across clusters, but, within a cluster, only a single copy may be maintained, taking advantage of the low remote/local access time ratio within a cluster.

In all of these systems, the impact of caches (non-coherent) are not discussed, primarily because they were lacking in many of the older machines. However, when tied to the coherence policies, caches can reduce average latency by caching read-only or single writer pages, thus reducing the need for replication and migration. A re-examination of the effectiveness of such policies under these new conditions may well alter previous results.

In order to supplement the results from experimental work, several groups have tried analytic approaches.

Bolosky and Scott investigate the performance of various page-coherence strategies by comparing them to the optimal policy calculated off-line [BS92]. Although they demonstrate the benefits of such an approach to help understand the performance of the different policies, they present no concrete results from the use of this technique.

Similarly, LaRowe, Holliday, and Ellis present an MVA model of memory management policies and use it to explore the space of options and the impact that policy errors can have on performance [JHE92]. Although it does reinforce the intuitive notion that concurrently write-shared pages pose the most difficulty, little more was concluded.

Black, Gupta, and Weber present some results on competitive online algorithms for page migration and replication [BGW89]. Although optimal for online algorithms, they are still a factor of 2–3 from the offline algorithm, and it is questionable whether they provide any advantage over the heuristics used by the systems we have already discussed.

Finally, although mostly neglected in NUMA memory management work, closer ties between the application and the memory management system seem essential in dealing with the problems of false sharing, page thrashing, and premature page freezing.

LaRowe, Wilkes, and Ellis discuss the importance of system-program interaction and point at the benefits of even simple techniques such as notification of phase changes in applications [JWE91]. This can be used to reevaluate past decisions, and removes the need for the expensive defrost daemon. A similar technique was also discussed by Unrau, Stumm, and Krieger [USK92].

A method of segregating data to avoid false sharing among different objects is discussed by Adema and Ellis [AE91]. Type information similar to that in the Munin system is used to ensure that different options with different access patterns are not co-located on the same page. The intention is to have the user do the partitioning, and let the system manage locality.

Malkawi and Patel discuss techniques for allowing the compiler to direct memory management policies in a uniprocessor environment [MP85]. Their results show that knowledge of the access patterns can be used to better manage the memory hierarchies in a uniprocessor. Similar policies would likely benefit multiprocessors, particular in the face of replication with large local working sets.

2.3 Application Characterization

In this section we look at approaches used to characterize communication and sharing patterns in applications and the impact such communication has on multiprocessor performance.

Eggers and Katz show a relationship between the sharing patterns of the applications and the performance of the different hardware cache coherent protocols [EK88]. Weber and Gupta [WG89] divide the different types of sharing into 6 categories: read-only, migratory, synchronization, read mostly, and concurrent read-write sharing. This is extended further by Bennet, Carter and Zwaenepoel in [BCZ90b, BCZ90a], who add producer-consumer and result types to the list. The reason for the differences are that Weber and Gupta are looking at cache coherent hardware, while Bennet, Carter and Zwaenepoel have distributed shared memory in mind. However, the final results can be applied to both fields to understand the needs of the applications.

A more abstract model of locality and communication is developed by Johnson [Joh92], who uses a combined model of the application, the processor, and the network, to gain insight into the impact of communication on applications, without concern for the specifics of the architecture or the application. The general conclusion is that the improvements due to locality are at most linear in the average communication distance saved; the effect of contention levels off for large numbers of processors, so that the per-hop latency approaches a constant value. Although this would seem to limit the benefits of locality, it does assume a perfectly uniform reference distribution. Thus with an imbalance in communication, the benefits could be greater.

Sevcik and Zhou develop a model of a NUMA multiprocessor and used it to evaluate how applications might perform on such a system, as compared to a hypothetical UMA machine [SZ91]. The results show that if both locality and the sharing patterns are considered, then comparable performance on a NUMA multiprocessor is possible.

Finally, a number of papers have examined individual applications to assess their needs in terms of locality. Singh and Hennessy report that there is a strong impact on performance due to communication, and that taking advantage of the inherent locality in the algorithm (a model of ocean currents) is critical to achieving good performance [SH92]. In another study, this time of the N-body problem, Singh, Hennessy, and Gupta once again find locality present in the algorithm and important to performance, and comment on the benefits of coherent caches in taking advantage of this locality [SHG92]. On the other hand, Lin and Snyder present preliminary findings that using a message passing programming style, even on a shared-memory machine, results in better performance because of the locality that comes from such an approach [LS90]. Similar results on the tradeoffs between message passing and shared memory can be found in [PL92, MG89, LeB86]. However, they all appear to agree that locality to help reduce communication is important to good performance on all the systems.

Chapter 3

System Overview

This section provides an overview of the shared-memory multiprocessor Hector, and its operating system Hurricane. These two systems form the foundation on which the rest of this thesis is based. As such, our goal in presenting this information is to aid the reader in understanding the motivations behind our work, and to present the necessary background information required to understand the implementation and experimental details presented in subsequent chapters.

3.1 Hector

Hector [VSWL90] is a hierarchical, general-purpose shared-memory multiprocessor. It consists of a set of processor-memory (PM) pairs connected together by a bus (called a station), several buses connected together by a local ring, and several local rings connected together by a global ring (see Figure 3.1). The rings are uni-directional bit-parallel slotted rings. All levels in the interconnection network are capable of transferring up to 64 bits of data in parallel. The architecture can scale up to several hundred processors.

Hector provides a single global physical address space. Each memory module contributes one portion of the global memory, and can be accessed directly by any processor. Access time to memory increases with distance. Since all accesses within a subtree of the hierarchy (a station or local ring) are less expensive than accesses outside the subtree, a natural hierarchical notion of locality emerges. The notion of "degree of locality" is therefore quite natural and explicit in this architecture. Although every processor has a cache, the hardware does not provide cache coherence at any level in the system.

Because of the relatively slow memory and fast interconnection network, the NUMAness in Hector is quite mild. It requires approximately 20 cycles to read a cache line from local memory, but only 1 more cycle per ring segment to read from a remote memory. In a large system this might add a maximum of 30 to 40 cycles to the memory access time. In the smaller prototype system used, the difference is only about 10 to 15 cycles.

Since the cache access time is only 1 cycle, caching has a much stronger influence on performance than the cost of remote accesses; the factor of 2 or 3 difference for remote memory is swamped by the difference between local memory and cache.

The prototype used for the measurement experiments reported in this thesis is configured as follows:

- The entire system runs at 16.67Mhz (60ns cycle time).

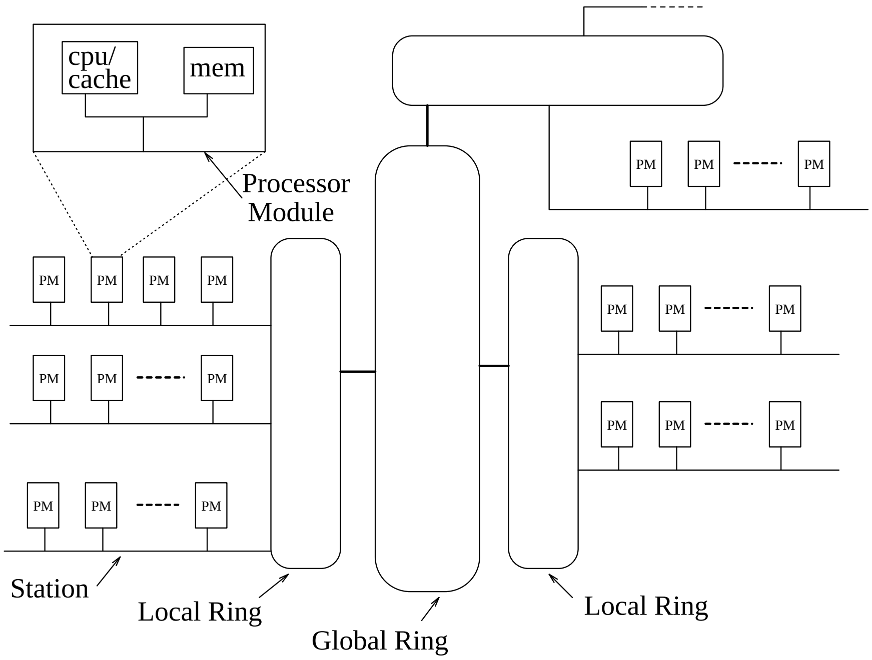


Figure 3.1: General architecture of Hector.

- Each processor board contains a Motorola 88100 cpu, a 16 KB instruction cache, a 16 KB data cache, and 4 MB of globally addressable memory. Pages are 4 KB.
- There are a total of 16 processors.
- There are 4 processor boards per station, and 4 stations per local ring.
- There is no global ring.

Because of the low remote access cost, the benefits of memory locality in an architecture such as Hector's are not clear. Therefore, any benefit of memory replication observed is real.

3.2 Hurricane

Hurricane [FSUZ88, BSU89] is a general purpose, multiprocessor, multi-tasking, and multi-user operating system. It is modular in design and structure. It consists of a minimal kernel that provides primitives for Remote Procedure Calls (RPC), plus virtual memory services. The rest of the operating system consists of servers communicating through RPC calls and shared memory.

A program in Hurricane consists of an address space, and one or more processes executing in the address space (on one or more processors). All of the resources of an address space are shared by all of its processes.

Scheduling in Hurricane is still quite rudimentary, but sufficient for our purposes. Processes are either allocated round-robin around the system, or placed explicitly on each processor. Once assigned, processes generally do not migrate. It is also possible to reserve processors for exclusive use by a program, preventing interference from other programs.

The Hurricane memory manager [Unr92] supports traditional page-based virtual memory. Every program is provided with an independent linear virtual address space partitioned into address ranges mapped to files. Each range of addresses is known as a Hurricane region (not to be confused with the shared-regions of this thesis). Each Hurricane region has its own set of attributes that control its allocation and coherence policies.

Allocation policies supported include First-Hit (each page is allocated from the processor that first touches it), Round-Robin (pages are allocated in a round-robin fashion from around the system), and Fixed (all pages are taken from a single chosen physical memory module). Beyond these controls, the notion of the physical location of memory is completely hidden from the application by the virtual memory layer.

Several different coherence strategies are supported; both for main memory and for the cache. For main memory, page migration and replication options are provided. In addition, the degree of replication and migration can be controlled, so that page movement only occurs for accesses outside of the local cluster (where a cluster might be a station, a group of stations, or a ring). This provides a compromise between the benefits of locality and the cost of moving pages. Excessive migrations and replications are currently controlled by a simple counter per page that limits the number of migrations. A reset call is provided to reset the counter to zero.

For the cache, a simple page-based coherence protocol is provided by default. Caching is enabled when a page has either reader(s) or a single writer, but when a page has multiple readers and writers (and migration is not in effect), it becomes uncachable for all processors. This was a major source of performance problems in many of the earlier programs, as false sharing and sequential read/write sharing caused large portions of memory to become uncachable. An alternative policy handles the multiple readers/writers state by migrating the mapping rights: all current mappings are removed and the new processor is mapped in as a single reader or writer, thus allowing the page to be cached. As long as the page is not being concurrently write-shared, this maintains the cachability of the page without the cost of full migration. As with page migration and replication, a counter limits the total number of mapping migrations, and can be reset, along with the coherence state of the page, by a reset call.

A final policy, on which this thesis is based, removes all coherence controls on the cache and main memory from the operating system. When this policy is in effect, no effort is made by the operating system to maintain the consistency of the caches or main memory replicas for any pages in the specified Hurricane region. Instead, calls to operate on the cache and main memory copies are provided, allowing the programmer complete freedom in managing coherence.

The operating system is still undergoing development. Structuring issues related to scalability and locality are being investigated [Unr92], and significant improvements in the performance of the memory manager have been observed. Unfortunately the version used for this work only contains a portion of the enhancements being investigated, and hence does not perform as well as it could.

3.3 Programming Support

There are a number of different ways to implement a parallel program on Hurricane. The most direct way is to create a number of Hurricane processes, and have them communicate using shared memory and synchronize using locks and barriers. Alternatively, light-weight threads built on top of Hurricane processes can be used, since they have lower overhead for many of the common operations (thread creation, blocking synchronization, barriers). Other models of parallelism are also being investigated, including self-scheduled loop level parallelism, single program multiple data parallelism (SPMD), and coarse grained data flow. Message passing, both within and across address spaces is also being investigated. In all cases, C and C++ are the languages of choice.

Allocation of data is another important consideration in the implementation of a parallel program in Hurricane. Avoiding false sharing, choosing the appropriate policies, and controlling the layout of data in memory are all still in their primitive stages. Calls to allocate dynamic data from distinct Hurricane regions are provided, but it is generally impractical to allocate every object from its own Hurricane region. Although one can also place objects with common attributes in the same region, it remains difficult to separate the different objects on different pages where necessary, and to avoid interference from the memory allocation routines themselves.

3.4 Summary

The following aspects of Hector/Hurricane motivated much of the work in this thesis:

- The cost of remote accesses, despite the mild NUMAness.
- The high cost of memory accesses (independent of network delays).
- The hierarchical architecture promoting locality at multiple levels.
- The limited control available over data placement and coherence.
- The impact of false sharing on the management of replicated data.

Since many of these features are common to other systems, we expect that our work will have application beyond Hector/Hurricane.

Chapter 4

Algorithms and Implementations

A popular method of obtaining locality in a shared memory multiprocessor (apart from caching, which has its limitations as discussed previously) is through page migration and replication. It is relatively straightforward to implement, and can be done transparently to the application. Given the limitations of memory management hardware, there are only a limited number of options open to the implementor: choosing when to migrate or replicate, and when to map a page remotely (on some machines with more complex topologies, an additional consideration is the placement of pages, especially pages that are frozen). These choices generally involve some policy based on the frequency of coherence faults (page faults used to enforce consistency, such as write faults used to catch the first writer to a page with multiple readers) and remote accesses. The goal is to match the actions of the memory manager to the perceived needs of the application by dynamically adjusting to the application's page-reference activity. When it works, such an approach is ideal, as it involves minimal work on the part of the programmer. Unfortunately, it does not always work.

The hierarchical structure, the availability of caches, and the mild NUMAness of Hector suggest additional strategies such as clustered replication, mapping migration, and less aggressive policies. However, such variations are only minor incremental improvements on the same basic theme of dynamic page management, and all such schemes appear to come up short in two major areas:

Obliviousness. By attempting to be completely transparent to the application, they are forced to rely on indirect cues to determine what the application is actually doing. This creates a constant tension between what the operating system thinks the application is doing, and what the application is actually doing.

Granularity. Memory management systems are generally restricted to operate on entire pages. For some problems, where the granularity of sharing is much larger than a page, this mismatch is not a major concern. But for many other applications, this can be a severe drawback, due to the problem of false sharing.

4.1 Shared Regions

Assuming a program does not contain data races, there must be some form of synchronization that guarantees orderly access to shared data; that is, ensuring that there is only either a single writer or multiple readers for any data item at any given time. If synchronization was associated with the data it should protect and the type of accesses to this data (reading or writing) that is to follow, then it becomes possible for the system to determine the movement of data according to the synchronization patterns of the program. For instance, if one process, A, were to acquire a write lock associated with a data item, and at some later time another process, B, were to acquire a read lock associated with the same data item, then the system could determine that the data was moving from A to B. This information could then be used to maintain coherence according to the precise sharing patterns of the application. This is the idea behind Shared Regions.

A shared-region is a portion of some shared data object that is accessed as a unit by a task. Examples might be a row or column of a matrix in a numerical algorithm, an object descriptor in a simulator, or a tuple in a database. Shared-regions are used in such a way as to specifically eliminate the problem of false sharing.

When a region needs to be accessed for reading or writing, this fact is indicated by a special call the programmer inserts into her program: `ReadAccess` begins a read-only phase, and `WriteAccess` begins a read/write phase. When the phase is completed, a corresponding `ReadDone` or `WriteDone` call is made. These calls for read or write access can be used either for synchronization (to enforce a reader/writer lock on the data) or for coherence (to ensure that the latest version of the data is returned on the next read), or for both. The calls bracket sets of accesses, giving the underlying system the opportunity to perform whatever operations are necessary before and after the accesses to keep all views of the data consistent. This often entails moving data between the caches and main memory modules.

Using shared-region synchronization primitives to support coherency is similar to the various relaxed consistency models that have been proposed for hardware cache coherence, where synchronization operations are made visible to the hardware so that the hardware can ensure that data is kept consistent.

A simple example may help clarify the use of shared-regions. Consider a simple shared counter, c, being updated by two processes A and B. In order to ensure that the counter is incremented atomically, the processes may have an agreed upon procedure to first acquire some lock l, read the old value, increment it, write the new value back, and then release the lock l. For just reading the value of the counter, a similar procedure would apply. To treat the data as a shared-region, the lock would simply be replaced by a shared-region descriptor r that is explicitly bound to c. To increment c, the processes would simply acquire write access to c through r using the `WriteAccess` call, and release it using `WriteDone`. To just read the variable, `ReadAccess` and `ReadDone` would surround the read statement. Figure 4.1 shows the two versions, written in C++ with classes for locks and shared regions.

Although the two examples appear similar, there are some notable differences. The first difference is the need to perform the bind operation to associate the region r with the shared object c. This informs the region of the location and size of the data that it is to manage. The next difference is an explicit `WriteAccess` call when the shared data is to be written. This allows the shared-region software to provide exclusive access to the writer, as well as providing key information (the identity of the last writer) required for coherence. Finally, note that in the second access, `ReadAccess` is used since the data is only to be read. This gives the program shared read-only access to the data, allowing other readers to concurrently read the same data.

Of course, shared-regions would not generally be used for such small objects, which probably would not benefit from caching or locality. However, the same procedure as described above would be applied for larger objects, such as a shared buffer or a shared row in a matrix.

Using Locks	Using Shared Regions
`int c ;`	`int c ;`
`lock l ;`	`SharedRegion r ;`
	`r.bind(&c, sizeof(c)) ;`
`. . .`	`. . .`
`/* increment c */`	`/* increment c */`
`l.lock ;`	`r.WriteAccess() ;`
`c = c + 1 ;`	`c = c + 1 ;`
`l.unlock ;`	`r.WriteDone() ;`
`. . .`	`. . .`
`/* read c */`	`/* read c */`
`int k ;`	`int k ;`
`l.lock ;`	`r.ReadAccess() ;`
`k = c ;`	`k = c ;`
`l.unlock ;`	`r.ReadDone() ;`

Figure 4.1: A C++ example, comparing lock version and Shared Regions version.

4.2 Protocols

Although more details are presented in the following sections, a high-level overview of the basic concepts is presented here first.

There are two parts to shared-regions: the replication protocol and the coherence protocol. Naturally, the two depend quite heavily on each other. The replication protocol determines under what conditions a replica of a region is created, while the coherence protocol determines how the different replicas are kept consistent. To maintain coherence simply requires that the data travel from the last writer, call it A, to the next reader or writer, call it B, at some time between the `WriteDone` of the initial writer, and the next `ReadAccess` or `WriteAccess`. In between, there may be multiple (possible inconsistent) copies of the data in the caches and in main memory. Somehow, the data has to move from the cache of the writer, to the cache of the next reader or writer. Although hardware cache-coherence, if present, can transfer data directly between caches, this will not help us if the two processors have different views of the data due to replication in main memory. Since the shared-region software controls replication, it is aware of all replicas, and can ensure that all copies at all levels of memory (cache and main memory) are kept consistent. However, since there is no way (in present systems), for software to initiate a direct cache-to-cache transfer, main memory must always be used to bridge the caches (see Figure 4.2).

This still leaves open the question of how to perform the transfer, since it requires both communication between caches and memory, and between memory and memory. Figures 4.3(a)-(c) show the three components to a complete coherence system: cache-to-memory coherence, memory-to-memory coherence, and memory-to-cache coherence. Cache-to-memory coherence ensures that any changes made in the cache are reflected in main memory when necessary (before another processor reads the main memory copy) by copying the dirty data from the cache to main memory. Memory-to-memory coherence ensures that a copy of the data is updated with the latest version before another processor reads the old copy. This is accomplished by simply copy the data from the new region to the old

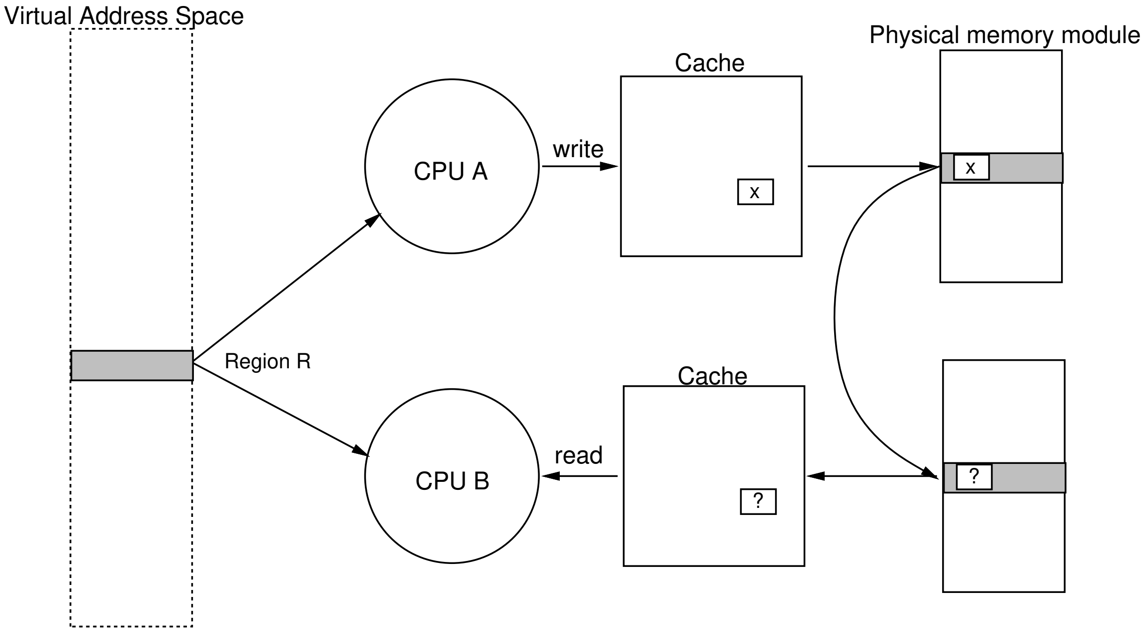


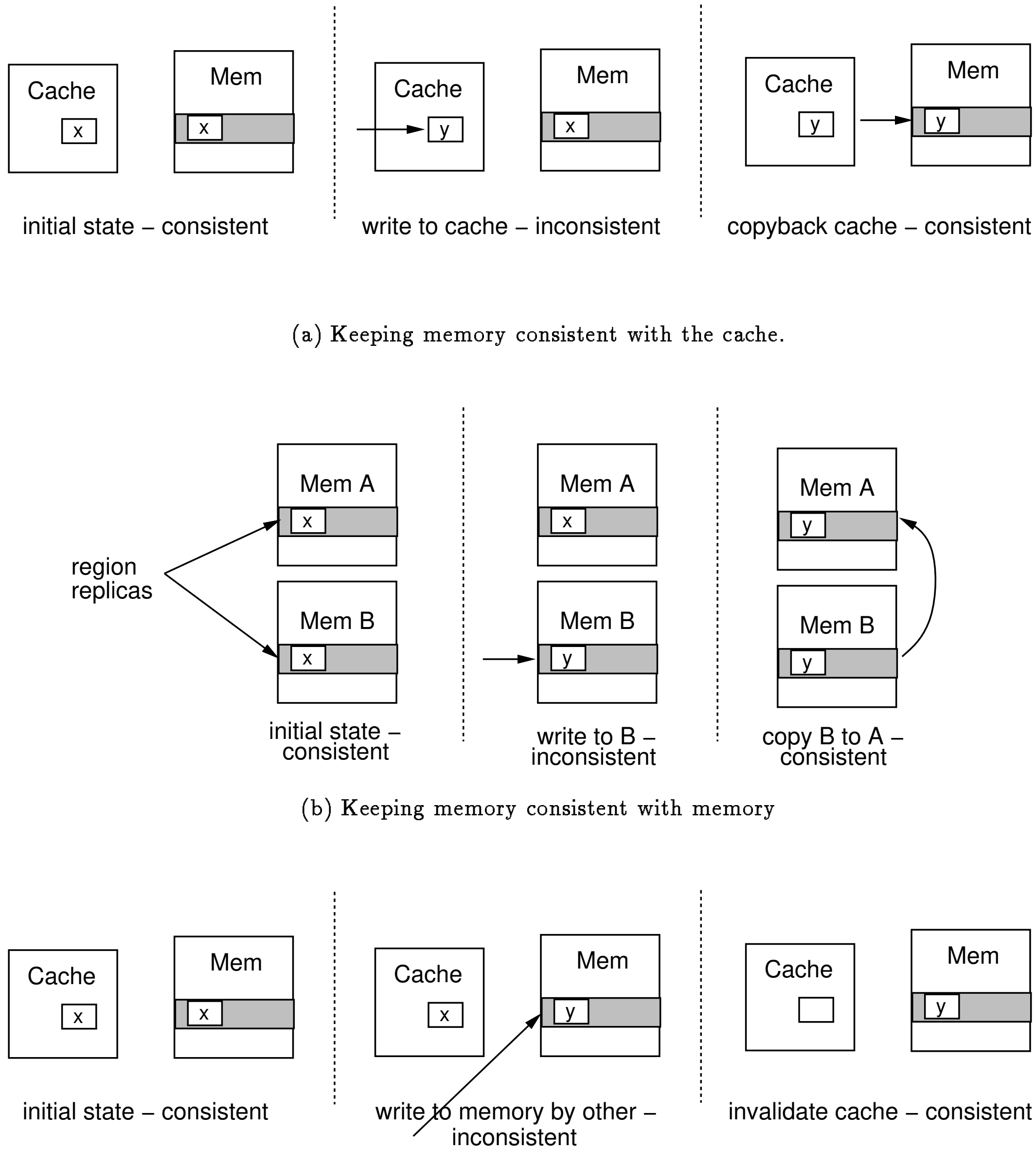
Figure 4.2: This figure shows the state of the system after a write has been performed by cpu A and written back to memory. Nothing is known of the contents of the cache or main memory copy for B, so some method must be used to move results from the memory A, to the B's memory, up through the B's cache, and final to cpu B.

one. Memory-to-cache coherence ensures that what is read from the cache reflects the latest version of the data in memory (assuming the memory has the latest copy). This is done by invalidating any stale data from the cache before accessing it. Together, these three operations can keep two or more caches and memory modules mutually consistent. Although certain operations, such as performing a cache copyback, must be performed by the local processor, the initial request can originate from another processor and be forwarded on using an inter-processor interrupt. Hence, all the coherence operations can be initiated by any of the processors.

We also have not specified when the transfer takes place. In fact, different parts of the transfer can take place at different times. For instance, at `WriteDone` the data could be copied from cache to main memory, and from main memory to other copies, and then left to the next reader/writer to flush its processor's cache of any stale data before reading it. Alternatively, the data could simply be copied from cache back to memory at `WriteDone` time, with the rest left up to the next reader. There are many different ways to divide the responsibilities, although, as we shall see, some are more efficient than others.

Because of the relationship between the cache and main memory, the consistency policies chosen must consider the problem of both cache and main memory coherence together.

A further complication arises because of our dependence on pages for replication. Although we often refer to replicating a region, in actual fact we can only replicate the pages the region is contained in. Consequently, the parts of pages not in the shared-region may also be "accidentally" replicated. This limits some of the flexibility of the coherence and replication protocols, and makes it impossible to fully separate the protocols from the implementation details surrounding the use of pages.



(a) Keeping memory consistent with the cache.

(b) Keeping memory consistent with memory

(c) Keeping the cache consistent with memory.

Figure 4.3: The three components of consistency: cache→memory, memory→memory, memory→cache.

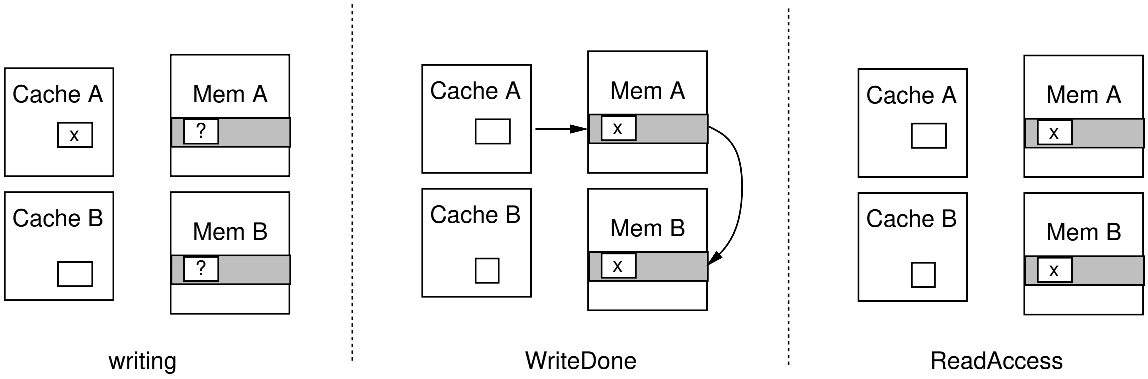


Figure 4.4: Update algorithm. Processor A writes to its cache. B's cache is empty (for this region) as it always must be between access with the Update protocol. Processor A then performs a `WriteDone`, thus writing its cache back to memory, and updating all main memory copies, including B's. It also flushes its cache. Finally, B performs a `ReadAccess`, which in the Update algorithm, requires no action.

4.2.1 Main Memory Coherence Protocols

This section deals with protocol options for managing coherence at the main memory level using shared-regions. Two main protocols are discussed, although variations on these protocols are also possible.

Update Protocol

One of the simplest protocols, and the one that was first used for software cache management, is called the *Update* protocol. Under this protocol, all copies of data are kept consistent at `WriteDone` time. This means that all replicas of the region being written must be updated as part of every `WriteDone`. The only restriction on the cache protocol is that it must ensure that no dirty data remains in the cache after a `WriteDone`. However, to simplify the discussion, we match the Update memory coherence protocol with the Update cache coherence protocol. This means that not only are the cache contents of the region written back to memory at `WriteDone`, but the region is invalidated from the local cache as well, for both `ReadDone` and `WriteDone`, to ensure that there is no stale data left in the cache at the next `Access`. This ensures that copies at all levels, memory and cache, are always consistent at `Done` time (see Figure 4.4).

The Update protocol is a simple protocol to implement, requires no coordination between the different processors (apart from any synchronization support explicitly requested by the user), requires no state information, and has the additional feature that all data is consistent between `Accesses`. This last point may prove important, as it provides a simple way to determine a safe point for reorganizing regions.

To map the protocol onto a page-based system requires little change. Any replication protocol may be used as long as the operating system keeps track of page replicas and can map the addresses of the region to the copies of the pages containing the region (facilities which are generally required of all replicating page-based operating systems). All that is then required is some facility for updating the page replicas with the data from the region. The details for accomplishing this under Hurricane will be discussed later in Section 4.3.

Although simple to implement, the Update protocol has a number of limitations. The

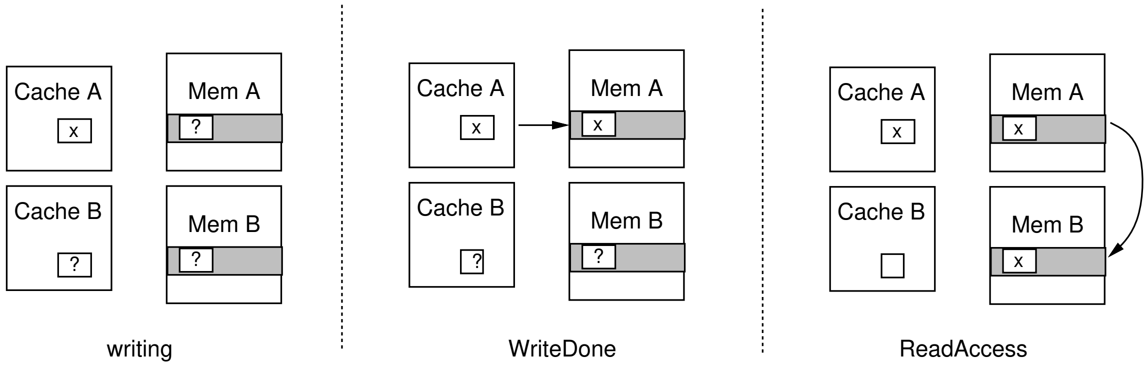


Figure 4.5: Validate algorithm. Processor A is writing to its cache. Nothing is known about the same of the main memory copies or the other caches. At `WriteDone` Processor A writes its cache back to main memory, but does not invalidate it. Processor B then does a `ReadAccess` and determines that its cache and memory copies are invalid. It invalidates its cache and obtains a new copy of the data from A.

primary limitation is that consistency is performed at every `Done` operation. This means that data copies must be located and updated on each `WriteDone`, even if none of the replicas will be accessed before the region is written again. A second limitation is that because pages are used for replication, many of the replicas may only exist because some other region on the same page caused a replication (i.e. false sharing); the region replica being updated may in fact never be accessed at all. In both cases, a large number of unnecessary and costly updates may be performed with this algorithm.

Validate Protocol

The *Validate* protocol differs from the Update protocol primarily by enforcing consistency at `Access` time rather than at `Done` time. When an `Access` occurs, the shared-region software checks whether the copy that is about to be accessed is valid or not. The validity of the region copy is determined by checking whether another processor has written to the region since it was last `Accessed` by this processor. If the region copy is found to be invalid, a fresh copy is retrieved from the closest processor that has a valid copy. As with the Update protocol, although this memory protocol can work with different cache protocols, it is simplest to pair it up with the corresponding cache Validate protocol. The cache Validate protocol copies data from the cache to main memory at `WriteDone` time, like the cache Update protocol, but does not invalidate the cache. It instead validates its cache copy on each `Access` in the same manner we have just described for the memory Validate protocol, and invalidates the cache if it finds the copy to be invalid (See Figure 4.5).

In many cases, it may be possible to determine which processors are likely to access the region following a `WriteDone` (either from past region access patterns, or from extra hints provided statically or dynamically by the user). Under such circumstances sending updates to those copies at `WriteDone` time can have several advantages:

- Copying from the source may be cheaper than copying from the destination (see Section 5.2.2).

- Making multiple copies with one call may be cheaper than having each processor making its own copy (see Section 5.2.2).

- The time spent by the producer may be less valuable than the time spent by the consumers, so that it may be advantageous for the producer to do the work of updating copies.

On the other hand, having each processor grab the newer data has the advantage of only requiring data movement when it is needed (on demand) and only to the processors that are actually going to access it. It also naturally takes advantage of the parallelism in the system by allowing each processor to manage the region copying itself.

In order for the implementation of the Validate protocol to keep track of the different copies, a status vector is kept with each region descriptor indicating the status of each copy and the processors referencing it. The status is maintained by marking the status of the other copies invalid on `WriteDone`, and marking the status of the local copy valid on any `Access`.

With this protocol, the operating system must provide the tools necessary to manipulate the cache, to copy region data between memory modules, and to instantiate new page replicas. It is not responsible, however, for keeping track of page copies.

One limitation of the Validate protocol is the extra overhead in both time and space required to manage the status vector. In particular, in the absence of special concurrent fetch-and-ϕ operations [GGK$^+$83], it requires a regular lock to protect updates to the vector, which can be expensive.

A potentially more serious limitation is the lack of flexibility in replication protocol selection. Because the user-level software is responsible for keeping track of which copies are valid, and which are not, it must be aware of all copies in existence and which processors are referencing them. This is in contrast to the Update protocol, which lets the operating system keep track of these details. Not only is it complicated to keep track of which processor is referencing which copy (note that it is not necessarily required that every processor have its own copy), but due to false sharing, it may actually not be feasible at all (this will become clear in the next section).

Because of these complications, certain restrictions on the choice of replication protocols are required. To begin with, all regions that share a page must use the same replication protocol. Under Hurricane, this generally means that all regions allocated from the same Hurricane region must use the same replication protocol. Second, because replication is partly under the control of the operating system, the policies chosen for the Hurricane region must be compatible with the replication protocol selected for the shared-regions it contains (in general this means that both must have the same notion of when to replicate and when not to replicate). Finally, it must be possible and efficient to determine the particular copy of a region that will be referenced by any processor, simply from the processor id.

These restrictions should become clearer in the implementation section.

4.2.2 Replication Protocols

The replication protocol determines when copies are made and where they are placed.

Replicas may be created either as a result of a region `Access` or `Done` operation or as the result of a page fault.

Although ideally we would like to discuss replication of regions, the only means of transparent replication available is through the use of pages. Hence the two protocols presented here are really page replication protocols, and not region replication protocols (although they are driven by operations to regions that reside on the pages).

Degree of Replication

It is possible to control the degree of replication by using the distance to the nearest replica to determine whether to replicate. If the nearest replica is within some predetermined set of memory modules, no replication is made and the existing copy is simply mapped to directly. Otherwise, a local copy is made. To ensure that no processor ends up with two copies in its local set, the space of memory modules must be partitioned into sets that are disjoint, yet together cover all modules. This of course, matches perfectly the notion of locality as present in Hector, where sets generally consist of either individual processor's memory modules (full replication using a set of one), all memory modules in a station, or all memory modules in a local ring.

Lower degrees of replication (for example, one per station in Hector) may be beneficial for applications for which the extra locality of full replication may not outweigh the cost of the additional replications.

Replicate on Write/Reference

The decision to replicate can also be based on the type of access. Although it is common in page-based dynamic replicating systems to replicate on reads, and although in our case we can replicate on reads and writes, it may also be beneficial to consider replicating on writes only. Replicate-on-write is useful for applications that have a high intensity of accesses from writers, and a lower intensity from readers, as occurs in many domain-decomposition type problems. In these circumstances, keeping the data local to the writers may make sense, but the coherence work required to keep replicas for the readers up-to-date may outweigh any benefits gained from the additional locality.

Unfortunately, the replicate-on-write option can only be used with the Update protocol, and not with the Validate protocol. The problem is caused by the following scenario: consider two regions, x and y, that occupy the same page, and that are using full replication with replicate-on-write. Assume both have been read by two processors A and B, and that the page they are contained in is currently located on B. Processor A then writes to region x. This replicates the page onto A. B then writes to y. It does not know that A now has a copy of y as the result of the page copy that happened due to x, so it does not know to mark A's copy invalid. The next time A accesses the region, it accesses the local, out-of-date copy. Since the Update protocol always updates all copies, it does not have this problem.

4.2.3 Cache Coherence Protocols

Although the focus of this thesis is on the two memory protocols outlined previously and on the two cache protocols that go with them, it is possible to mix and match the memory protocols with the following cache protocols.

- Invalidate the cache on `Done`: This is used by the cache Update algorithm. It can be used by either memory coherence protocol. Caches are always clean, and no status information needs to be maintained to control it. This may result in extra invalidations of data that could otherwise be reused.

- Invalidate the cache on `Access`: This is used by the cache Validate algorithm. It can be used by either memory coherence protocol. This requires some form of status information to intelligently determine whether the data is stale or not.

- Copyback the cache on `WriteDone`: This is currently used by both cache algorithms. It requires no status information to be maintained, but may result in unneeded copy-backs.

- Copyback the cache on `Access`: This requires the next processor that accesses the region to request the last writer to copyback any dirty data in its cache. This can only be used by the Validate protocol (but is not currently used). It requires status information to determine the last writer, and pays the extra expense for a remote operation.

4.3 Implementations

This section describes the implementation details of the coherence protocols described in the previous section. It attempts to give a sense of the level of complexity of the implementation, and the overheads involved. Where options were available they are briefly discussed and a rationale for those chosen given.

The implementation of shared-regions is split between the operating system and a user-level library. The operating system implements a set of general calls that can be used to implement a number of different coherence strategies. These calls generally deal with the copying of data and the allocation of page replicas. The user-level library uses these calls to implement specific coherence policies. This makes it relatively simple to experiment with different policies at the user-level, without requiring any changes to the operating system.

4.3.1 Operating System Support

As was briefly outlined in Section 3.2, a Hurricane address space is composed of a set of (Hurricane) regions, each of which consists of a contiguous set of pages mapped to some file object. Each Hurricane region has a set of attributes that control how it is managed. Attributes that can be set on a region-by-region basis include:

- Whether the region is cachable.

- Whether the region is read-only.

- Whether each page in the region should be allocated on the processor that first touches it, or whether it should be allocated in round-robin order around the system.

- Whether the page should be migrated when accessed remotely.

- Whether the page should be replicated.

Additional options that were required by the coherence and replication protocols described earlier (and for experimentation purposes) include:

- Whether coherence of the caches and main memory replicas should be maintained.

- Whether pages should be replicated to all processors that access them (full replication), or only one per station (station replication, where a station is the bus-based component of Hector; see Section 3.1).

- Whether replication should occur on any remote access, or only for remote writes.

- The ability to specify a processor from which all pages should be allocated.

In order to support these new options, many changes were necessary. Because these changes did not all fit in cleanly with the original implementation, the cost of some of these operations is higher than would be necessary in a more carefully tuned implementation. For the most part, this should have little impact on the results, but where necessary, actual and projected costs will be taken into consideration when analyzing results.

In order to allow the user-level software to manage the cache, three new system calls were required:

- `copyback`, to request the cache to copyback a specified portion of the virtual address space from the cache to main memory.
- `invalidate`, to request the cache to invalidate a specified portion of the virtual address space in its cache.
- `flushcache`, to request the cache to copyback and then invalidate a specified portion of the virtual address space.

To move data among the memory modules, another system call was added, `CopyRange`. This call is fairly complicated, primarily because of the large number of options available to the user. The options include:

- The source address and length of the data to be copied.
- Which processor should be considered the source of the data. The source for the data specifies not the particular memory module that contains the pages to be copied, but the processor which is currently mapping the source. This is because under some policies, such as replicate-on-write or station-replication, the processor accessing the data may not be the one that contains the memory module with the data. Since the user-level software is primarily aware of the processors accessing the data rather than the data's explicit location, specifying memory copies in this way turns out to be easier.
- The set of processors that are to be updated; if zero, all existing copies are to be updated.
- A set of (mostly) orthogonal flags to indicate:
  - Whether a new replica should be allocated if one does not already exist (if new replicas should not be allocated and a replica does not already exist, the target is skipped during a copy operation).
  - Whether the source data should be read through the cache or read uncached.
  - Whether the destination data should be written through the cache in write-through or writeback mode, or whether the cache should be bypassed.
  - Whether the destination data should be flushed after being written (required only if using writeback mode for writing).
  - Whether the copy operation should be migrated to the source processor (this requires a remote interrupt to the source processor).

```
Process options and determine whether request should be handled locally
If not local Then
   interrupt each target processor passing them a procedure to call
   wait for all of them to complete
Else
   For each virtual page that the region overlaps Do
      find the source page based on the physical address of the region
            as mapped by the source processor
      find all replicas of the page
      For each processor Do
         If it is one of the processors to be updated Then
            If it needs a replica but does not have one Then
               allocate one
            Endif
            copy the data from the source page to this replica
         Endif
      Endfor
   Endfor
Endif
```

Figure 4.6: `CopyRange` algorithm.

- Whether the copy operation should be migrated to the destination processors (this operation will be done in parallel if there is more than one destination processor).
- Whether the Hurricane region containing the data is using a full-replication or station-replication strategy (in theory this could also be derived from the available data structures).
- Whether the tree-copy algorithm should be used (this option will be discussed later in Section 4.3.3).

The basic operation of any `CopyRange` call is presented in Figure 4.6.

4.3.2 Shared Region Implementation

All of the user-level support for shared-regions can be found in a single library. The library and all shared-regions programs are written in C++, using a thread package similar to PRESTO [BLL88]. The library defines a shared-region class, and the methods for `ReadAccess`, `ReadDone`, `WriteAccess`, and `WriteDone`, as well as a few debugging calls.

The individual fields of the shared-region descriptor are presented in Table 4.1.

Note that for efficiency the bit vectors represent the status of both the cache copies and main memory copies. So, for example, if bit p for the `oldstatus` vector is set, it means that processor p might have some portion of the region in its cache and local memory and that

`start`	The starting address of the data.
`size`	The size (in bytes) of the region.
`atts`	A set of flag bits that store attributes for this region, such as the coherence protocol and the locking protocol that should be used.
`lock`	A lock to control access to the region descriptor.
`havestatus`	A bit vector with a bit set in position p if processor p has ever accessed the region (in other words, if it might have a copy).
`oldstatus`	A bitvector with a bit set in position p if processor p has an invalid copy of the data (that is, a write has occurred since the last access).
`rwlock`	A reader/writer lock used to allow multiple readers (using the `ReadAccess` call) or a single writer (using the `WriteAccess` call) access to the region data.

Table 4.1: Fields of a shared-region descriptor.

the region has since been modified by another processor. Before clearing the flag both the cache and main memory copies must be brought up-to-date (in the case of the cache this means invalidating the old copy, while in the case of main memory it means getting a new copy). This constrains the implementation to the two main protocols discussed, although extensions using more status vectors could support other cache and memory coherence protocol combinations.

In order to guarantee that the data that the descriptor refers to can be managed correctly, the programmer must allocate the object from a Hurricane region with the correct attributes: *consistency turned off*, and all other attributes (as described earlier) matching the protocol selected for the region. For instance, if the Validate protocol with station replication were to be used, then the data would need to be allocated from a Hurricane region with coherence turned off, and station replication turned on. Because the shared-region descriptor is read/write shared among multiple processors, the descriptor itself must be kept consistent. Although it would be possible to manage consistency of these descriptors explicitly, it is easier to simply leave the data in an uncached Hurricane region, and keep memory accesses to it to bare minimum needed in the implementation.

To give a better idea of how the region descriptor is used to implement various coherence policies, the algorithms for the the five main region routines are presented in Figures 4.7 through 4.11. The constructor function, outlined in Figure 4.7, stores options and allocates space for any optional descriptor fields. The `ReadAccess` function, described in Figure 4.8, performs a number of operations. First, it determines whether the region descriptor is expected to perform data synchronization, or the programmer's algorithm already provides this. If the region is responsible for synchronization, it acquires read-access (non-exclusive) to the data using the `rwlock`. Next it checks which protocol has been requested. For the Validate protocol it must determine whether it has a copy of the data, and if so, whether it is valid or not. If it is not valid, it must flush its cache, and optionally acquire an up-to-date local copy based on the replication strategy selected. The status vector must then be atomically updated (which requires temporarily locking the region descriptor). For the Update protocol, no work needs to be done at `ReadAccess` time.

Figure 4.9 describes the operations for the `WriteAccess` call. This is almost identical

```
store options in descriptor.
allocate additional fields based on selected attributes.
```

Figure 4.7: Shared-region constructor algorithm.

```
If synchronization is enabled Then
  acquire reader/writer lock for reading /* shared access */
Endif
If Validate protocol Then
  If this processor has an invalid copy Then
    invalidate region from the cache /* get rid of stale data */
  Endif
  If (no copy or copy invalid) and replicate Then
    find valid copy from status vectors
    If valid copy found Then
      CopyRange create/update local copy from valid copy
    Endif
    mark copy valid
  Endif
Else /* update protocol */
  /* empty */
Endif
```

Figure 4.8: `ReadAccess` algorithm.

to the `ReadAccess` call except that write-access (exclusive) is acquired if synchronization is to be provided, and all existing copies of the region are marked invalid at the end of the routine. Again, no coherence work is required for the Update protocol.

For `ReadDone` in Figure 4.10, the important thing to note is that no coherence work is required for the Validate protocol, while the cache is always flushed for the Update protocol. Similarly, for the `WriteDone` call, in Figure 4.11, the Validate protocol only needs to copyback any dirty data from the cache to the local copy, while the Update protocol must also update all existing copies. Also note in the `WriteDone` call, that any shared-data synchronization is performed at the end, after all coherence work has been performed.

One glaring omission from the above is the destructor for the region descriptor. It is missing because there is no implementation for remapping region descriptors to different regions or for mapping new region descriptors to data that is already bound to a region. Such operations are desirable in cases where the natural region partitioning changes between phases of the program. The call is missing because, in general, there is no way of guaranteeing the consistency of the data once the region descriptor is destroyed. For the Update protocol, there are obvious safe points for doing this for which it will work (the safe points being at any time between a `Done`and the next `Access`). For the Validate protocol, however, caches would need to be flushed and copies updated before a region could safely be remapped, and the coordination required for this has not yet been implemented.

```
If synchronization is enabled Then
  acquire reader/writer lock for writing /* exclusive access */
Endif
If Validate protocol Then
  If this processor has an invalid copy Then
    invalidate region from the cache /* get rid of stale data */
  Endif
  If (no copy or copy invalid) and replicate Then
    find valid copy from status vectors
    If valid copy found Then
      CopyRange create/update local copy from valid copy
    Endif
    mark copy valid
    mark other copies invalid /* could also be done in WriteDone */
  Endif
Else /* update protocol */
  /* empty */
Endif
```

Figure 4.9: `WriteAccess` algorithm.

```
If synchronization is enabled Then
  release reader/writer lock
Endif
If Validate protocol Then
  /* empty */
Else /* update protocol */
  flush region from the cache
Endif
```

Figure 4.10: `ReadDone` algorithm.

```
If Validate protocol Then
  copyback region from cache to memory
Else /* update protocol */
  flush region from the cache
  If replicating then
    CopyRange to update all existing copies
Endif
If synchronization is enabled Then
  release reader/writer lock
Endif
```

Figure 4.11: `WriteDone` algorithm.

4.3.3 Scaling Issues

Since the primary goal of this work is to explore memory management techniques for obtaining locality and thus reducing access time in large-scale multiprocessors, the scalability of the implementation proposed is of paramount importance. A preliminary examination through experimental work is presented in Chapter 5, but an examination of the expected impact due to implementation decisions would seem appropriate first.

Page Allocation and Replication

Since replication requires allocating page frames by the memory manager, it is important that this operation scale well. Additionally, replication requires locking the source and destination pages, and the bandwidth to support the copy. Hence it is important that the locking scheme and data structures implemented in the operating system support concurrent page allocation and replication (from the same source page in particular), and that the memory and interconnection network have sufficient bandwidth to support multiple copies. As we will see in the experiments, neither of these is the case in the current system.

Region Descriptors

Since region descriptors are allocated as normal memory objects, contention for the data structures and the locks that protect the descriptors can become a bottleneck. Proper solutions require distributing the region descriptors across the system to reduce hot-spots, and either replacing the spin locks with more scalable synchronization primitives, or preferably implementing the shared-data structure operations with no locks at all; fetch-and-OR and fetch-and-AND atomic operations would allow us to atomically set and clear bits in the status bit vectors. Unfortunately, non of these techniques was tried, so we expect there to be noticeable impact on performance due the region descriptors lack of scalability.

Cache Operations

This is one case where the shared-region approach should perform well. Because all cache operations are performed as needed by the processor directly connected to the cache, this part of the protocol is completely scalable. However, some proposed variations do require remote cache operations, but these only incur the additional cost of the remote call which should scale well.

Multi-Copy Data Movement Operations

This is a critical problem in large-scale shared-memory multiprocessors. There are a number of approaches to distributing data efficiently, depending on the size and topology of the system and the amount of data to be transferred, but just a quick look at the options studied in this thesis is presented here:

Sequential – source driven. When there are multiple copies to be updated at once, the simplest method is simply to copy the data first to the first destination, then to the second, and so on. This approach is simple, has low contention, and only requires $O(PR)$ amount of work, where work is measured by the total number of cycles spent by all processors on this task, where P is the number of copies and R is the size of the region. It also has a latency of $O(PR)$.

Parallel – source driven. With this technique, the copy is performed in parallel by interrupting each processor connected to the target memory, and asking it to copy the data. This is slightly more complicated to implement than the sequential version. It also has certain performance problems. Whereas in the sequential case the cache could be used to avoid reloading the local data for each copy, in the parallel version each processor must load and store directly to memory. The parallel version also has a potential bottleneck in the code that initiates the remote copies and waits for them to complete, as this is performed serially. However, the biggest performance problem is due to contention at the source memory module which must support P concurrent readers. Ideally, this method would still require $O(PR)$ amount of work and have a latency of $O(R)$, but, because of contention, latency quickly degenerates to $O(PR)$, only now because all P processors are being used, the total amount of work is $O(P^2R)$.

Parallel – destination driven. For some applications, there may be a choice between updating all copies at `WriteDone` time, or waiting for each processor to `Access` the region and make its own copy. With the latter, programs are often tightly synchronized so that the copies happen at roughly the same time. This generates a destination driven parallel copy. This approach is often easier to support since the parallelism is natural in the task, however the problem with contention in the previous case still applies, and hence it suffers from the same limitations.

Tree – source driven. In order to specifically address the problem of contention, a tree broadcast approach was investigated. Here, instead of the source updating all copies directly, it only updates w copies, for some $w = f(P)$. These $w + 1$ copies are then used as the sources for $w + 1$ parallel copies to w more processors each, and so on (actually, w can be different at every level depending on the number of copies and the architecture of the system). Each of these 1-to-w copies can be done either sequentially or in parallel (the choice depends on P, contention factors, and whether lower response time or less work is more important). This technique is ideal for a hierarchically structured architecture, such as Hector, where a properly organized tree can keep each level of communication completely independent of concurrent operations in other subtrees. Note that this approach is more complex than the others, and theoretically, it both has a higher latency (between $R\log_w P$ and $Rw\log_w P$) and requires more work than the ideal parallel case (depends on contention and choice between parallel or sequential approach), although this is not the case in practice. See Chapter 5 for results.

Tree – destination driven. Finally, in the case where the targets independently and concurrently request the data, it may still be possible to use a tree distribution, but it is a bit more tricky. The essential idea is for each processor that is about to make a local copy check whether some other processor in its local set is already retrieving a copy. If this is the case, then the processor simply waits for that copy to complete before starting its own copy (from the recently created copy). If no local copy is underway, then a check is made for a copy in the enclosing set, and the process is repeated. Dealing with asymmetries in the request distribution is also possible, although it complicates the algorithm significantly. A similar approach for a dynamic combining tree barrier is described by Gupta and Hill in [GH89]. None of the possible techniques were implemented in this thesis however.

4.4 Choosing Shared-Region Sizes and Options

Two of the most important decisions for the programmer when working with shared-regions is choosing the size of the region and the coherence protocol.

Choosing the best partitioning of an object or objects into shared-regions requires balancing the following conflicting goals:

Convenience. Ideally, regions are chosen to correspond to the natural decomposition of the data for parallelism. That is, for a given parallel unit of work, the data that is accessed by that task would be defined as a region.

Concurrency. Ideally, regions are chosen so as to provide the maximum concurrency, as only one process can write to a region at a time. In this case a region might be a single word, or a collection of data items that is always accessed atomically.

Overhead. Ideally, regions are chosen so as to minimize the overhead both in the space for the descriptors and the time to access them. Regions should therefore be as large as possible.

Clearly, the optimal granularity will depend on the application. Usually convenience is the first criterion examined. If this approach results in reduced concurrency or excessive overhead, a compromise is required. In practice, however, it has been found that there usually is an obvious choice for the region that strikes a fair balance.

Choosing among the coherence protocols is more difficult. The Validate protocol generally results in fewer coherence actions (cache and data copying operations), but because of the limitations in replication options using this protocol, it may not always be the best choice. In addition, the Validate protocol incurs greater overhead, relative to the Update protocol, since it must maintain the status vectors, often doubling the cost of region `Access` calls. Considering caching alone, if regions are likely to be re-acquired several times before a conflicting access, and if these regions are likely to remain in the cache, then the cache Validate protocol would provide some benefit by not unnecessarily invalidating the cache unless such a conflicting access occurs. With respect to replication, the choice will depend first on whether the extra locality due to replication is likely to be of benefit, and then on whether the replication options available with the Validate protocol are sufficient. If they are, then once again the Validate protocol will result in fewer data updates. If not, or if there is likely to be little reuse of regions before conflicting accesses, then the Update protocol is likely to be the best choice.

4.5 System Integration Issues

The previous section described the implementation of shared-regions, but did so effectively in isolation of the rest of the system. There are critical issues that have not been addressed, in part because no satisfactory solution has yet been developed.

Probably the most important unanswered question is how to release page replicas back to the free pool once they are no longer needed. Normally, the operating system would know that all replicas are consistent (that is, when it is enforcing consistency itself), and thus could choose to free any replica it desired. In our case, we have specifically removed the responsibility of maintaining coherence from the operating system, and hence must carry some of the burden of responsibility for managing the replicas.

In the current implementation each region is completely independent of the others; it does not know of the existence of any other regions, and hence does not know whether other regions share the same page. Therefore, consistency of an individual region does not guarantee the consistency of the pages it is contained in. One solution would be to have the mapping of regions to pages (essentially the way in which the regions tile the shared data space) available to all regions. This would provide the necessary information to determine the coherence of a particular page as well as a method for making a particular page coherent. An efficient method of managing this information would be required however, and a policy for deciding when to free pages would also be necessary. This information could also be provided to the operating system, which could then use it to determine that it was safe to free a page by the fact that all regions overlapping the page had valid copies elsewhere, or it could update other copies of the region itself so that it was safe to free the page.

Another concern is what to do if a particular memory module runs out of memory, and a local replica cannot be created. Under normal conditions the operating system would simply force the process to reference a remote copy of the page. Under the Update algorithm, this is still an option, but using the Validate protocol, we run into problems similar to those with replicate-on-write, where the region descriptor software does not know if a local copy for a region exists or not. Unfortunately, for the moment, there is no provision for such a situation, and hence failed replications are not handled gracefully.

Finally, there is the issue of process migration. Since coherence is maintained on a per processor basis and not on a per process basis, process migration can be tolerated provided it never happens in the middle of a region `Access`. For the moment, this is accomplished simply by not permitting migration (which is easily accomplished in Hurricane simply by choosing the appropriate scheduler).

4.6 Other Limitations

Apart from the unaddressed system integration issues above, there are a number of other limitations that are not addressed in this thesis.

- Deciding where and when to replicate is limited to the simple replication rules outlined in Section 4.2.2: full replication, station replication, replicate on reference, replicate on write. These options are further reduced to simply the degree of replication when the validate protocol is chosen. These limitations are the result of having incomplete control over replication of regions that share the same page with other regions; there must be a uniform policy applied to all regions that will function correctly when regions act independently. Tight coordination between the regions and the memory manager as outlined in the previous section may improve this situation.

- Because entire pages are used for replicating regions, memory is wasted when only a small region on a page is accessed from a processor. This might occur, for example, with regions that are mapped to the columns of a matrix that is allocated in row-major order. There is no easy solution to this problem barring either changing the hardware or the algorithm.

- A potentially severe limitation is that all regions must be aligned to cache line boundaries. This is because of false sharing in the cache lines and the fact that most caches do not keep track of which words (or bytes) in a line are dirty. There is no such limitation with respect to the memory coherence portion of shared-regions.

- As we will see in later chapters, the centralized nature of the region descriptors may prove to be a bottleneck in large systems. In particular, a general scheme to distribute the region descriptors around the memory modules is required, but has not yet been developed (many techniques exist, but were beyond the scope of this thesis).
- Irregularly shaped regions are difficult to define and expensive to manipulate. Non-contiguous regions with a regular stride would be easy to define, but it is unclear how well the cache coherence strategies would perform with such regions.
- There is no provision for dynamically changing the mapping between regions and objects. This was discussed in detail earlier.
- The overhead for region access, although small, can still be substantial when the regions are small and the `Accesses` are frequent.

Chapter 5

Experiments and Results

In the previous chapter we presented several replication strategies and coherence protocols that allow us to obtain locality for objects of arbitrary granularity. The questions concerning the costs involved and the benefits that can be gained, however, have yet to be addressed. We answer these questions by analyzing the performance of the implementation discussed in the previous chapter on the prototype multiprocessor Hector. We use these results to establish expected behaviour on other hardware and software platforms.

We begin by examining the performance of the bare hardware to help put the results that follow into proper perspective. This is followed by performance results for the components used to implement shared-regions: page management, data copying, and region descriptor operations. Finally we examine the impact of shared-regions on some synthetic programs.

5.1 Hardware Performance

Since the results for all experiments in this chapter depend on the performance of the underlying hardware, a summary of the timings for various operations in an idle and in a contended system is presented here. The results are a synthesis of information gathered from the hardware design, from previous studies [HKS91], and from experiments.

The cost of performing simple memory operations on an unloaded system are presented in table 5.1. Some notes about these numbers:

- Access time for the cache is 1 cycle.
- Cache line reads return 128 bits of data in two 64 bit packets. Writes can only send 32 bits at a time.
- Remote (single word) writes are fast because they do not wait for the memory access to complete (local writes do wait however).
- Remote cache writebacks can be expensive (particularly in-station writebacks) because of interference at the memory module among the multiple write requests that are generated.

From these numbers we can see that while the impact of remote accesses is noticeable, their cost are generally only between 2 and 3 times longer than the local case, and only for the most remote memories. Taking into account the latency-hiding impact of the cache, it might seem that with such a mild NUMA system there would be little benefit in localizing

	32bit load	32bit store	cache load	cache writeback
local	10	10	19	19
station	19	9	29	62
ring	$19 + ks$	$9 + ks$	$29 + ks$	$10 + 4(4 + s)$

Table 5.1: Memory access times (in 60ns cycles). s is the number of stations on the ring (the prototype has only 1 ring), while k is a cost factor between 1 and 2 which depends on the position of the source and destination stations with respect to each other.

data at all. For many applications, this is true. However, there are a significant number of applications with heavy memory access requirements. When memory traffic increases, contention becomes a factor, and total memory latency increases.

The following aspects of Hector are the primary contributing factors to high latencies under contention.

- The memory subsystem is busy (cannot accept another request) for the duration of its access time plus all protocol processing and arbitration overhead. For a remote cache load, this is 18 cycles.
- Local requests (requests from the processor or other devices on the same board as the memory), have first priority for access to the memory.
- Off-station requests have priority on the local bus over on-station requests.
- There are no queues anywhere in the system.
- A memory request that fails (because the memory is busy) is repeated, with delays between attempts of 13, 89, and 5 cycles (depending on the number of failures).

Because of the relatively slow memory, efficient interconnection network, and small system size, memory contention, not network contention, has the largest impact on performance in Hector.

The impact of these various factors can be seen in Figures 5.1(a)-(d) and Figure 5.2, which present results from experiments investigating the performance of the memory system under contention.

In Figure 5.1(a) each processor is reading 16 KB from the same memory module. The data is read using a simple loop of single word (32 bit) reads. From experimentation, the delay between reads in the loop implementation has been estimated to be 2 cycles. As the number of processors increases, the latency for the hot spot case increases linearly from about 2 or 3 processors onward (the point of saturation for the memory). Given that the memory is busy for 18 cycles and there is only about 20 cycles between requests (based on unloaded latencies and the read-loop in the test program), this result is to be expected. The vertical bars in the graph give the range of response times for the different processors. Most of this variation is due to the network and bus priority schemes and can be seen more clearly in Figure 5.2. Since the priority of the local processor has such a strong impact, its time is separated from the others, and presented as the dotted line near the bottom. Notice how there is only a slight increase in the local processor's response time, reflecting the extra time spent waiting when it finds a memory request ahead of it already in service (because of its higher priority, it only needs to wait for requests in service). Because the local processor's access time is so much lower (due to locality and higher priority), it actually

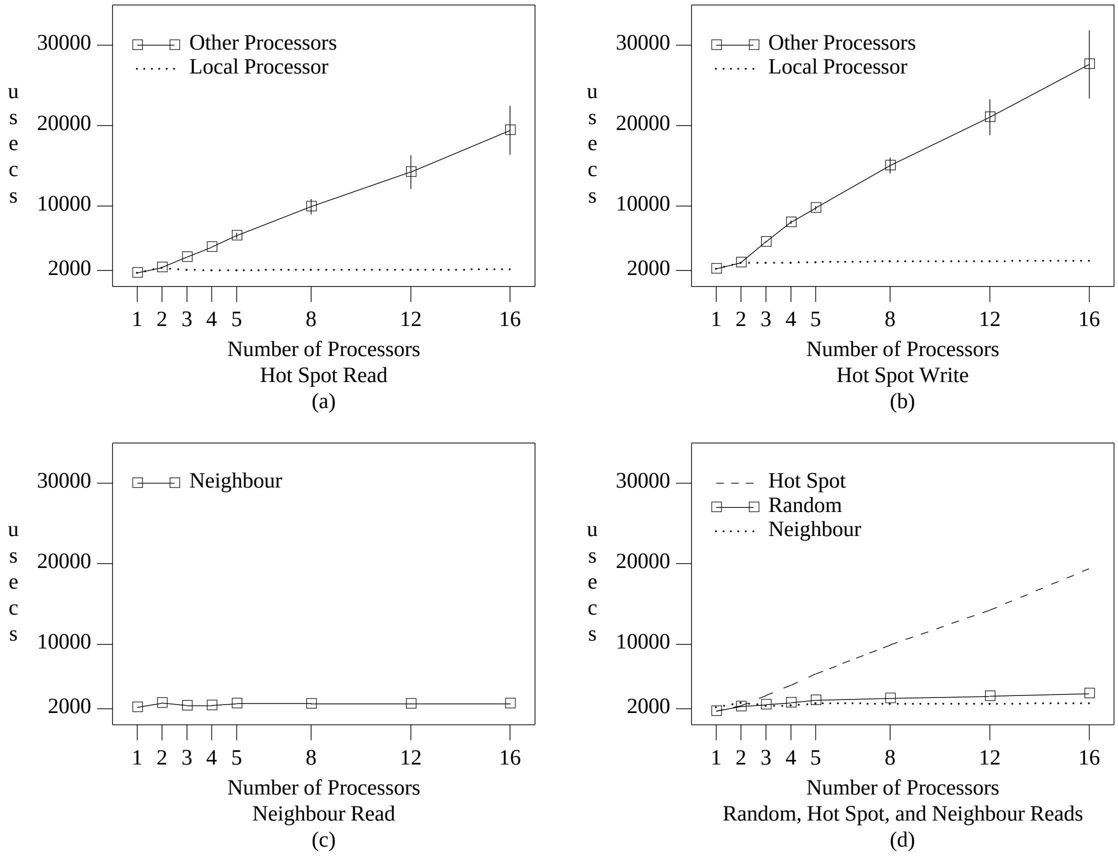


Figure 5.1: Measurements of the effect of contention on memory performance when reading and writing 16 KB of data (times in microseconds).

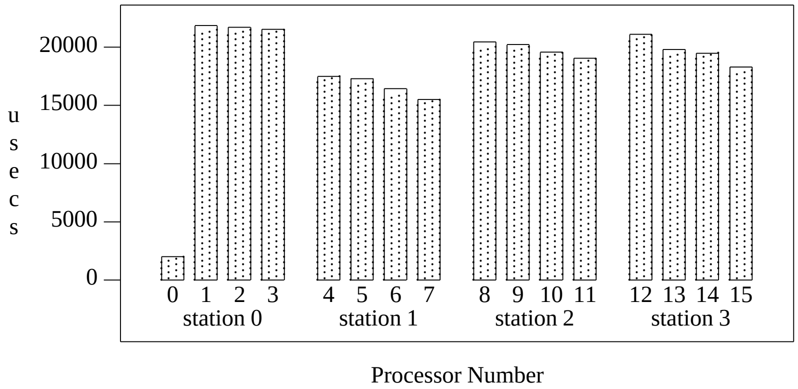


Figure 5.2: Effect of priorities on hot-spot reads with 16 processors reading 16 KB from a single memory module. The data is located on the first processor.

finishes far before the others, and as a result has little effect on the response time of the other processors for this test.

In Figure 5.1(b) we see the results for concurrent writers. The trend is similar to that seen for concurrent readers.

Figure 5.1(c) shows the results when each processor reads from the memory of its neighbour. The results indicate that such access patterns result in virtually no discernible contention in the network.

Finally, in Figure 5.1(d), we see the results when each processor reads from a random neighbour (several hundred iterations were performed). The hot-spot and neighbour read times have been superimposed. This shows that with a random distribution of data (and requests), the system performs fairly well. However, there is a small but noticeable increase in response time for the random case, and later we will see that this minor difference can actually have a strong impact on application performance.

Overall, the results indicate that although network contention has a noticeable impact on hot-spot performance (in the form of variable response times), the primary bottleneck is the memory. It also shows the devastating impact of hot-spots in this system, and how randomizing memory can help improve this situation.

5.2 Supporting-Software Performance

In this section we explore the performance of the software used to implement Shared-Regions: the operating system support software and the shared-region operations themselves.

5.2.1 Page Management Performance

Because the implementation of Shared-Regions depends on some of the facilities of the Memory Manager, the performance of these facilities is important to our understanding of the performance of Shared-Regions. Table 5.2 lists the performance of some of the more common operations of the memory manager:

- soft faults are page faults to map pages already in memory into the program's address space.

- migrations are faults that trigger a page migration, requiring the invalidation of all existing copies, the flushing of corresponding caches and TLBs, and the creation of a new copy for the faulting processor. We assume for this test only a single copy exists at the time.

- replications are faults that trigger a page replication, requiring the creation of a new copy for the faulting processor.

- independent faults for any of the above types, refer to concurrent faults that are to different pages.

- hotspot faults, of whatever kind, refer to concurrent faults that are to the same page.

Unfortunately, in many ways, the version of the operating system used for testing purposes was neither highly tuned, nor configured for optimal performance. Consequently, the

	measured			projected		
	1	8	16	1	8	16
independent soft faults	300	400	500	150	225	300
independent migrations	3500	4800	5800	2500	2650	2800
independent replications	2400	2800	3000	2300	2450	2500
hotspot replications	3100	21000	50000	2300	6000	10000

Table 5.2: Memory manager page fault performance on 1, 8, and 16 processors. Projected times are those predicted for an improved implementation (times in microseconds).

call overhead	22
per page	110–120
remote call	150
allocate page	350 first, 265 subseq.
per copy (parallel)	$20 + 5 \times$ num_copies

Table 5.3: `CopyRange` overhead costs (in microseconds).

measured performance is significantly poorer than is possible given the known alternative implementation techniques.

Table 5.2 therefore shows both the measured times and the projected times based in part on other existing implementations of the operating system, known improvements scheduled for implementation [USK92], and the limitations and capabilities of the hardware. For example, the cost of a single in-memory page fault has already been reduced to under 200 microseconds in other implementations, and could likely be reduced still further. The cost of migrations is artificially high in the current implementation due to changes made to increase flexibility in the system for experimentation purposes. The use of clustering [USK92] has already shown major improvements in parallel fault handling, eliminating much of the contention due to shared data structures.

The important point to observe from these numbers is that for concurrent page management operations on independent pages, the system scales quite well. However, for concurrent operations on the same page, in particular for concurrent replications of the same page, the system's performance is far below the capabilities of the hardware. Therefore, the performance benefits we will see are likely lower than what is achievable.

5.2.2 Data Copying Performance

The most important aspect of the overhead of using shared-regions within the context of this thesis is the cost of moving the data between memory modules to gain locality and maintain coherence. All data movement is performed by the `CopyRange` call. There are two main costs associated with this call: the overhead required to set up the data copying operation, and the cost to actually copy the data.

Table 5.3 presents overheads for certain key operations, required to support the different `CopyRange` options. The "call overhead" shown is the time to copy zero bytes using the `CopyRange` call. The "per page" cost is the time required to find each page that the region overlaps, so that it can be used as the source for the copy operation. The "remote call" overhead is the cost required to request another processor to do some work on the initiating processor's behalf (implemented using inter-processor interrupts). The "allocate page" cost

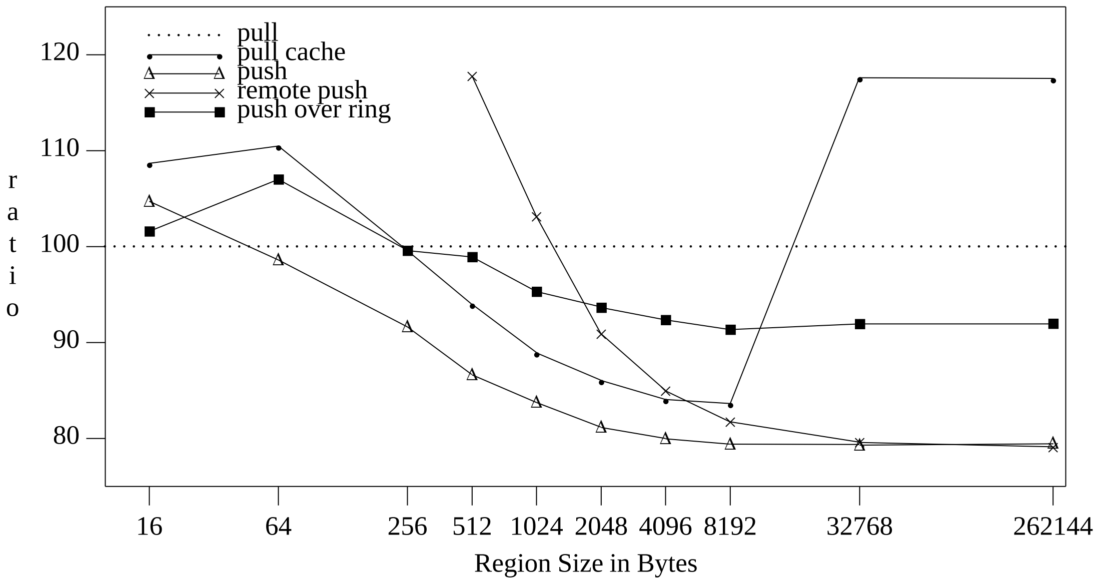


Figure 5.3: `CopyRange` costs for varying sizes and techniques expressed as the ratio to the "pull" time.

is the time required to allocate a new page frame for the destination of the copy, if one is not already present. Finally, the "per copy" overhead is the time required to initiate a number of parallel copy operations and wait for them to complete. Note that this last cost increases per copy as the number of copies increases, due to increased contention and variance.

The cost of actually copying the data depends heavily on the particular options chosen. For the most part the costs follow directly from the memory access latencies presented in section 5.1. Figure 5.3 gives some results for different situations. In this graph, 5 different copying scenarios are shown:

pull. This is when data is copied from an on-station memory module to the local processor's memory.

pull cache. In this case we pull as well, but now we write the data through the cache (normally copies are done uncached).

push. The data is copied from the local memory to a remote (on station) memory module.

remote push. Similar to push, except the request is initiated from the destination processor which requires the request to be forwarded to the source processor for "pushing".

push over ring. Like push, but to a memory in a different station.

The results are normalized against the "pull" time, so that the time for a pull always appears as 100, and numbers greater than 100 indicate poorer performance relative to pull, while numbers less than 100 indicate better performance relative to pull. Also note that the X-axis of the graph is presented in logarithmic scale.

A number of features are clearly evident from this graph. First, all the different techniques are within 20 percentage points of the "pull" technique, indicating that there is not much difference between them. Second, the default approach of pulling the data has the

Size	Pull	Pull Cache	Push	Remote	Ring
16	122	132	127	234	124
1024	589	524	493	608	562
4096	2035	1720	1627	1728	1879
32768	16183	19029	12833	12875	14880

Table 5.4: Sample `CopyRange` data movement costs for single copies (in microseconds).

lowest overhead and hence performs well for small region sizes. However, as the size gets larger, the lower cost of uncached remote writes as used by the pushing strategy, compared to the cost of uncached remote reads as used by the pulling strategy (see Table 5.1), gives the pushers the advantage. Because of the fixed overhead of remote execution, the remote push strategy only performs competitively for regions larger than 2 KB. For the pull cache method, we see that the cache can help hide the cost of writes, but when the capacity is exceeded (at 16 KB), the extra costs involved in using the cache (loading each cache line before the first write to the line) dominates. Finally, the cost of pushing the data over the ring is clearly more costly than pushing the data to an on-station processor, but still less costly than pulling on-station data, showing that sometimes the techniques chosen can have a greater impact than the NUMA factors. Note however, that the choice of pushing versus pulling generally arises from the protocol structure, rather than from strict performance considerations.

Table 5.4 presents a sample of the costs for the different techniques. From these costs and the access times presented earlier, for an average region of 1 KB, the data would have to be accessed 15 times before the cost of replication could be amortized by the increased performance of local accesses. This may seem excessive; however, for many problems the reuse rate is a function of the input size, so that for large problems, there is a large amount of reuse. Additionally, with such large problems, the cache size is often insufficient, resulting in many remote accesses due to capacity misses. More importantly, however, when contention is taken into consideration, the break even point drops considerably.

Up until this point, we have been primarily concerned with the costs of copying data from a source memory to a single destination memory. In many cases however, the data is needed by several processors, and if this is known, optimizations can be used to deliver the data to all processors more efficiently.

There are a number of different methods that can be used, each of which trades-off response time for work (total number of cycles used across all processors) in different ways. Figure 5.4 shows the performance of five methods of copying data (as described in Section 4.3.3). The "serial" method simply copies data to each destination memory module one at a time. It reads through the cache so that it only has to read the data from local memory once. The "parallel" version starts up a remote pulling operation for each copy (see parallel copy overhead above) and waits for them to complete. The "parallel request" method is similar, but here the requests are initiated independently (but simultaneously for this test) at each of the destination processors. The "serial tree" method uses the tree broadcast approach with each parent sending to its children serially (using the same serial method as above). Finally, with the "parallel tree" method each parent asks its children to copy from itself in parallel (using the same parallel method as above).

From these results we can see that for small numbers of copies, the serial version has the lowest overhead and performs the best. As the number of copies is increased, the copy time

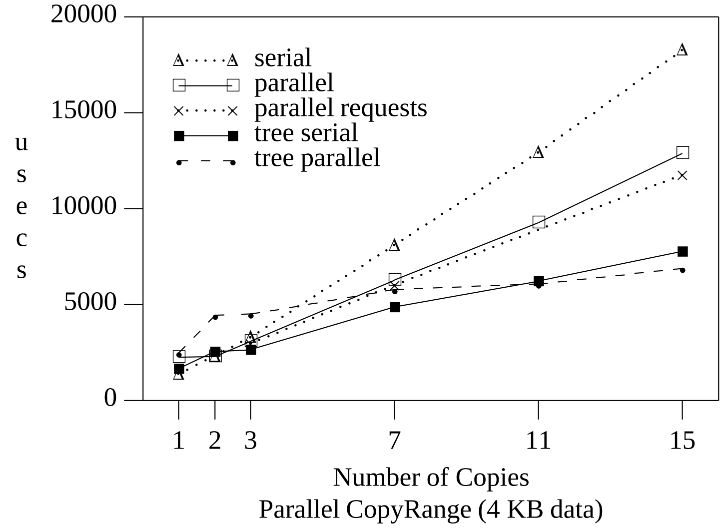


Figure 5.4: `CopyRange` costs for multiple copies.

using the serial method naturally increases linearly, so that it quickly becomes the worst of the five techniques. With the two parallel approaches, performance is improved over the serial version, but they too grow linearly, although at a slower rate, as the source memory module becomes saturated. Although the parallel versions offer faster response time than the serial version, each processor is involved for the entire duration of the copy, and hence the amount of work performed is approximately N times the response time (for N copies). Thus these methods must perform much more work to obtain their improved completion time. Finally, with the two tree versions, we see response time drop even further. The tree serial approach has lower overhead (fewer remote operations) and so performs well for small numbers of copies, however its slope is greater than the tree parallel version, and hence eventually the tree serial version is surpassed by the tree parallel version. Once again, however, the trade-off between response time and work must be considered.

We now compare the time to copy 4 KB of data to all (16) processors using the tree parallel approach and then access the data locally once, against the time for all processors to simply read the data (simultaneously) directly from remote memory. From the numbers used to generate the graph in Figure 5.4, we find that the tree version requires approximately 7000 μs to copy the data, to which we add the 400 μs required to read the data locally, giving 7400 μs. For the 16 processors to read the data simultaneously from the remote memory requires 5500 μs, based on the results used to generate the graph in Figure 5.1 (note that the graph shown is for 16 KB of data). This demonstrates how the gap between the cost of replication and the cost of remote access can close considerably under high contention. For repeated accesses, as presented in Table 5.5, we see that replication becomes more efficient at 2 accesses (assuming the cache is ineffective), and provides greater benefits for greater reuse. Hence, as expected, the trade-off point for replication will heavily depend on the contention at memory, the reuse factor, and the cache hit rate for the data in question.

5.2.3 Region Performance

Finally, we consider the performance of the region operations themselves: `ReadAccess`, `WriteAccess`, `ReadDone`, and `WriteDone`. Each of the various protocols supported will

	Number of Accesses			
Access Method	1	2	3	4
Tree Replicate	7400	7800	8200	8600
Direct Access	5500	11000	16500	22000

Table 5.5: Comparison of costs of repeated concurrent memory accesses with and without replication. Times are estimated based on previous tests.

incur different costs, partly from the coherence operations required by the protocol, but partly out of the supporting structures needed to implement the protocol. Since the data movement operations were covered separately, and since the cache costs follow from the costs of the memory operations, we examine here only the costs attributable directly to the regions themselves.

These costs can be separated into the following two groups:

- Lock acquisition and release costs, if using regions for application synchronization.
- Data structure manipulation (e.g., status vector updates) and associated internal synchronization costs.

The following aspects of the shared-region implementation are responsible for most of the overhead incurred when using shared-regions:

- The basic latencies for accessing the data structures involved will depend on the protocol being used (the validate protocol has the additional cost of maintaining the status vectors, while the update protocol does not), the distance to the memory holding the region descriptor, and the contention at that memory and the network along the path.
- Additional work is required by the validate protocol when coherence activity is required (beyond the data movement operations). Therefore, repeated accesses to a region, without conflicting accesses from another processor, will generally be more efficient than repeated accesses that require coherence activity due to interleaved conflicting accesses.
- The cost of the locks used both to protect the internal data structures and to implement application synchronization can grow quite dramatically (especially in this implementation using simple spin-locks with backoff) when there is contention for the lock.

Because of the large number of factors and the complex way in which they interact, only a summary of the range of costs is presented in Table 5.6. The ranges cover most of the effects discussed above, except for lock contention, which, as noted in the table, can cause times to increase ten-fold for 16 processors.

These results indicate that the costs of the regions themselves cannot be ignored. In particular, with a high degree of sharing, contention for the region descriptors can have a significant effect on performance.

Processors	Time Range
1	30–70
4	50–150
8	100–200
16	100–400

Table 5.6: Approximate range of costs for Shared-Region operations for different numbers of concurrent `Access` and `Done` operations (times in microseconds). Times are given as ranges, covering `Read` and `Write` and different protocols and region distributions for `Access`/`Done` pairs, but ignoring data coherence cost and lock contention. With high lock contention, times can reach several milliseconds.

5.3 Synthetic Programs

There are several advantages to using a synthetic workload instead of a comprehensive suite of real applications. First, less effort is required. To port existing applications requires significant effort because

1. Our system does not support Unix^{tm}.
2. Our system does not have a high quality FORTRAN compiler.
3. Our techniques require a slight modifications to the programs to use the Shared Regions approach.

Second, the behaviour of real programs is often difficult to understand at the level of detail necessary for our purposes. Finally, synthetic workloads are more flexible, in that they can be easily parameterized to adjust their behaviour in order to study specific effects in greater detail.

The main disadvantage of using synthetic programs is that it is often questionable how representative they are of real applications. We attempt to address this issue by patterning the synthetic programs after real applications, and verifying that they have similar performance characteristics. The main question therefore remains as to the representativeness of the real applications used as a basis for the synthetic ones.

Hence the approach taken here is to build a small collection of synthetic programs which can imitate common algorithms and access patterns in typical application kernels. By completely controlling data size, granularity, initial placement, and communication patterns, it is possible to emulate many different applications with complete control. Additionally, because of this control, although only kernels are tested, it is easy to construct more complex applications from the concatenation of these tests by simply choosing the tests with the appropriate starting and ending conditions.

One note of caution: although most of the applications are patterned after real computational kernels, the important aspect of the synthetic programs is not the problem they are solving, but the data access patterns they have. Whether they are implementing the most efficient method for solving the problem is not a major concern.

5.3.1 Examination of Synthetic Approach

We briefly outline the structures common to all the synthetic applications, and provide some support for their relevance by comparing one of the synthetic applications to a real version.

All the applications consist of the following:

- The data to be operated on. This consists of a small number of large contiguous portions of memory (each corresponding to a main data structure in the program, such as a matrix), each allocated from its own Hurricane region. Each of these objects is partitioned into a number of equal-sized shared-regions that will be the units of access for the synthetic application. The size and attributes of the Hurricane regions, the size and attributes of the shared-regions, and the initial placement of the data, are all controlled through parameters adjustable on an object-by-object basis.

- N threads, each placed on a separate processor, used to perform the parallel operations (all these programs are implemented using coarse grained parallelism).

- Calls to read and write the shared-regions. Currently only sequential read and write calls are provided. These calls allow the user to specify the percentage of the object to be accessed, and the delay, in cycles, between each 32-bit access to the data. This last option lets the user control the access intensity according to the algorithm being synthesized and the performance of the hardware. However, in all our tests (except the validation tests), we set the delay to 2 cycles. This is to reflect current trends in microprocessors which are reducing the cost of most arithmetic operations to one or two cycles.

Using these tools, a number of interesting access patterns (as found in many algorithms) can be examined in isolation or in combination with each other:

- 1 to N – broadcast.
- N from 1 – accumulation.
- N to N shift to neighbour – nearest-neighbour.
- N to N shift to anywhere – general redistribution.
- Random – non-locality.
- Gaussian – access intensity dropping off with distance.

5.3.2 Validation of Synthetic Approach

To show that these synthetic programs do a reasonable job of approximating the access patterns intended, we compare the response time of the implementation of the LU Decomposition problem with the synthetic version based on the same algorithm. This application was chosen because of its regular but relatively complex access patterns. A complete description of the algorithm is presented in the next section.

Since the synthetic version contains only data accesses and no computation, the computation must be replaced with suitable delays between accesses. The delay setting was chosen by comparing the performance of the two versions for one data size on one processor. A delay of 7 cycles between each word access was found to provide a good match. This delay was then used for all subsequent validation tests (but only for the validation tests).

Figure 5.5 shows the results for a number of tests using 360×360 and 480×480 double precision matrices. The data pages were distributed randomly among the processors. Two different shared-region protocols were compared:

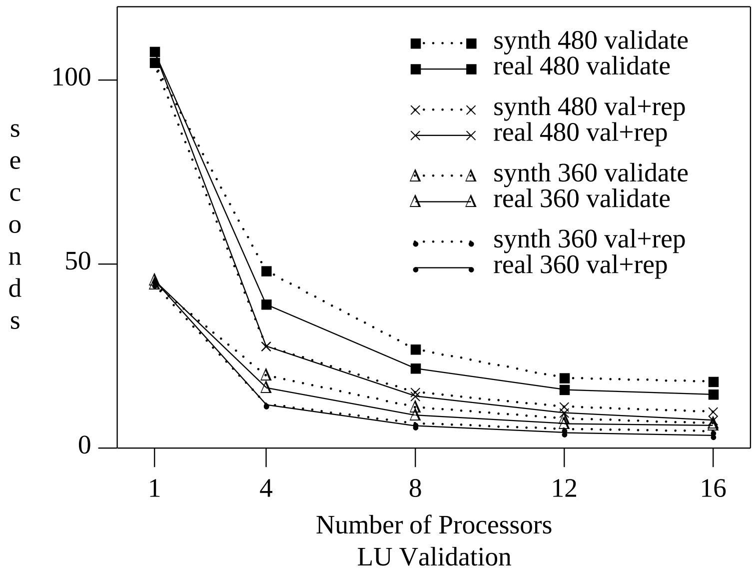


Figure 5.5: Comparison of actual LU Decomposition program to synthetic version of the same algorithm for different problems sizes and shared-region protocols.

- The validate cache coherence protocol, without any form of replication.
- The validate cache and memory coherence protocol, with full replication.

As the results show, the synthetic application follows the real application's curves reasonable closely through all variations. This should provide some confidence that the synthetic programs faithfully represent the important characteristics that they claim to implement.

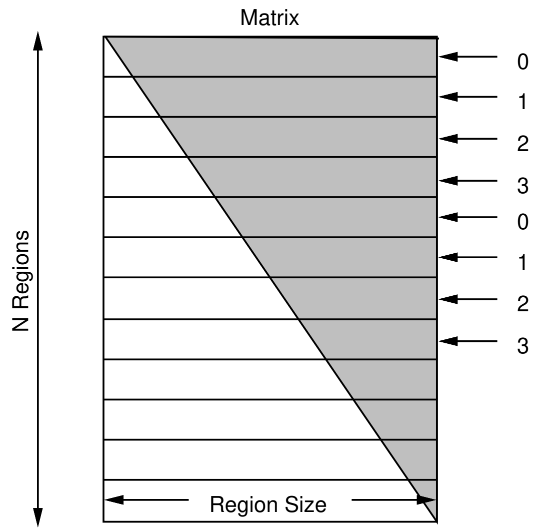
5.3.3 LU Decomposition

Row-oriented LU Decomposition decomposes a dense $N \times N$ matrix M into a lower triangular matrix L and an upper triangular matrix U. The sequential algorithm goes through $N-1$ phases. At phase i: 1) row i becomes the pivot row and is normalized, and 2) all rows j $(j > i)$ are reduced using the pivot row. This continues until all rows have become the pivot once. LU Decomposition is performed in parallel by assigning the rows of M to the P processors in a round-robin fashion (see Figure 6(a)). At each phase of the computation, the pivot row is normalized by the processor that "owns" it, and the pivot row is then used by all processors to reduce their own rows (Figure 6(b)).

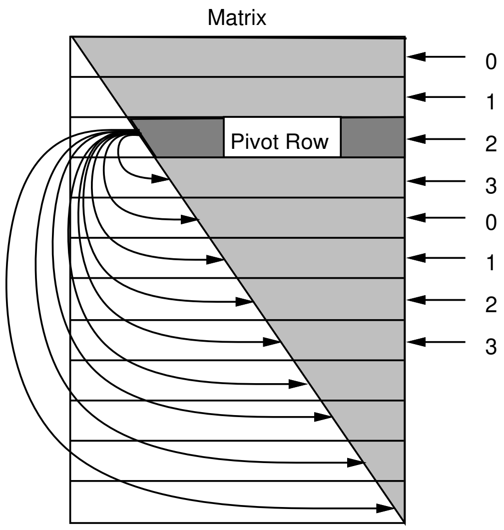
In the synthetic version, the algorithm remains the same, but the row computations are replaced with simple read and write accesses to the regions (mapped one-to-one with the rows). In addition, when we talk about an $N \times M$ problem, we are now referring to a problem with M regions, each of size N bytes. In this and all other synthetic applications, the delay between accesses is set to 2 cycles.

Some important aspects of this particular algorithm:

- The "owned" rows are read and written by their owning processor $O(M)$ times.
- Each row, when it becomes the pivot row, has an 1-to-P distribution.
- Each pivot row is read $O(M/P)$ times by each processor.



(a) Region partitioning and assignment with $P = 4$.



(b) Phase 2 of algorithm as processor 2 normalizes the pivot row.

Figure 5.6: Synthetic version of LU Decomposition.

- The overall time complexity is $O(M^2N)$.
- It is expected that the matrices will be large, with MN/P much larger than the processor cache. However it is also expected that at least two rows (the pivot row and the row to be reduced) will fit in the cache.

Because of these factors, there are two aspects to this program that have a strong influence on its performance: the placement of the rows with respect to the processor that "owns" them, and the method of distributing the pivot row to the other processors. Because the rows are expected to be significantly smaller than the cache, there is likely to be no benefit from replicating the pivot rows to the consuming processors. On the other hand, in the absence of replication, due to false sharing at the page level, it is not possible to place all of the rows local to each of their processors. In this case, the best that can be achieved is to ensure that each page is placed on a processor that owns a row located on that page. With shared-regions however, replication is not constrained by page-level false sharing, and hence every row can be placed in its owning processor's local memory.

We therefore intend to show the impact of locality in the placement (initial or dynamic) of the rows, the cost incurred when using shared-regions, and the impact of shared-regions on the way the pivot row is distributed and the cost of this distribution.

Four different coherence protocols are used:

- Validate: The basic cache-validate algorithm without replication.
- Validate with replication: cache and memory validate protocol with full replication on all references.
- Update with replication on write: Update protocol for cache and memory, with replication on writes only (hence pivot row accesses by readers does not cause a replication).

- No Coherence with replication: this might be considered overly optimistic, since perfect locality is obtained using replication, but all coherence concerns are ignored. Note that this is an unrealizable case for comparison purposes only.

These three different protocols were compared using a number of different initial placement options, including what is considered to be the optimal placement given protocol restrictions (for example, lack of replication). One of the goals was to determine the sensitivity of the various protocols to the initial conditions. This is of great importance since most applications involve multiple transformations of a single data structure, and what is optimal for one stage may not be for the next stage. The following placement options were tested:

- The rows are placed round-robin, matching the allocation of rows to processors. However, since the matrix is allocated as a single contiguous chunk, if replication is not used, then constraints imposed by using page-level coherence requires that only the first row on a page can actually be located on its owning processor. If replication is used, then the page will simply get replicated to each processor that owns a row on that page.

- The pages of the matrix are assigned round-robin among the processors ignoring the allocation of rows. This essentially distributes the data randomly among the participating processors.

- The matrix is placed entirely on a single processor.

- The matrix is pre-replicated to all participating processors.

Figures 5.7(a)-(d) show speedup results for these various options on a 1840×700 matrix (the size 1840 is essentially a random size that is not a multiple or divisor of the page size). All speedups are relative to the best sequential time.

Locality of the "owned" rows is very important as is shown in Figure 5.7(a). Here, no replication is used, so at best, only partial locality can be achieved. In particular note how the speedup curve for the case where all the data is placed on a single processor is flat, clearly indicating that the memory has become saturated. This completely flat speedup follows from the linear growth in response time for hotspot accesses that was observed in Figure 5.1(a). When the data is distributed without considering locality, performance improves, but it improves still further if locality is taken into account by ensuring that the pages are placed on at least one of the owning processors. It is worthwhile pointing out that one of the reasons for the strong impact of locality is the high cost of on-station cache writebacks, as shown in Table 5.1. This means that, to a certain extent, the results are exaggerated.

Turning our attention to the version using full replication (Figure 5.7(b)), we notice that the performance is improved (the best from the previous version is presented as the dotted line below the others). Perhaps more importantly, the performance appears insensitive to the choice of initial placement, which is one of the prime advantages of dynamic memory management. Unfortunately, performance still suffers because the pivot row, once normalized, is replicated on demand to each processor. These replications are generally not required since the pivot row is fully cached. In addition, the replications tend to occur at roughly the same time, slowing the replications down because of the hotspot contention.

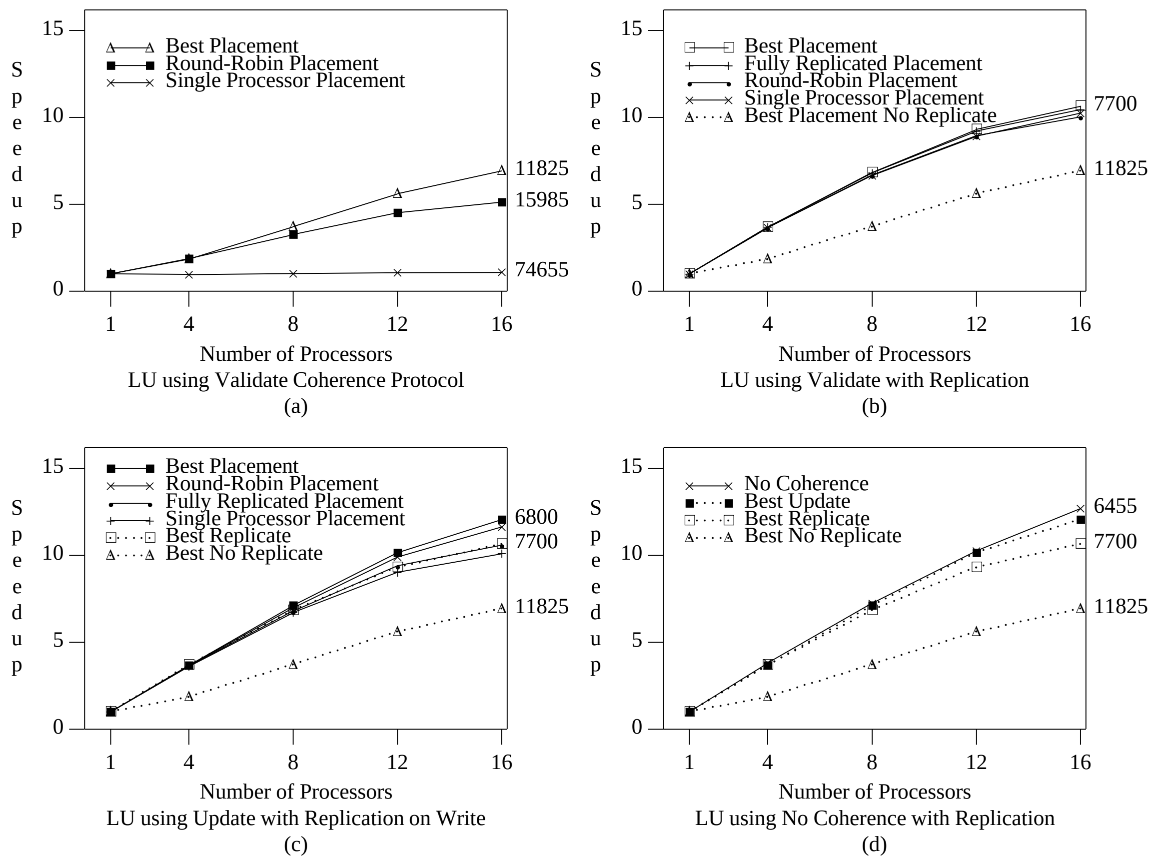


Figure 5.7: Speedup results from various protocols on a synthetic version of LU for a problem size of 1840×700. All speedups are based on the best sequential implementation of the algorithm. Response time in milliseconds is shown at the right of some of the curves.

This then brings us to the third protocol considered, the update protocol with replication on write accesses only (Figure 5.7(c)). With this protocol, all copies (including those created by accident due to false sharing) must be updated when the writing phase is done, but it means we can use a protocol that only performs replications on write accesses to the data. This means that only the "owners" of the rows (the only processors that actually write to them) will obtain copies of the data. Since the rows are large enough that there are at most three rows on a page, there are generally only two extra copies to be updated. This version shows respectable performance, but it is now somewhat sensitive to the initial placement options. In particular, when the data is replicated to all processors, many extra copies are updated after each pivot is completed, slowing down both the processor completing the pivot row, and all processors waiting for the pivot row. Additionally, when the data is placed initially on a single processor, the cost to do the initial replication on demand adds significantly to the overall cost. This is partly due to the fact that this algorithm relies on replication at the page level in the operating system (the validate replication protocol does not), and as we have seen, the memory management implementation used in this work is less than optimal.

Finally, in Figure 5.7(d) we compare the best of the previous three protocols, with the No Coherence protocol that replicates pages to keep data local, but incurs no coherence overhead. As we can see, although No Coherence outperforms the best of the three previous protocols, it does so only by a small margin. This indicates that any limit in speedup is not due to the use of shared-regions.

Figures 5.8(a)-(c) show the effect of varying the region size and the number of regions on the performance of the three main protocols, using the best placement for each. Since the synthetic version does not depend on the matrix being square, different combinations of row sizes and column sizes (region sizes and number of regions) were tried, as each presents different trade-offs in terms of overhead per data item and reuse of data.

Figure 5.8(a) shows the results for a small number of large regions. For this problem, there is not much reuse, so the benefit of locality does not easily cover the cost of replication. Full replication performs very poorly because of the cost of replicating the pivot row. Figure 5.8(b), however, shows that for LU problems with large numbers of regions, and hence plenty of reuse, the benefits of replication are significant. Even the extra replications of the pivot row performed by the full replication protocol are insignificant compared to the benefits of locality.

Finally, in Figure 5.8(c) we see that when the region is exactly one page (4096 bytes), rows can be assigned optimally to processors without using replication. The result is that the protocol without replication performs the best, since it gets perfect placement for free (the initial placement cost is not charged to the application). However, note that compared to the Update protocol, the difference is insignificant, and depends on the rows having this special size. The replicate protocol performs very poorly however, since it must replicate the pivot row (a full page) to each processor on every iteration, and gains nothing compared to the others in terms of locality. This shows that the overhead for using regions is not significant for this problem, provided the protocol matches the needs of the application.

In summary, replication is effective for this application, but how effective is dependent on the reuse factor, which in turn is dependent on the number of rows in the matrix. Additionally, it can be seen that replication where not necessary (distribution of pivot row) can have a significant negative impact, suggesting some caution in the application of replication, and the value of a more flexible replication policy.

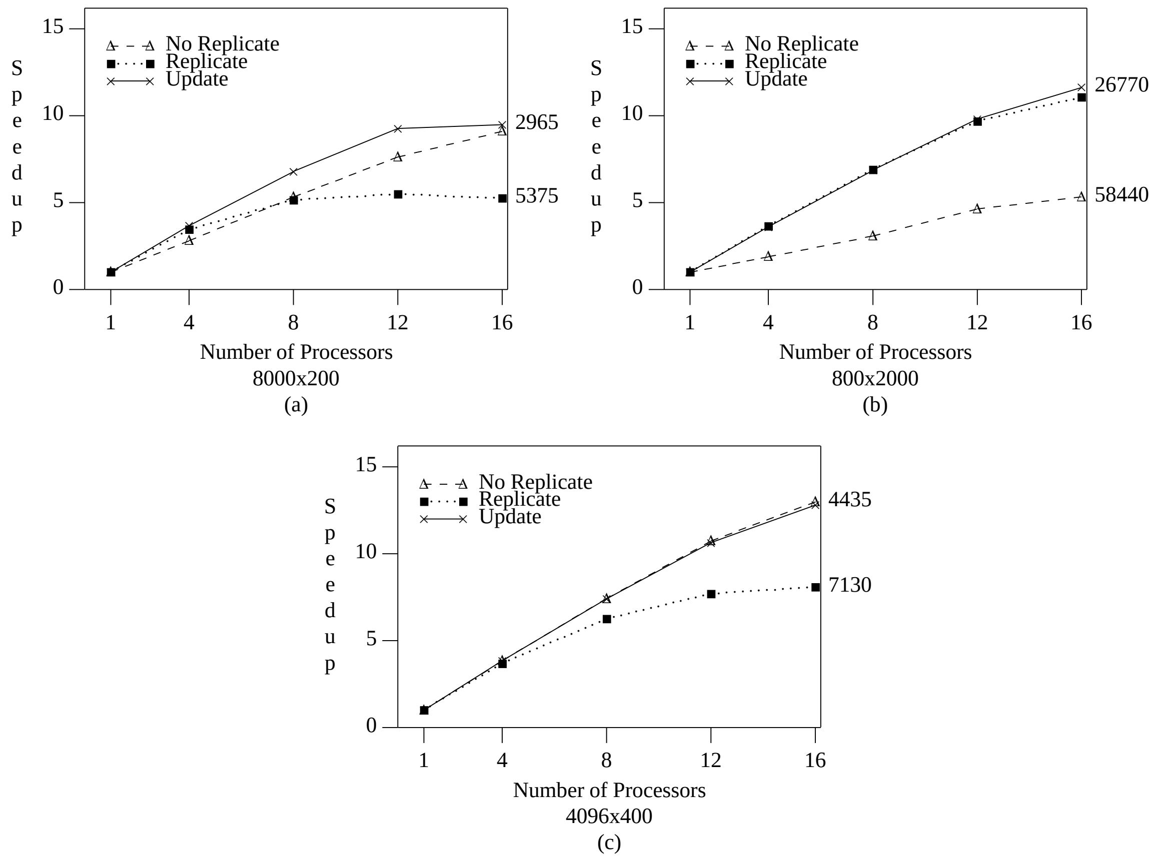
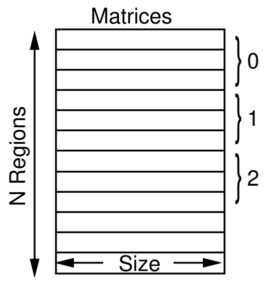
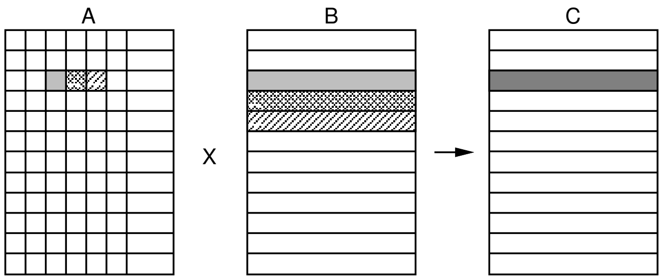


Figure 5.8: LU: best initial placement results for each of the protocols on different sized problems.



(a) Region partitioning and assignment.



(b) Each element i of row 3 (only 3,4,5 shown in picture) of A is combined with the corresponding rows (3,4,5 shown) of B and summed to give row 3 of C. This computation is performed by the owner of row 3 for A and C

Figure 5.9: Synthetic version of Matrix Multiplication.

5.3.4 Matrix Multiplication

This synthetic program performs the standard multiplication of two matrices producing a third ($A \times B \rightarrow C$). As with LU, this is just an application kernel, not an application itself. There are many ways of performing the multiplication. Because of the benefits of the cache, and the small cache size, one of the faster uniprocessor algorithms on Hector is to compute each row of C by successively combining elements from row i of A with each row j of B producing j partial sums which together give a full row of C (see Figure 9(b)). With this technique, B is read in row order, which matches the memory allocation strategy of the C language, and helps improve the effectiveness of the cache (it is assumed that B is much larger than the processor cache).

As with LU, the rows of the different matrices are designated as individual regions (see Figure 9(a)). Each processor is assigned a group of N/P contiguous rows from C for which it is responsible. Because only the access patterns are implemented, the matrices do not need to be of compatible sizes: for these tests all matrices are the same dimensions, and are not necessarily square. Because all rows of matrix B are needed to compute each row of C, the entire matrix B ends up being read N/P times, and is therefore responsible for the largest amount of traffic. As will be shown, the traffic from the other matrices is negligible, and, because of the high locality, easy to deal with.

Figure 5.10(a) shows the impact of placement of both the A and C matrices. To highlight any effect, B was pre-replicated to all processors. Nonetheless, the placement of A and C, even when restricted to a single processor, has little effect on performance. Because of this, all further tests were done with optimal placement of A and C (the most likely scenario).

It is also worth noting that the program does achieve nearly perfect speedup using pre-replication of B. This demonstrates that solving the distribution problem for B (how to get the matrix to all the processors as efficiently as possible) should be sufficient to give us high performance for this problem.

With this in mind, we turn our attention to the different protocols and placements:

- Cache validate with no replication.
- Validate with full replication.
- Validate with station replication.

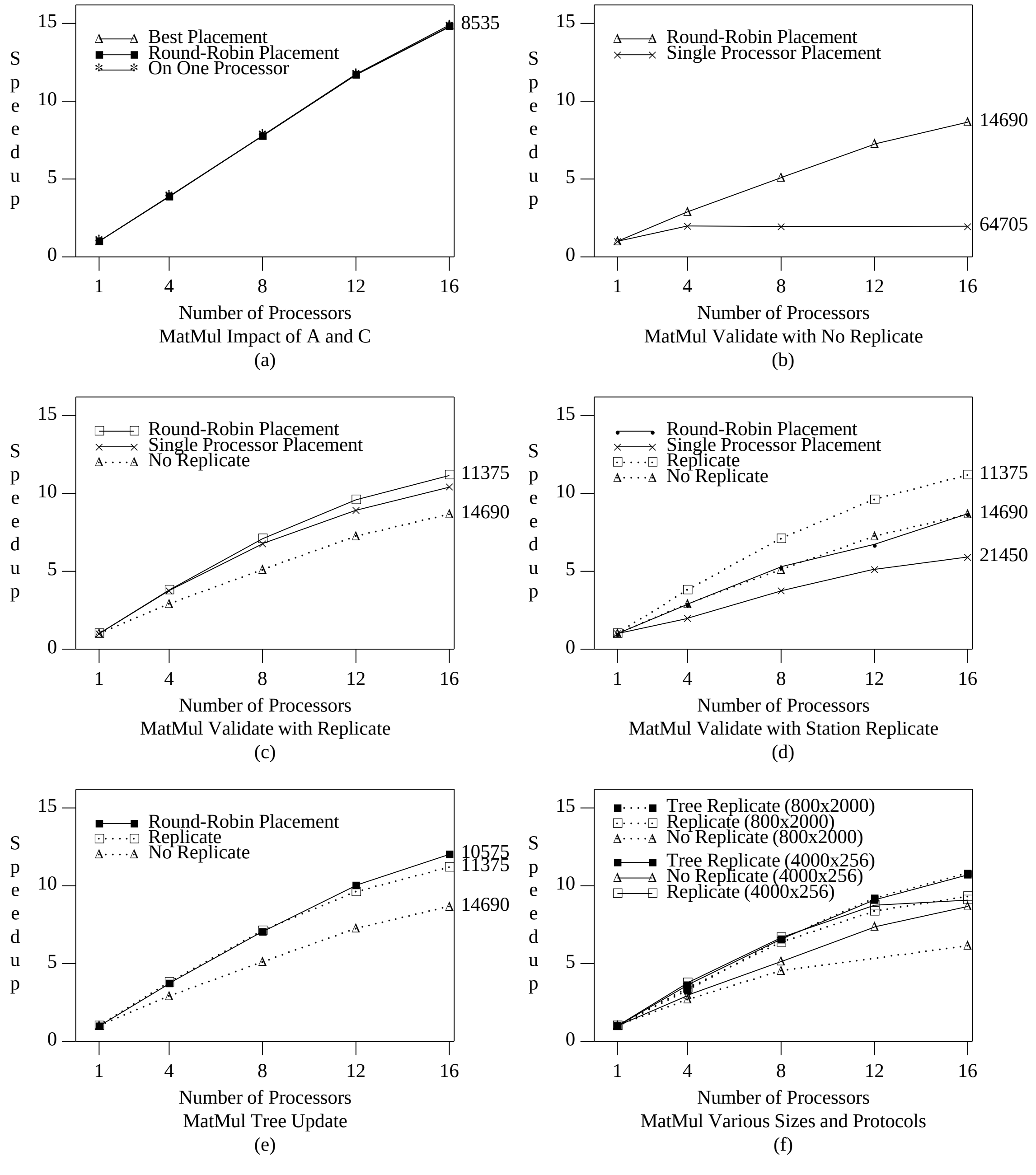


Figure 5.10: Speedup results for various protocols on a synthetic version of Matrix Multiplication for a problem size of 3008×500. All speedups are based on the best sequential implementation of the algorithm. Response time in milliseconds is shown at the right of some of the curves.

- Update using a tree broadcast implementation.

Two different initial placement options (for B) were examined:

- Round-robin by page.
- All data placed on a single processor.

Figure 5.10(b) shows what happens when replication is not used. When the B matrix is placed on a single processor, the memory quickly becomes saturated, and as with LU, the speedup curve remains flat. When the data is evenly spread out, performance improves, but remains poor. From these results, it should be clear that the problem is not just the additional latency of the remote accesses (30 cycles instead of 20); contention at memory and to some extent in the interconnection network is primarily to blame.

When the replication protocol is used, we see a noticeable improvement (Figure 5.10(c)). After the first access, at which point the row is fully replicated, all further accesses to B incur no additional cost (latency or contention). Initial placement does have a small but noticeable impact however, since replicating in parallel from a single memory module is expensive as already shown.

In an attempt to strike a better balance between locality and replication cost, the station replication protocol was considered (Figure 5.10(d)). Here replication only occurs if the data is in a different station. This should result in fewer replications, and keep accesses relatively close. However, the current implementation does not handle concurrent station replications as efficiently as it might, and the cost of in-station accesses compared to off-station accesses is not large enough to make up the difference. More importantly however, the lower memory contention expected does not materialize. The problem is that the first processor to access the first row of B replicates it to its own memory. All the other processors in the same station must wait for this replication to complete. Once completed, the processor that now has the row local will execute faster, and will likely access the next row first. That row too will be replicated onto the same processor. In the end, all rows of B end up on a single processor in each station, and performance suffers accordingly. For this problem, a round-robin in-station allocation scheme for replicas would probably be more appropriate. Additionally, in larger systems with greater network and memory contention, partial replication of this sort is likely to prove more useful, especially considering its lower memory requirements.

Finally, since full replication seems to provide the best performance, but suffers from the cost of the initial replication, a new protocol that uses the update replication protocol, was considered. With this particular protocol, the fact that the data, once it has been initialized, will be needed by the other processors, is recorded in the region descriptor. When the `WriteDone` for each region is performed during initialization, the data is immediately replicated to all processors, using the parallel tree replication method (as described in Section 4.3.3). The full initialization phase is included in the final cost of the algorithm. As can be seen in Figure 5.10(e), this version performs slightly worse for small numbers of processors, but is better starting around 12 processors; it appears to have a lower slope due to the logarithmic nature of the replication scheme.

One of the most important factors in assessing the benefits of replication is the expected reuse rate of each region. Another important factor is the cost of replication, which is proportionally greater for smaller regions due to the fixed overheads. To examine these effects, the size of each region (row) and the number of regions (rows) were varied (Figure 5.10(f)). As expected, replication still outperforms the non-replication version, and performs better on problems with more regions because of the greater reuse (reuse being $O(N/P)$). With small numbers of regions, the reuse becomes low enough that the cost of replication only barely offsets the gain when using the non-tree based replication scheme, and the high contention during replication shows a pronounced effect on the overall performance with 12 and 16 processors. Interestingly, the tree replication version seems mostly unaffected by the different sizes and numbers of regions, although its performance is still lower when compared to the 3008×500 sized problem used in the other examples.

In fact, overall performance is significantly below optimal for all cases examined. This is primarily due to the small size of the problems used for experimentation purposes. Such small sizes were required partly because of the limited memory available per processor (4 Megabytes), partly to keep execution time within reasonable bounds to allow multiple experiments to complete within a short time, and partly because of limitations in the memory manager which prevents us from freeing replicated pages.

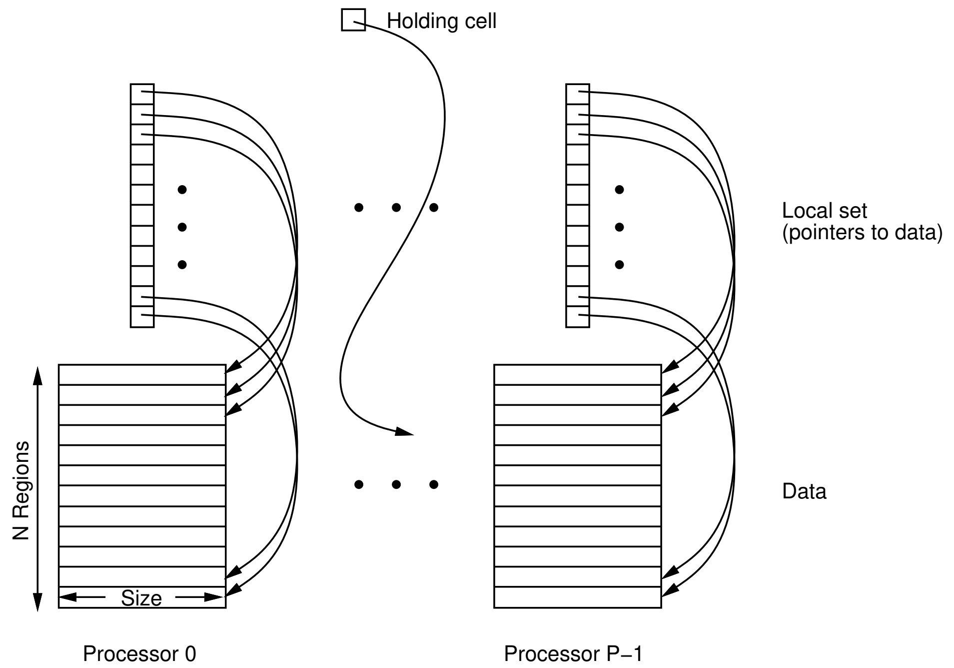
This brings up one final issue. Although full replication provides superior performance for this problem, it also requires a lot of memory, and for many problems, there will not be enough local memory to support such a protocol. Partial replication can address part of the problem by reducing memory requirements by a constant factor, but a complete solution requires two additional features: first, the application needs to use a blocked algorithm, similar to the blocked algorithms that take advantage of the processor's cache, but now to take advantage of local memory (which is acting essentially as a cache); and second, the memory manager needs to implement efficient means of releasing memory that has been replicated, similar to the ejection of cache lines due to capacity misses (note that with shared-regions, releasing pages is made trickier due to the disjoint requirements of region and page coherence).

5.3.5 Migrating Locality

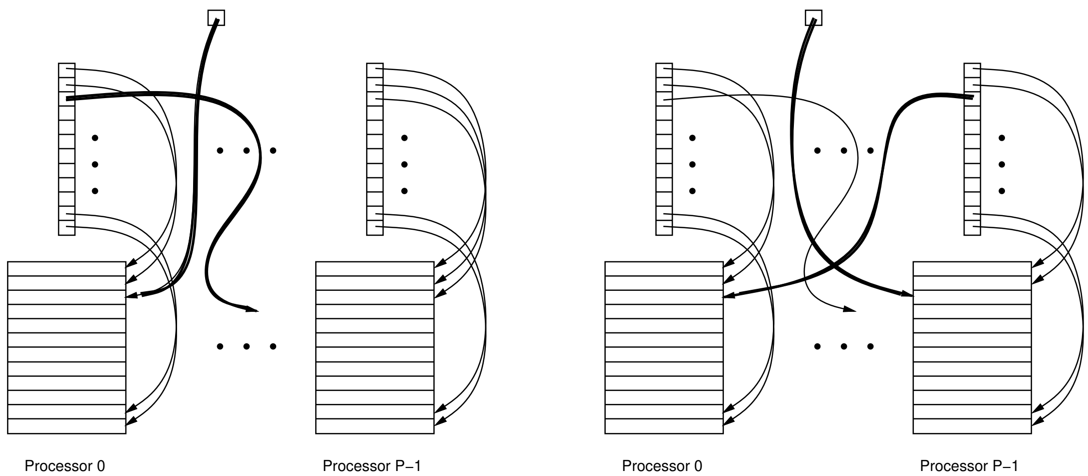
This program is patterned in some ways after n-body simulation programs, but only to a limited extent. Its essential characteristic is that each processor has a local working set which moves around the shared data space over time, exhibiting control over locality both in space and time.

The structure of the application is as follows. A fixed number of equally sized regions are placed in each processor's memory. Each process (one per processor) keeps a private list of the regions that are in its set. Initially this list corresponds directly to the regions that have been placed on its processor (see Figure 11(a)). Each process then repeatedly performs the following operations (as graphically illustrated in Figures 11(b) and 11(c)),

```
Pick a random number p
If p < psame Then
   read/write the last region accessed
Else
   Pick a new random number p
   If p < plocal Then
      read/write some random region in local set
   Else
      switch a random local region with the global ''holding cell''
      read/write the new region
```



(a) Initial configuration.



(b) Processor 0 swaps an element (shown with thicker lines).

(c) Processor P-1 swaps an element (shown with thicker lines).

Figure 5.11: Migrating Locality synthetic program.

```
    Endif
EndIf
```

The variables psame and plocal are user-controlled parameters. The "holding cell" is a global variable that points to some region on some processor. As each process switches one of its region pointers with the one in the holding cell, region pointers are effectively exchanged among processes, and the locality present in the initial assignment slowly disintegrates. The more iterations the program runs, and the higher the probability of doing a switch with the holding cell, the more of the original locality in the mapping between regions and processors is lost. Additionally, the more frequent the switching, the less temporal locality there is, since the local set is constantly changing. This therefore gives us a program with a floating working set, whose temporal and spatial locality can be controlled.

Due to the structure of this application, the amount of work performed (in terms of application visible work) by each process is independent of the number of processors involved. Hence this is a constant work-per-processor test, so that a perfect system would show equal response time no matter how many processors were involved. Since the application generally slows down as more processors are added, results are shown as slowdown (rather than speedup) with respect to the best single processor version. Hence a flat line with low values near 1 is the goal.

A number of the parameters that could be varied have been frozen at certain "reasonable" values for these tests:

- The number of regions and their size are chosen so that the data associated with a processor is larger than the cache, but *all* data can still fit in one processor's memory when replication kicks in.
- The number of iterations is fixed at 40000; which is large enough to give consistent and reliable results, but small enough so that tests complete within a few seconds to a few minutes.
- The size of the regions is fixed at 2 KB.
- The data is always placed on the processor that initially owns it.

There are three key parameters that were studied:

psame. Controls the probability that the same region that was accessed in the last iteration will be accessed this iteration.

plocal. Given that we are not required to access the same region as last time (probability $1 - \texttt{psame}$), this parameter controls the probability that the region we choose is from our local set (it could still end up being the same region, however). With probability $(1 - \texttt{psame})(1 - \texttt{plocal})$ we switch a random element in the local set with the holding cell. The new region is then used for this iteration's access.

pwrite. This is the probability that we write to the data. If a write is to be performed, the region is first accessed using WriteAccess. Then the entire region is read, then written, and finally closed using a WriteDone. For a read, ReadAccess and ReadDone are used, and no writes are performed.

The impact of these parameters on the performance of the Migrating Locality program can be found in three main areas:

- Cache hit rate. This closely follows `psame` in the sense that the higher the probability of reusing the same region, the higher the cache hit rate. However, `plocal` also has a noticeable impact on cache hit rate, in that it keeps accesses within the local set thus increasing the chance of finding the region already in the cache (because of the relatively small size of each process' set). The number of processors used in the test can also affects the cache hit rate, since the hit rate will in part depend on the ratio of the cache size to the total data size.

- Spatial locality. This refers to how much of the locality in the original region distribution is reflected in the program's access patterns. More specifically, it is the ratio of the number of accesses to (physically) local regions over the total number of accesses, over the entire duration of the program. When `psame` is high, spatial locality is unimportant. When `psame` is low, `plocal` controls the probability of performing a switch, which reduces the spatial locality. With high iteration counts, more switches are performed, which reduces the original spatial locality. Hence `psame`, `plocal`, and the number of iterations together control the spatial locality.

- Temporal locality. This is $(I - S)/I$, where I is the total number of iterations (or accesses), and S is the total number of switches (a switch occurs when a local region pointer is exchanged with the global holding cell). It reflects the stability of the local set, and thus the expected reuse of local regions. It is determined by `psame` and `plocal` which together determine the rate at which regions are switched in and out of the local sets. The switching rate, S/I, is more commonly used here, however.

As with the other programs, a number of different region protocols were examined:

- Cache validate with no replication.

- Validate with full replication.

- Validate with station replication.

- No Coherence. The regions are used, but no protocol processing or synchronization operations are performed. No replication is used. Since no coherence is maintained, this can be considered an upper bound on performance for the no replication case.

The No Coherence option requires a special note. Because of the limitations of the current cache management facilities provided by the shared-regions, dirty data is always copied back to main memory at `WriteDone` time. For the previous tests, this was not significant, but for some combination of parameters for the Migrating Locality program, this can add greatly to the cost, especially when `psame` is high. With No Coherence however, coherence is not maintained, so that no copyback takes place.

Figure 5.12 demonstrates the effect of `plocal` and `psame` on the spatial locality and switching rate. The X-axis shows $1 - $ `plocal` on a logarithmic scale, to highlight the areas of interest. The effect of the two parameters on spatial locality and switching rate is seen in the descending slope of the "local" curves, and the ascending slope of the "switch" curves, as well as in the increased locality when `psame` is high, and the increased switching rate when `psame` is low.

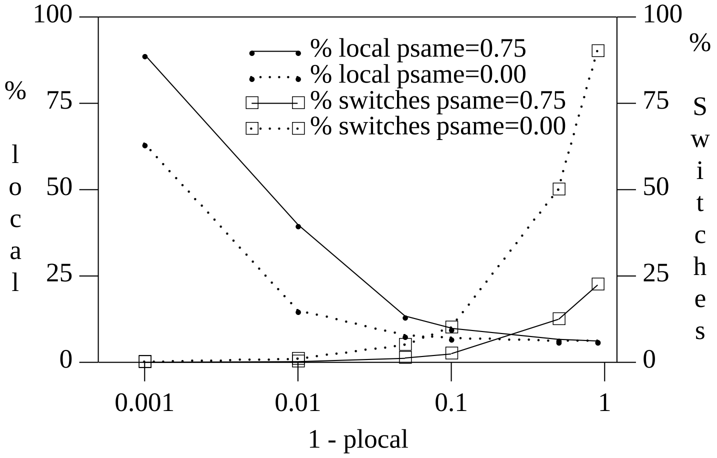


Figure 5.12: Effect of `psame` and `plocal` on spatial locality and switching rate.

Figure 5.13 shows the effect of varying the number of processors for different parameter settings. In Figure 5.13(a) the performance with `psame` $= 0.0$, `plocal` $= 1.0$, and `pwrite` $=$ 1.0 is shown. Since the data is always local, the only reason for slowdown is due to contention at the memory module where the region descriptors are allocated (unfortunately all regions descriptors end up allocated on a single memory module in the current implementation, so the overhead seen in the tests is primarily due to the implementation and not the algorithm). This can be seen from the way all versions that use regions grow together as the number of processors increases, whereas the No Coherence version remains flat. In Figure 5.13(b) we see the effect of lowering `plocal` to 0.999. This results in about 35 percent of all accesses going to some remote memory, enough to have a noticeable effect. In particular, note how the Validate version performs the worst, while the two versions using replication perform comparably better, even occasionally outperforming the No Coherence case, despite the extra overheads due to region accesses and copybacks at `WriteDone` time.

In Figures 5.13(c)-(d) results for the same two tests, but with `psame` set to 0.75 are shown. Here the savings that No Coherence sees by not having to do copybacks at `WriteDone` time is significant, and swamps all other factors. Ignoring the No Coherence version then, in the case with `plocal` $= 1.0$, we see a slight advantage of the Validate protocol over the other two region protocols. The Validate protocol, being simpler, has slightly lower overheads. With `plocal` dropped to 0.999 however, about 10 percent of all accesses are to remote memories (thus we have higher spatial locality than we did with `psame` $= 0.0$). This is enough to show the advantage of replication once again, as the Full Replication and Station Replication protocols turn out better performances than the simple Validate protocol.

Although slowdown is quite severe in many of these tests, this is primarily due to a limitation of the caching algorithm, as can be seen in comparisons with the No Coherence policy.

To determine the effect of locality and switching rate on performance, Figure 5.14(a) and (b) show slowdown results on 16 processors with `psame` fixed at 0.0 and 0.75, and `plocal` varying from 0.999 to 0.1. Similar to Figure 5.12, the X-axis uses a logarithmic scale and shows $1 -$ `plocal`.

Some general conclusions that can be drawn from these two graphs:

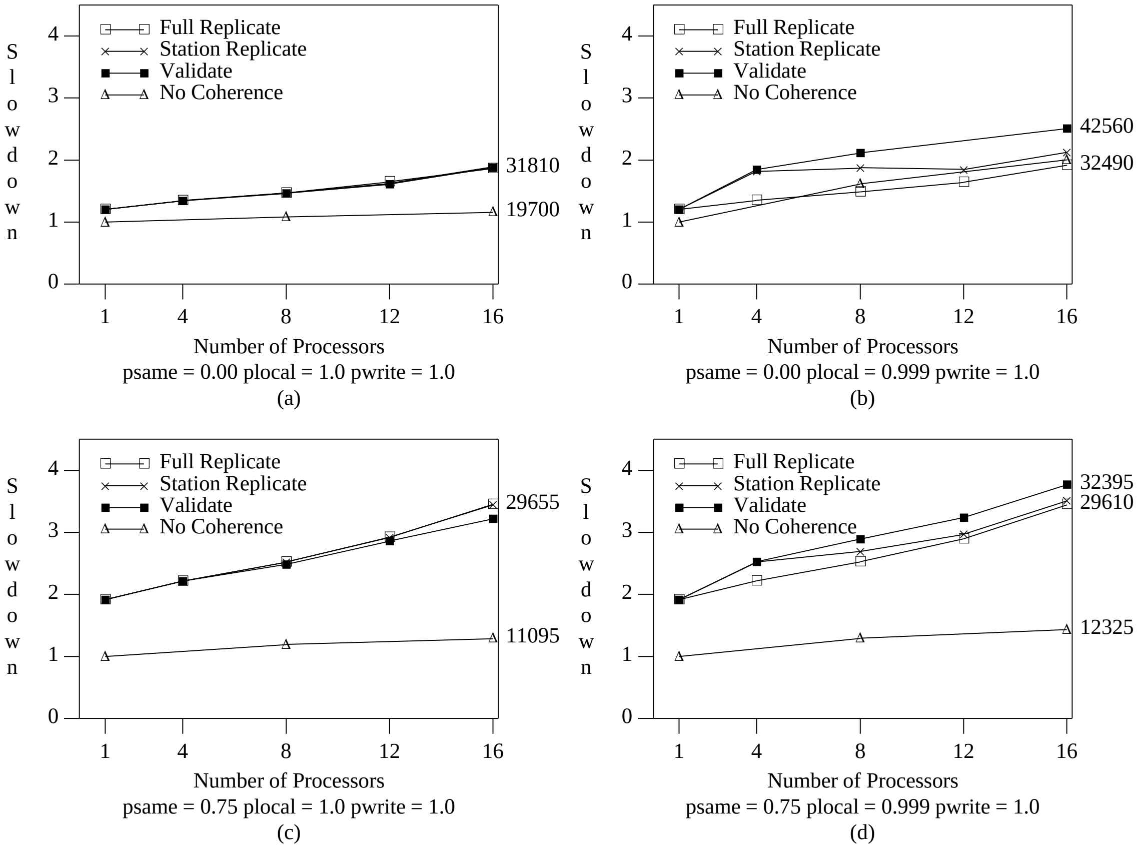


Figure 5.13: Impact on Migrating Locality program of `psame` and `plocal` for varying numbers of processors.

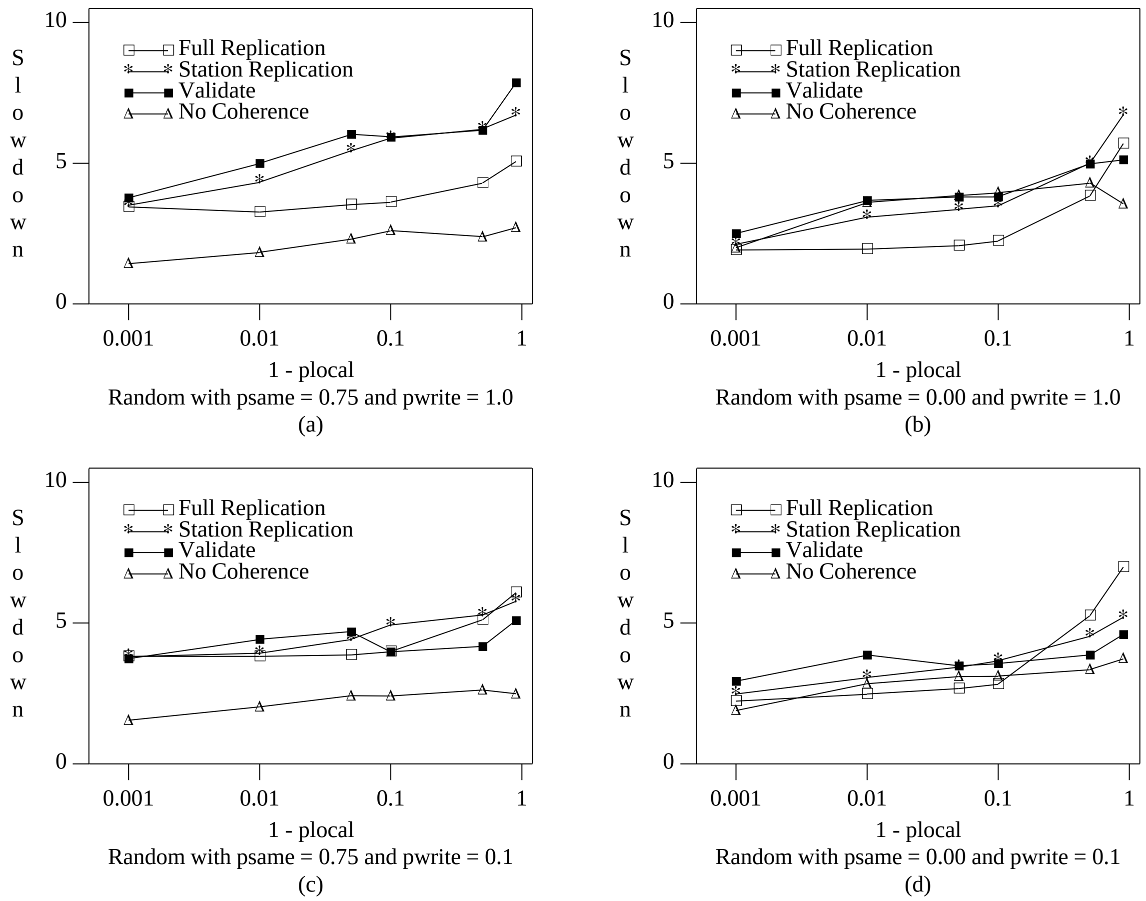


Figure 5.14: Tests of the Migrating Locality program with varying parameter settings. All results presented as slowdown on 16 processors.

- No Coherence does well when the cache hit rate is high, but suffers due to large numbers of remote accesses and copybacks as a result of capacity misses, when the hit rate is low.

- Validate without replication does fairly poorly overall. It pays the region and coherence overhead, but does not gain any locality. As `plocal` decreases, spatial locality is quickly lost. As it decreases further, the decreasing temporal locality begins to be felt as the number of coherence operations required increases, and performance drops even further.

- Validate with station replication performs similarly to the Validate protocol without replication. Although it does improves on the no-replication case slightly, the extra locality of the station is not sufficient to make much of a difference. In particular, on average, 75 percent of all copybacks are still to non-local (on-station) memory, and we have already seen that on-station copybacks are actually more expensive than off-station copybacks (Table 5.1). Additionally, the application has no locality beyond processor locality, so it cannot take advantage of the benefits of station-local data. For algorithms that have a more Gaussian distribution of accesses (rather than a simple local/non-local distribution as for this program) the benefits of station replication might be felt more strongly.

- Validate with full replication performs quite well overall. Although it cannot match the performance of the No Coherence version due to the cost of the copybacks when the cache hit rate is sufficiently high, it outperforms all the others in almost all cases. The one exception is when the switching rate becomes so high that reuse drops very low and the cost of replicating is not made up for by subsequent local accesses. This requires a very high switching rate of about 1 switch every 2 accesses however.

Finally, in Figures 5.14(c) and (d) we have the same tests except with `pwrite` = 0.1 instead of 1.0. Essentially the same trend can be seen, except that the benefits of locality are not as strongly felt in this case, since fewer expensive non-local memory operations (copybacks following `WriteDone`) are performed. As a result, the switching rate does not need to get very high before the cost of replicating outweighs its benefits (for both full replication and station replication). One interesting result, however, is that because of the lower number of copybacks, the benefits of station replication are more pronounced, and with `psame` = 0.0, station replication remains steadily in the middle in terms of performance, outperforming the non-replicating version when the switching rate is low, while outperforming the full replication version when the switching rate is high.

In summary, this application shows explicitly the relationship between spatial and temporal locality and the performance of region coherence. When spatial locality is high, region replication is unnecessary since caching is effective. When spatial locality is low but temporal locality is high, region replication can take advantage of this temporal locality and increase performance. When both spatial and temporal locality are low, region replication can significantly decrease performance, due to excessive data movement.

5.4 Summary

From the experimental results presented in this chapter, a number of conclusions can be drawn:

- Replication is effective in cases where access intensity and reuse are both high, since accesses following the initial replication are local and see neither the cost of remote accesses nor the effects of contention.
- For data that is the source of a hot spot, tree-based replication is an effective means of distributing the data to all the readers.
- When there is little reuse, the cost of Shared Regions' coherence protocols can severely degrade performance.
- A mismatch between the chosen coherence protocol or replication strategy and the access pattern can result in poor performance.
- Shared Regions are effective in eliminating the problem of false sharing.
- The overhead of Shared Regions is minimal.

Chapter 6

Larger Problems and Systems

In this chapter we extend the results obtained in the previous chapter using two different methods. We first present experimental results on a version of Hector modified to study the impact of the memory management techniques on larger systems. This is followed by the development of a model of the hardware and software system, allowing us to further extend the results already obtained to both larger problem sizes and larger hardware systems.

6.1 Delay Hardware

One of the limitations of the experimental approach taken in Chapter 5 is that it is often difficult to extrapolate from the measured results to other system environments. With Hector, however, it is possible to control the speed of the network to a limited degree, thus gaining some insight into the expected performance on larger systems of the same design, and to some degree into shared-memory multiprocessors of slightly different design.

The system provides a set of switches which allows the user to add additional delays to off-station packets, from 1 to 16 cycles, in powers of 2. Every packet that is destined for an off-station processor-memory pair (PM) is held up at the source processor for the number of cycles selected. Packets destined for an on-station processor are unaffected. Once in transit, packets are unaffected by the delays, irrespective of their source or destination. The intent of these delays is to make the stations appear further away from each other as in a larger system. It also more closely resembles the remote latencies in more complex cache-coherent systems such as KSR [BIFKR92] and DASH [LLG+90].

Table 6.1 shows the latencies of the common memory operations with the delays. To explain the effect of the delays, the following needs to be understood:

- A 32 bit read requires one request packet, and one response packet.
- A 32 bit write requires one request packet, but no response packet.
- A cache line read requires one request packet, and two response packets (carrying 64 bits of data each).
- A cache line writeback requires 4 request packets (carrying 32 bits of data each), but no response packets.

Assuming that each packet (request or response) incurs a delay of d cycles, a single word read is therefore delayed by $2d$ cycles, a single word write by d cycles, a cache line read by $3d$ cycles, and a cache line write by $4d$ cycles.

	32bit load	32bit store	cache load	cache writeback
local	10	10	19	19
station	19	9	29	62
ring	$19 + ks + 2d$	$9 + ks + d$	$29 + ks + 3d$	$10 + 4(4 + s + d)$

Table 6.1: Memory access times with hardware delays (in 60ns cycles). s is the number of stations in the multiprocessor, while k is a cost factor between 1 and 2 which depends on the position of the source and destination stations with respect to each other. Accesses that traverse the ring (off-station requests) incur an additional delay of d cycles for each packet required. Note once again the discrepancy between station writebacks and ring writebacks due to the effect of retries.

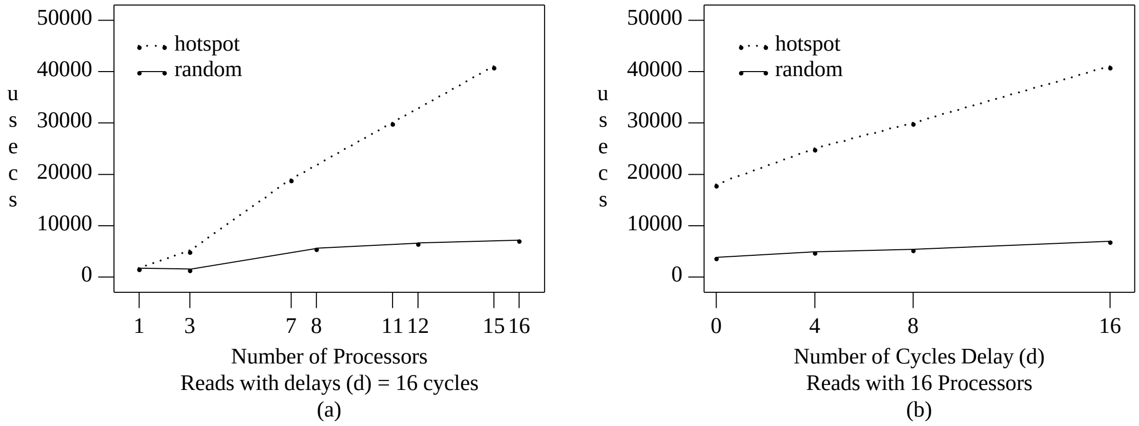
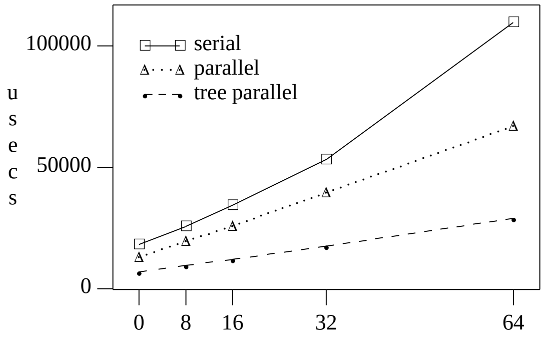


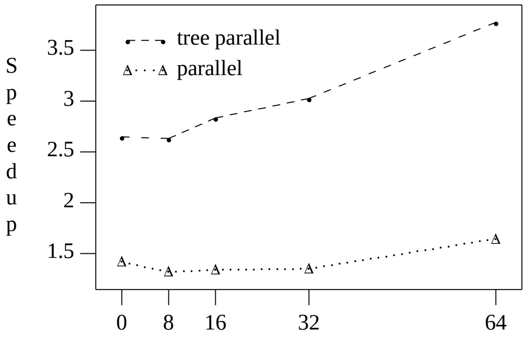
Figure 6.1: Measurements of the effects of hardware delays on memory performance (time in microseconds). Data points for the hotspot case in (a) are presented for 3, 7, 11, and 15 processors because the local processor has little effect on the results, and so was excluded from the tests.

An unexpected side effect of these delays is that, because of a lack of buffers in the memory subsystem, the memory is actually held busy for the duration of the extra delays for each response packet. This means that the memory is not accessible for an additional d cycles when an off-station 32 bit read occurs, and $2d$ cycles for a cache load. The memory busy time is not affected by writes because no response packets are generated. This means that contention does not decrease with the increased latency, but in fact often increases. Figure 6.1(a) shows the effects of a delay setting of 16 cycles on random reads (accesses randomly distributed among the PMs) and hotspot reads (all accesses to a single PM). Apart from the lower slope up to 3 processors (where all accesses remain within a single station, and hence the delays do not take effect) what we see in this graph is very similar to the results with no delays: hotspot response time increases linearly with a fairly steep slope, while random accesses increase at a much slower rate. In Figure 6.1(b) the effect of varying delays is shown on the performance of hotspot and random reads using 16 processors. Since the time for hotspot reads increases far more rapidly than the time for random reads, it seems safe to conclude that contention is increasing with higher delays. The effect cannot be due solely to the increased delay to reach memory or we would expect to see the random access times increase at a similar rate.

The effect of slower remote memory requests and higher memory busy times affects



Response time
(a)



Speedup of parallel versions over linear version
(b)

Figure 6.2: CopyRange costs for varying techniques and delays, for 15 copies (times in microseconds).

the performance of all software subsystems, including the kernel and the region operations (neither of which are currently configured to take advantage of locality). Most importantly to this thesis, all data copying operations that cross station boundaries are strongly affected. Figures 6.2(a) and (b) show the effect of the delays on the time to update 15 copies of a 4K region using different CopyRange copying schemes. The hardware delays for this particular test were varied from 8 to 64 cycles (delays up to 64 cycles were available for this one test only). The copy methods used were:

serial. The data is copied to each processor in turn (the cache is used for the source of the copy to reduce memory accesses).

parallel. Each destination processor is requested to perform the copy in parallel.

tree parallel. The tree broadcast scheme described in Section 4.3.3 is used with each stage being done completely in parallel.

In Figure 6.2(a) we see that although the response time for all cases is increasing with increasing delays, it is much steeper for the linear and parallel versions than for the tree broadcast approach. One reason for this is because the tree broadcast method never has more readers for a particular memory module than the maximum number of children per node in the broadcast tree, so contention is kept low. A second reason is that only the first stage requires off-station accesses, since the second stage is designed to keep the accesses in-station (this is true only for the 16 processor case with a fan-out of 3). Therefore, the extra delays only effect the first stage of the algorithm.

In Figure 6.2(b) this difference is highlighted by showing the speedup of the two parallel versions over the base time of the serial version. Here we clearly see the benefit of the tree broadcast approach, both in its higher performance as shown by its higher speedup, and in its greater slope growth indicating that its benefits are increasing with increasing delays.

We expect similar results in larger systems without these extra delays, due to the increased contention and latency as the result of greater numbers of processors and larger distances.

We now look at how these delays affect the synthetic programs. Figure 6.3(a) shows the effects of the delays on the speedup of LU Decomposition. The graph shows speedup times on 16 processors for three different protocols:

rep-on-write. The Update protocol with replication only on write accesses.

replicate. The Validate protocol with full replication.

validate. The Validate protocol with no replication.

and two different placement strategies:

best. This attempts to place each row as close to the "owning" processor as possible, given the limitations of the protocol.

rr. This is a page-based round-robin distribution scheme that effectively randomizes the placement of data with respect to the "owning" processor.

As with all speedup results in this thesis, the curves represent speedup with respect to the best sequential version of the algorithm. The delays vary from 0 to 16 cycles. As the delays are increased, the rankings of the different algorithms and placement options remain constant, although the speedup is decreasing in all cases. Surprisingly, as the delays get bigger the discrepancy between the "round-robin" and "best" allocation schemes decreases. One would expect that the loss of locality due to round-robin placement would be more strongly felt for higher delays. Similarly, the benefits of replication are reduced for higher delays, although the extra locality would seem critical with higher off-station access times. No satisfactory explanation is currently available, although there may be some link between the small dip in performance of the replication schemes shown in Figure 6.2(b) at delays of around 8, and the drop off in performance shown here at similar delay values.

For Matrix Multiplication, Figure 6.2(b) presents similar speedup curves for the following protocols:

tree replicate. The data is replicated to each processor using the tree broadcast method.

replicate. The Validate protocol with full replication.

station replicate. The Validate protocol with station replication.

validate. The Validate protocol with no replication.

Round-robin allocation was used in all cases. Here the results are more in line with what is expected. The benefits of using tree replication become stronger for higher delays, and replication in general proves more beneficial. Additionally, with the higher remote access costs, station replication begins to show better performance with a delay of 8 cycles. In fact its performance drops off only slightly as the delays are increased, and hence begins to approach the performance of the full replication versions.

Finally, Figures 6.4(a)-(d) show the effect of delays on the Migrating Locality program. The same four protocols were used as in the original experiments:

No Coherence. No cache coherence is maintained, removing all the overheads of regions, plus any overheads that might be required to maintain coherence. No replication is performed.

Validate. The Validate protocol without replication.

Station Replication. The Validate protocol with replication at the station level.

Full Replication. The Validate protocol with full replication.

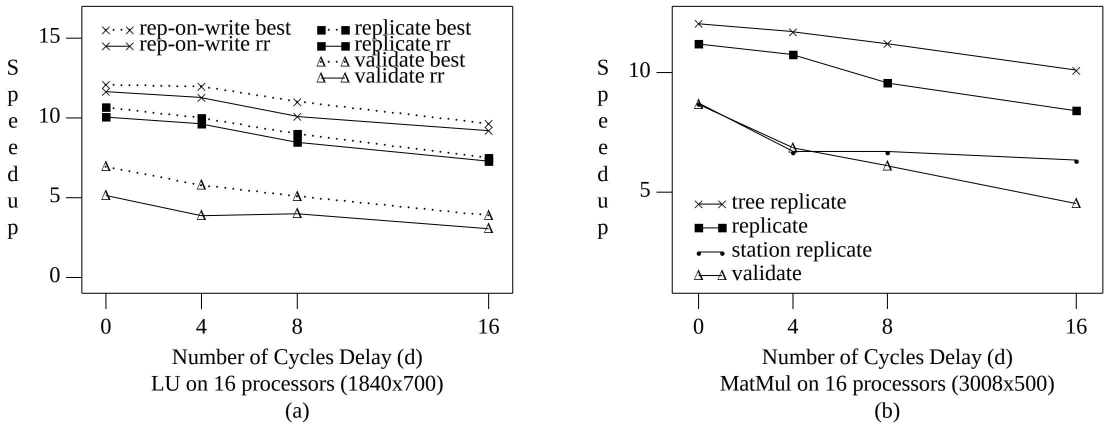


Figure 6.3: LU Decomposition and Matrix Multiplication performance on 16 processors with varying hardware delay settings.

Figure 6.4(a) reproduces the results with no delay with `psame` = 0.0 and `pwrite` = 1.0. Figure 6.4(b) shows the results for the same parameters but with a delay setting of 16. Here we see the increasing importance of replication, especially around the 0.01 mark. Both full replication and station replication are clearly superior to the non-replicating versions. However, as the locality decreases and the frequency with which data shifts from processor to processor increases, the cost of the replications begins to dominate. With the higher delays, the cost of cross-station replication is greatly increased, so that the performance of the replicating versions ends up significantly worse than the non-replicating protocols, when compared to the case with zero delay.

Figures 6.4(c) and (d) show similar results for `psame` = 0.0 and `pwrite` = 0.1 with zero delay, and a delay of 16 respectively. Here once again we see the increased benefit of replication when the data is relatively stable and there is some temporal locality, and the increased costs (due to the higher cost of replication with the extra delays) as the locality diminishes. We also find that the relative performance of the station replication protocol improves. Here, the in-station locality provides increased performance, while a reduced replication rate dampens the negative effect of non-locality when compared to the full replication case. Overall, station replication provides a fair compromise between the different options.

In general, there is some indication that, with larger systems, as reflected by the higher delays, locality will be more important, and the overhead of using shared-regions will be more easily amortized due to the savings from the reduced latency and contention. However, the results are not conclusive. In particular, although the Shared-Region approach improves locality and performance in the applications we examined, it seems clear from the decreasing performance with higher delays of all the applications, that more work is necessary, in the region implementation, in the region paradigm, and in the applications as well, if reasonable performance is to be achieved in larger systems.

One final note: because on-station accesses may be substantially faster (and less contended) than off-station accesses, the reduction in locality due to false sharing (when not using Shared-Regions) may be tolerable provided that processors sharing a page reside in the same station. When sharing processors are located in the same station, a single copy of the page can be kept close to all of them by simply placing the page somewhere in the

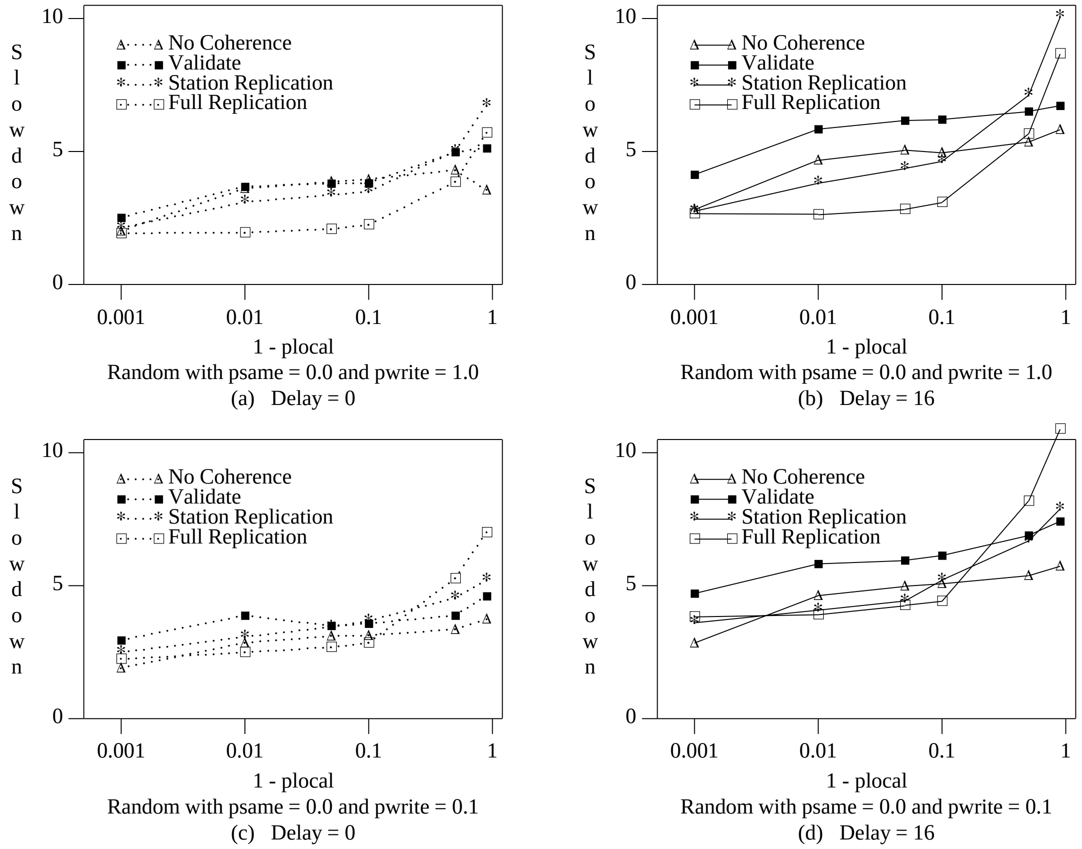


Figure 6.4: Performance of the Migrating Locality program with and without hardware delays of 16, shown as slowdown on 16 processors.

station. Under these circumstances, page-based approaches may prove sufficient (which is one of the motivations behind the clustering approach in Hector and its operating system Hurricane, as well as the station replication option in Shared-Regions). This reflects to a fair degree what one would expect to see in a larger system, where the loss of locality can be tolerated provided the data is still somewhere "nearby".

6.2 Modeling

In order to extend the results obtained so far (beyond what was possible with the delays in the previous section), we present a simple hardware and software model. It is designed to be accurate enough to capture the essential performance characteristics of common access patterns on a variety of machine configurations, while simple and flexible enough to lend insight into interactions responsible for the performance variations.

6.2.1 Modeling Techniques

The model is based on the results obtained in Chapter 5. It consists of simple cost equations for memory accesses expressed in terms of resource usages, simple queueing models to handle contention, and parameterized costs for common operations (such as region accesses) based on experimental results.

The cycle times for the various memory operations have already been presented in Section 5.1; however, in order to obtain an accurate model of the hardware for validation purposes, a more detailed understanding of the operation of the Hector memory subsystem is required.

- A cache line may be written back as the result of a read or write miss that must eject a dirty cache line, or as the result of an explicit cache flush or copyback operation.

- When a store misses in the cache, the block in main memory corresponding to the cache line must first be loaded into the cache, using a normal cache load operation. The write is then passed through the cache to main memory (this is required by the 88200 CMMU cache protocol). All subsequent writes to this cache line will not write to main memory.

- To understand contention, the resource requirements of the various requests must be understood.

  - Every network packet occupies the resource it passes through (station buses and ring segments) for exactly one cycle.
  - Every memory request occupies memory for 4 cycles for the first word, and 3 cycles for each subsequent word. In addition, protocol processing and arbitration keep memory busy for an additional 2 cycles for local accesses, and 5 cycles for off-board requests, totalling 15 and 18 cycles respectively.

- If a request arrives at a memory module and finds it busy, a negative acknowledgment is returned. The issuing processor then enters a retry loop where it delays for 13 cycles before the next retry, then 89 cycles, and then 5 cycles. Each stage is repeated several times. The whole cycle is repeated 16000 times before the request is considered to have failed.

- Packets arriving from the ring have priority on the bus. Within the bus, there is a descending order of priorities by cpu number. These can have a noticeable impact on the distribution of memory response times.
- Overall, except for the effect of priorities, the bus and ring network appear to have little impact on the performance of the system as compared to the effects of memory.

To see how this information is used, we present the formulae to estimate the cost of some simple memory operations, along with the actual times as measured on Hector.

We begin with some simple reads and writes to local, on-station, and off-station memories. Using Table 5.1 we can easily estimate the time required to read N bytes from local memory (assuming none of it is already in the cache) to be

$$\left(19\frac{N}{16}\right) \text{ cycles}$$

Similarly a read of N bytes from an off-station memory module in a system with 4 stations (the 16 processor prototype used for testing), requires

$$\left((29+4k)\frac{N}{16}\right) \text{ cycles}$$

where k is generally taken to be about 1.25. Writing N bytes (again assuming none of the data is present in the cache) to local memory requires:

$$\left(19\frac{N}{16}+10\frac{N}{16}\right) \text{ cycles}$$

where the first component is the cost of each cache load, and the second component is the cost of the first write that must always go to memory. When data is forced back to main memory either because of an implicit or explicit cache writeback, the cost of the writeback must be taken into consideration. So for example, assuming the cache size is C and the amount of data to write to local memory is qC, the total cost would be:

$$\left(19\frac{qC}{16}+10\frac{qC}{16}+19\frac{(\max(q,1)-1)C}{16}\right) \text{ cycles}$$

because all but C of the data will need to be flushed during the course of the sequence of writes.

Similar models are used for uncached reads and writes, as used by some of the copying operations.

Most of the operations we are interested in modeling consist of reads or writes of long vectors with a fixed amount of computation between each component accessed. In particular, all the synthetic programs are built from routines which read and write shared-regions with a delay between each word accessed, to simulate the computation performed on the data. Although adjustable by the programmer, the default delay is 2 cycles. Hence, any memory operations built up from these routines include the extra 2 cycles (or whatever the setting is) for each word. Where appropriate, all cost equations have this delay factored in.

To give an idea of the accuracy of these equations, Figure 6.5 presents some measured times and the corresponding estimated times. All reads and writes in the tests use the routines just described, and so the equations include the required 2 cycle delay per word.

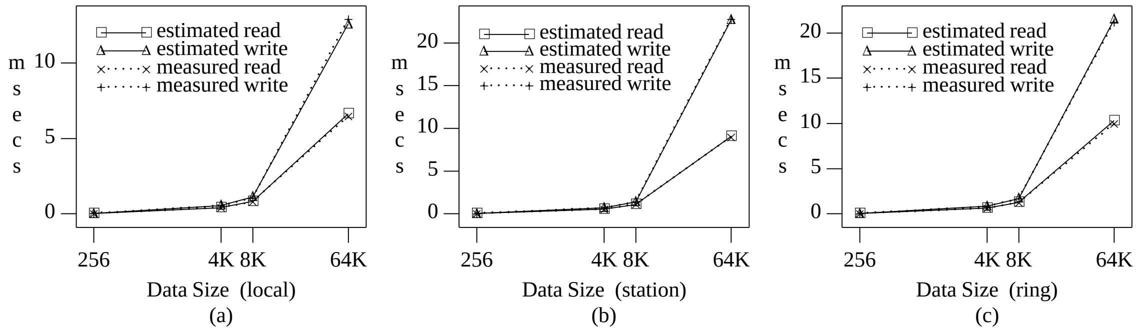


Figure 6.5: Comparison of estimated and measured read and write times.

From these results we can claim reasonable confidence for this part of the model.

Calculating latency in the face of contention is more complicated, however, both because of the difficulty in estimating the load at memory, and because of the difficulty in accounting for the effects of the priorities. As we saw in the previous chapter (in Figure 5.2) the effect of priorities and asymmetries in the system results in highly skewed access times under contention. Because of these difficulties, a number of simplifications have been made in building the model. First our model only attempts to handle two types of access patterns: hot spot accesses, with a fixed number of readers or writers accessing the same memory module for a fixed number of accesses; and random accesses distributed uniformly across all processors.

We model hotspots using a very simple single-server queueing model. The model consists of a single server, the memory module in question, plus a single class of customers consisting of the processors actively accessing the memory module. Since each processor can have only one outstanding memory request, the total number of customers is simply the number of processors P. The think time, Z, for these customers is the time spent in computation C plus the time spent in transit to memory T (this is included as part of think time because network contention is not being modeled, so what concerns us is simply how much time is spent away from the memory). The network transit time is based on the average distance between the processors and the memory module. The load at memory from each customer is the memory busy time B, for the particular request type (read, write, cached, or uncached).

It is reasonably safe to make these simplifying assumptions since memory is clearly the primary bottleneck. In particular, this model ignores the effects of network contention, priorities, and the lack of queues in the system.

Figure 6.6(a) shows the results from the model compared to the actual system response time. The experiment uses the standard routine (discussed earlier) to read 16K of data (through the cache) from a single memory module. With the default parameters, this results in a load of 18 cycles on memory (as described earlier) for each cache line, $4 \cdot 2 = 8$ cycles for computation, and approximately $29 - 18 = 11$ cycles for transit time, depending on how many off-station processors are involved (the 11 cycles is the difference between the total time for a cacheline fill and the memory busy time).

The measured response time presented is the average response time over all processors, since the times vary from one processor to the next. From the graph we see that, as the number of processors varies, the model tracks the experimental results fairly well, although it always under-predict the actual response time slightly.

The random access model is a simple extension of the hotspot model. We let the single

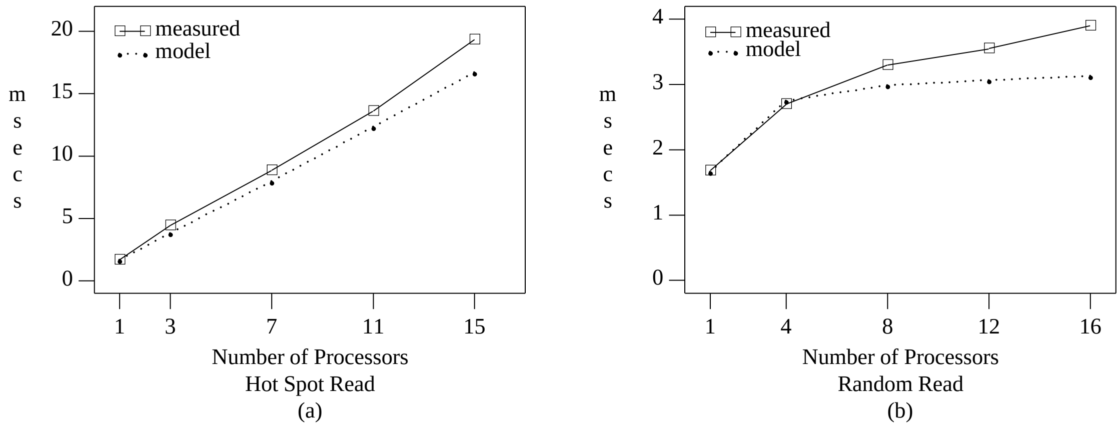


Figure 6.6: Comparison of model and measured results.

center be representative of an arbitrary memory module. Since all memory modules and all customers are statistically identical (assuming uniform distribution of memory requests for all processors), with P processors and P memory modules, we expect $1/P$ of the requests from each processor to go to any given memory module. We therefore use B/P, as the load at memory. Since we are using think time as the time spent away from this memory module, we need to include the time spent at other memory modules in addition to the time spent in transit and the time computing. But since all memory modules are the same, this is just the residence time, R^{mem}, of the memory module we are modeling. Hence, we can estimate the think time as:

$$Z = C + T + (P-1)R^{\mathrm{mem}}$$

By repeatably solving the model, and then using the computed residence time to recompute the think time, we converge on the solution.

Figure 6.6(b) compares the measured response time and the estimated response time for repeated random reads of 16 KB of data, with a varying numbers of processors. The figure shows a clear discrepancy between the model and the measured results. The model predicts that the response time should grow very slowly (it actually converges if the transit time is kept fixed), whereas the experiments show a consistent linear increase starting at 4 processors. It is difficult to say how much of this discrepancy is due to an incomplete model (network contention is not accounted for), and how much is due to the peculiarities of Hector's priority scheme and its lack of buffers. However, because of this discrepancy, we have chosen to use the model only when we are interested in lower bounds. For most cases, we use a linear regression based on the measured results, and use this to predict response times for random accesses.

From these simple models we can easily obtain other results, such as the time to perform memory copy operations. For example, the cost for P processors simultaneously to copy data from one processor (as happens in the LU Decomposition problem when using validation with replication), can be estimated by using the model for hot-spot reads, with an adjustment of the customers think time to account for time spent writing to local memory. This approach can also be used to estimate the cost of copying using the tree broadcast approach, by modeling the cost of each stage and then adding the stages together.

To model an application, it is necessary to take into account its access pattern, data distribution, and coherence protocol. Generally, the time for an application is computed by breaking the algorithm into roughly steady state phases, and using models similar to the ones above to estimate the response time of each phase. These are then summed together to give the total response time.

Since the synthetic programs are built on simple reads and writes of shared-regions, these operations make up the bulk of the execution time. We therefore frequently find the need for the following functions:

RU(N). The time to read N bytes, with a default delay of 2 cycles between each word access, assuming none of the data is presently in the cache.

RC(N). The same as RU, except that all of the data is assumed to be in the cache.

WU(N). The time to write N bytes, assuming none of it is in the cache. No writeback time is included.

WUB(N). The same as WU, but includes the cost to writeback the data.

WC(N). The time to write N bytes assuming it is all in the cache.

WCB(N). The same as WC, but includes the cost to writeback the data.

There are actually other parameters that control the cost of the individual memory operations based on the distances involved and the expected contention. These will be explained where necessary.

To demonstrate the application of the estimation techniques presented above, we show below the method used to estimate the response time for LU Decomposition and Matrix Multiplication for a sample of data distribution and coherence options.

Modeling LU Decomposition

The first thing required in modeling this algorithm is an understanding of the access patterns and caching effects. For LU, there are two phases to the algorithm: normalizing the pivot row, and using the pivot row to reduce the rest of the rows. It is expected that the accesses to the pivot row, apart from the initial access, will always result in cache hits. It is also expected that accesses to the rows being reduced will not be found in the cache (because the cache is not large enough to hold the entire local set). Although the LU algorithm as implemented attempts to reduce the impact of the pivot row normalization phase by overlapping it with other processors' reduction phase, processors still often wait for the pivot row to be normalized, and so create a hot spot by simultaneously accessing the row when it is completed. The first access to the pivot row (either to bring it into the cache or to make a local copy) is therefore treated as a hotspot access with $P-1$ readers (assuming P total processors).

To first verify that the model captures all the accesses, we first validate a uniprocessor version. For this case, we can model the execution time using the following equation:

$$\sum_{k=1}^{N}\left\{\mathrm{WUB}\left(R_p\frac{N-k}{N}\right)+\sum_{i=k}^{N}\left[\mathrm{RC}\left(R_p\frac{N-k}{N}\right)+\mathrm{RU}\left(R_r\frac{N-k}{N}\right)+\mathrm{WCB}\left(R_r\frac{N-k}{N}\right)\right]\right\}$$

where N is the number of rows, and R is the size of each row in bytes (the subscript indicates whether it is the pivot row, or one of the reducing rows). The term in the outer sum is the time to writeback the pivot row to memory. The first term in the inner sum is the time to read the pivot that is assumed to be in the cache (because it was just written, and this is the uniprocessor version). The second term is the time to read the next row to be reduced, and the third term is the time to write the row back to memory (assumed to be still in the cache). Computation time is included in the various delays between memory accesses and is therefore accounted for. Using this formula, the estimated execution time comes to 81111 milliseconds, while the measured execution time is 82060 milliseconds.

Next, we consider 8 processors with no replication, with the data distributed randomly. There are two major changes required to the above formula. First, the inner summation only counts every 8th row. Second, the time to remotely read the pivot row for the first time (after which it is cached) must be included. To compute the times for the various reads and writes we must now also take into account the remote accesses due to the random distribution, the contention also due to the random distribution, and the hotspot that occurs when all processors attempt to read the pivot row at approximately the same time. In each case we use the models described previously to adjust the cost of the individual memory access, and use these to parameterize the read and write cost functions. This gives an estimated execution time of 24007, compared to 25145 in the actual program.

Finally, we consider the case of 16 processors using the validate protocol with full replication. Now, we must take into account the fact that with replication, all non-pivot row accesses are local. We must also include the cost to perform the replications, in particular the replication of the pivot row. The pivot replication cost is estimated using a hotspot replication model as outlined earlier. This gives us an estimate of 8379 milliseconds, while the actual measured time is 8170.

Although these are only a few sample results, they indicate that the model is capturing the essential aspects of the algorithm, the region protocols, and the architecture.

Modeling Matrix Multiplication

Estimating the execution time of Matrix Multiplication is relatively straightforward, since the only difficult cost to model is the contended access to the B matrix; the accesses to A and C are almost always from the cache or local memory.

First, we begin with the estimated execution time on a single processor:

$$\sum_{k=1}^{N} \left\{ \mathrm{RU}(R_A) + \mathrm{RU}(R_C) + \sum_{i=1}^{N} [\mathrm{RC}(R_C) + \mathrm{RU}(R_B) + \mathrm{WC}(R_C)] + \mathrm{WCB}(R_C) \right\}$$

The first term inside the outer sum is the time to read a row of A (all other accesses to A are made from registers). The second term is the time to read a row of C into the cache. The first term in the inner sum is the time to read a row of C from the cache. The second term is the time to read a row of B, which is not expected to be in the cache, as the particular row of B that is read changes for each iteration. The final term in the inner sum is the cost to write C back to the cache. Finally, the final term in the outer sum is the cost to write C back to main memory at `WriteDone` time (this is actually a model of the validate algorithm on 1 processor). With this equation, the estimated single processor time is 121880 milliseconds, while the measured time is 127240 milliseconds.

Next we try 16 processors using round-robin distribution and no replication. The only change required is to limit the inner sum to N/16 rows (all rows are the same), and to adjust the cost of reading B to account for the expected contention due to random accesses from 16 processors. This gives us an estimated execution time of 14518 milliseconds, compared with 14690 milliseconds as measured.

We now consider 16 processors using the validate algorithm with full replication. Since replication happens entirely at the start of the computation (while computing the first row of C), we can separate this cost out from the rest, and compute the contribution of the rest using the original equation, with local data and no contention. To estimate the cost of the replication, we note that each processor is accessing the same rows of B at roughly the same time, so that the same hotspot copy model as used for LU is appropriate here as well. Using these assumptions, we get an estimated time of 11166 milliseconds, compared to a measured time of 11375 milliseconds.

Finally, using the tree distribution method, the model is similar to the previous one, except that we use a model of the time to execute the tree broadcast algorithm, rather than the fully parallel replication algorithm. This gives us an estimate of 10264 milliseconds compared to a measured time of 10575 milliseconds.

6.2.2 Application of Model

The results of the last section help us establish reasonable confidence in our modeling technique. The final step is therefore to use this approach to extend the measurement results of Chapter 5. In this section we estimate the performance of the various memory management techniques on larger problems and larger systems.

Larger Problems

As observed in Section 5.3.1, the speedups for many of the individual synthetic programs were far from optimal. It was explained that this was probably due to the small problem sizes used for experimentation. Here we have the opportunity to examine the performance of these applications with much larger data sets.

To begin with, we consider Matrix Multiplication for 1000×1000, 3000×3000, and 10000×10000 matrix (remember that for these programs, an $N \times N$ problem refers to a problem with N regions each of size N bytes). Figure 6.7(a) shows the estimated speedup on 16 processors using no replication, full replication, and tree-update replication. From the figure we can see that the version without replication does not improve with larger sizes, but remains flat. This is most likely due to memory saturation because of the larger number of remote accesses. In contrast, the two versions that employ replication steadily improve with increasing problem sizes, approaching the speedup limit of 16. Note however, that with some of these larger sizes, full replication would not be practical due to memory constraints.

Figure 6.7(b) shows the speedup on 16 processors for LU Decomposition, for problems of size 1000×1000, 3000×3000, and 10000×10000. Included are results for the Validate protocol without replication, for the Validate protocol with full replication, and the Update protocol with replicate-on-write. Random allocation was assumed in all cases. Overall, we see essentially the same results: the non-replication case reaches a plateau of about 6.1, while the other two continue to improve with increasing problem size, approaching the limit of 16. Note however, that with larger region sizes, false sharing would be less of an issue, and page-based systems would likely also show improved performance.

In both cases, the reason for the improved speedup is due to the larger number of regions. This is because the amount of reuse for each region grows in both cases with the number of regions. Hence the more regions, the more reuse, and the easier it is to amortize the cost of the replications.

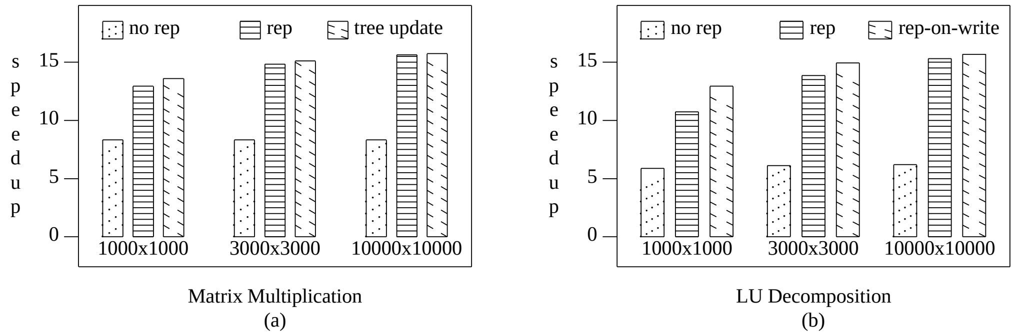


Figure 6.7: Speedups for different problem sizes for Matrix Multiplication and LU Decomposition on 16 processors.

It may also be worth noting that as the problem size increases, the performance benefit of the more intelligent replication strategies (avoiding replication in LU by using replication-on-write, or reducing its cost by using tree update for Matrix Multiplication) decreases. This is because when reuse is high enough, the cost of replication becomes insignificant, irrespective of which technique is used. However, as we shall see, for larger systems (and a fixed problem size), its importance returns.

Larger Systems

We now turn our attention to the impact of scaling the hardware. To keep things relatively simple, the same basic hardware configuration is maintained, with the exception of these changes:

- The number of processors is increased to 64, and then 128. For 64 processors we assume a configuration of 4 processors per station, 4 stations per ring, and 4 rings per global ring. For 128 processors we simply double the number of rings in the global ring. This increases remote access latencies by approximately 12 cycles for the 64 processor version, and 16 cycles for the 128 processors version. Network contention remains unmodeled.

- The impact of retries on in-station copybacks is removed, since this is an unnecessary artifact of the current system, and it tends to skew the results in favour of extra locality.

- We assume that it is possible for the system copy routines to read and write a whole cache line without actually going through the cache. This is a reasonable optimization to assume for future machines (a full asynchronous block-memory move operation is also possible).

We do not change the costs of the various operations, since we expect that, with the current technology trends, both processors and interconnect speeds will grow at roughly the same rate, and, although memory speed will grow more slowly, Hector's memory speed is presently quite slow for the processor being used. For simplicity we continue to present everything in microseconds or milliseconds using the 60ns cycle time of the original hardware.

As we saw in the last section, estimating contention in the face of random access distributions proved difficult. It remains unclear whether the model fails to capture an essential

	hotspot	random	random-light	tree	local
64	8910	1350	520	3330	210
128	17760	2510	550	4340	210

Table 6.2: Estimated memory access times (in microseconds) for reading 2048 bytes, on 64 and 128 processors.

aspect of the system, or simply that the experiment exposing exposes particular weakness in Hector. Because the experiments are the only reliable evidence we have, we continue to use them to estimate contention for random accesses for larger systems.

In order to complete the model, we also need an estimate of the region overheads. We assume that subsequent versions will solve many of the current problems, and allow the region descriptors to be better distributed, more efficiently implemented, and use contention-free locking or atomic-update schemes. We therefore conservatively fix the overhead at 3300 cycles (200 microseconds at Hector's processor speed) and scale it according to the estimated contention factor for random access patterns.

Before examining the applications, a brief look at the relationship between the different methods of accessing data is worthwhile. The scenario we have in mind is a situation where data is distributed randomly around the system, and at some point all processors will need all of the data. There are several different ways in which the data might be read, depending on the structure of the application. In all cases we use our standard method of reading memory, with 2 cycles delay between every word read. Table 6.2 shows the estimated latencies, based on our model, for reading the data for the following access patterns:

hotspot: The time to read the data assuming all other processors are trying to read the same data at the same time.

random: The time to read the data assuming all processors are actively reading random memory locations at the same time, using the estimation method described above.

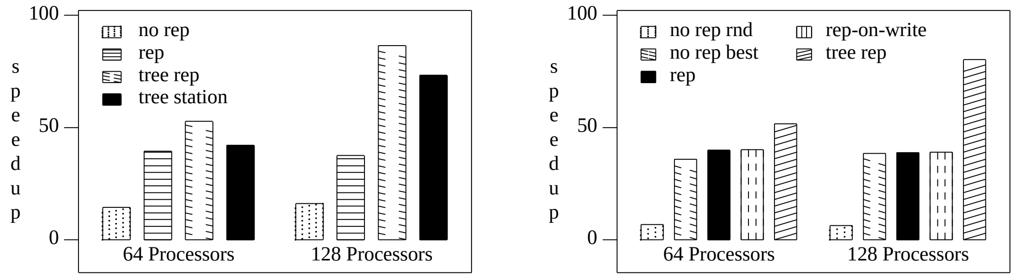
random-light: Similar to random, but using the more optimistic analytic model that predicts that memory contention will level off as the system size grows.

tree: The time to read the data assuming that it is first replicated to all processors using the tree replication scheme, and then read locally once. The time for the entire operation, including region overhead, is included.

local: The time to read the data locally, without contention.

From the table we see the tree version is actually quite competitive with the random version, despite the large amount of extra work involved, and the time appears to grow more slowly as well. The tree version clearly outperforms the hotspot case, indicating that even for a single read, tree replication may be profitable. For random reads, only 2 or 3 reuses of the data are required for the tree approach to prove beneficial. Even for random-light, the break even point for reuse is relatively small, and appears to be growing more slowly than the system size. For example, since the difference between a local access and random-light access is approximately 310 μs on 64 processors and 340 μs on 128 processors, it would take only $\lfloor 3330/310 \rfloor + 1 = 11$ total accesses on 64 processors, and $\lfloor 4340/340 \rfloor + 1 = 13$ total accesses on 128 processors, for the tree replication approach to be worthwhile.

We now look at some estimated speedup results for LU Decomposition and Matrix Multiplication. In both cases, the size is fixed at 3000×3000.



Matrix Multiplication
(a)

LU Decomposition
(b)

Figure 6.8: Speedups for different size problems for Matrix Multiplication and LU Decomposition.

In Figure 6.8 speedups on 64 and 128 processors is shown for both programs, for a number of different options.

For Matrix Multiplication, four versions were tried (all with random initial placement):

- no replication
- full replication
- full replication using tree broadcast
- station replication using tree broadcast

From Figure 6.8(a) we see once again that the version with no replication demonstrates only limited speedup and remains flat from 64 to 128 processors because of memory contention. For full replication, where we assume the pessimistic case of completely synchronous updates (and therefore hotspot contention), we see some improvement, but it too does not improve from 64 to 128 processors. With tree replication, however, we see strong benefits, as reasonable performance is obtained on 64 processors (a speedup of a little more than 50), and fair improvement with 128 processors (a speedup of a little less than 90). Most likely, better performance would be possible with a larger problem. We also include the time using the tree update scheme, but with only station replication. Although it is not as good as tree broadcast with full replication (since the benefits of the extra degree of locality with full replication far outweigh the cost of the extra stage in the tree replication algorithm) it is significantly better than full replication at 128 processors (one reason we need so many processors to see this is because of the mild NUMAness of the system).

For LU Decomposition, five different versions were used:

- no replication with random initial placement
- no replication with best placement (we assume half the data is local and the other is in the same station, as might occur with a page migration scheme)
- full replication
- replication on write only
- full replication, using tree replication for pivot row

The results are presented in Figure 6.8(b). Without locality, speedup is essentially frozen below 10. What we also see is that all but the last version perform poorly as well. This is partly because of the relatively small size of the problem (compared to the system size). However, looking at the performance of the last version, using tree-replication for the pivot row, we see that the main limiting factor in performance is the distribution of the pivot row. Using tree replication for the pivot row increases performance significantly, especially on 128 processors, despite the extra copies involved.

Summary

Using the model we developed, we found that for large problems, not only is replication and locality increasingly beneficial, but the performance achieved approaches the optimum.

For larger systems, the model supports once again the benefits of locality, but also shows that locality is not enough. In larger systems, how the data is delivered to its destination is as important as simply getting it there. The model indicates that tree based replication can provide significant performance benefits, even in cases where the cache is almost one hundred percent effective.

Chapter 7

Conclusions

7.1 Summary of Results

This thesis studied the performance of shared data replication in main memory and the accompanying coherence protocols in Non-Uniform Memory Access (NUMA) multiprocessors. Although the variable latency of memory was the initial motivation for studying this type of system, it was quickly determined that contention, primarily at memory, was the main culprit behind poor performance, and that replication would help primarily by distributing the load.

We introduced a method to support replication and coherence of main memory objects at arbitrary granularities. The technique relies on the programmer to delineate the boundaries of the objects, defining what are called Shared Regions, and to mark the synchronization points for these Shared Regions with special synchronization calls. Several protocols for replication and coherence, integrating cache and main memory, were presented, along with an implementation based on a 16-processor shared-memory multiprocessor.

Experimental results, using synthetic programs patterned after real applications, demonstrated the feasibility of the design and implementation, and showed significant improvements from the increased locality and reduced contention that was obtained. As is common with these types of studies, different replication and coherence protocol combinations worked best with different programs.

Further experimentation on a version of the hardware modified to simulate a larger machine verified the tendencies towards greater benefit of replication on larger systems. However, it also revealed greater sensitivity in the replication and coherence protocols to the access patterns of the applications; when the protocols matched the application access patterns, the system's performance was substantially improved over the alternatives, however when the protocols were ill-chosen, performance suffered terribly.

To further extend the results, a model of the system was developed and validated against the experimental system. The model was then used to extrapolate from the previous results to larger problems and larger systems. Once again, the benefits of locality, gained through replication, were observed. As expected, the amount of reuse of replicated data proved to be the key feature that determined the success (or failure) of the shared region approach. Additionally, significant improvements were seen from more intelligent data distribution schemes, in particular a tree-based replication algorithm, suggesting that such techniques will be of greater importance in future, larger systems.

Although the shared-region approach was found quite effective for experimentation purposes, it is not clear whether the interface and semantics are rich enough to support a wide variety of application domains.

Finally, despite some of the positive results from this thesis, the most natural conclusion to draw from this work is that we really do not understand well the issues surrounding system scaling, particularly in shared-memory multiprocessors, and only hands-on experience with actual large scale systems will prove or disprove any statements researchers care to make on this topic.

7.2 Future Work

For the topic of this thesis, as in any piece of research, there are many avenues left unexplored. Here we will list just a few.

From a performance analysis point of view, the results presented here should be considered only preliminary. Much work clearly remains to be done to better understand the effects of system size and other hardware characteristics on the final performance of applications (whether these applications use shared-regions or not). Models that consider contention in both the network and the memories, and handle a wider variety of access pattern distributions would also help in understanding what to expect as the system size grows.

From the experimentalist point of view, although the synthetic programs provide valuable and detailed insight into the access patterns of common algorithms, they cannot claim to be representative. Further application studies to determine the sharing characteristics of real applications at the natural granularity of the algorithm are essential to furthering this type of work, and determining where shared-regions might apply.

Perhaps more importantly, we saw in the thesis that the lack of communication between the shared-region layer and the operating system layer prevented many natural functions of the system from taking place, the most notable being the discarding of old pages. Since we do not expect the region-layer to become a complete memory manager for the entire system, closer cooperation is required if we are to have a fully functional environment.

Another potential avenue for investigation is determining how much benefit shared-region memory management could bring to a hardware cache coherent system. Although the synergistic relationship between the cache and main memory is an important component of this thesis, it is not essential to the performance results, and it might prove interesting to see whether similar benefits could be gained on a more modern cache-coherent machine.

Finally, many other methods of managing coherence are still open: languages and compilers provide some of the best opportunities for integrating coherence into the applications, and, although still in their infancy, they provide perhaps the best hope for the future in this field.

Bibliography

[AB86] J. Archibald and J. L. Baer. Cache coherence protocols: evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems*, 4(4):273–298, November 1986.

[ABHKK90] A. Agarwal, L. Beng-Hong, D. Kranz, and J Kubiatowicz. APRIL: A processor architecture for multiprocessing. In *Proc. of the 17th Annual Symposium on Computer Architecture*, pages 104–114, New York NY (USA), May 1990. IEEE.

[ACC+90] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera computer system. In *1990 International Conference on Supercomputing*, pages 1–6, 1990.

[AE91] Roger L. Adema and Carla Schlatter Ellis. Memory allocation constructs to complement numa memory management. In *Symposium on Parallel and Distributed Processing*, Dallas, TX, December 1991.

[AH90] S. V. Adve and M. D. Hill. Weak ordering - a new definition. In *Proc. of the 17th Annual Symposium on Computer Architecture*, pages 2–14, New York NY (USA), May 1990. IEEE.

[BCaCR92] Ray Bryant, Paul Carini, Hung ang Chang, and Bryan Rosenburg. Supporting structured shared virtual memory under Mac h. In *Usenix Mach Symposium.* Usenix, January 1992.

[BCZ90a] John K. Bennet, John B. Carter, and Willy Zwaenepoel. Adaptive software cache management for distributed shared memory architectures. In *SIGARCH*, pages 125–134, 1990.

[BCZ90b] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Munin: Distributed shared memory based on type–specific memory coherence. In *PPOPP 90*, March 1990.

[BFS89] William J. Bolosky, Robert P. Fitzgerald, and Michael L. Scott. “Simple But Effective Techniques for NUMA Memory Management”. *Operating Systems Review*, 23(5):19–31, 1989. Reprinted from *Proc. 12th ACM SOSP*, Dec. 1989, Litchfield Park, Arizona.

[BGW89] David Black, Anoop Gupta, and Wolf-Dietrich Weber. “Competitive Management of Distributed Shared Memory”. In *Spring COMPCON 89 Digest of Papers*, pages 184–190, 1989.

[BIFKR92] Henry Burkhardt III, Steven Frank, Bruce Knobe, and James Rothnie. Overview of the KSR1 computer system. Technical Report KSR-TR-9202001, Kendall Square Research, Boston (USA), Februrary 1992.

[BLL88] Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. PRESTO : A system for object-oriented parallel programming. *Software—Practice and Experience*, 18(8):713–732, August 1988.

[BR92] Bob Boothe and Abhiram Ranade. Multithreading: A new foundation for shared memory multiprocessors. In *Proc. of the 19th Annual International Symposium on Computer Architecture*, pages 214–223, Gold Coast, Australia, May 1992. IEEE.

[BS92] William J. Bolosky and Michael L. Scott. Evaluation of multiprocessor memory systems using off-line optimal behavior. *Journal of Parallel and Distributed Computing*, 15(4), August 1992.

[BSF+91] William J. Bolosky, Michael L. Scott, Robert P. Fitzgerald, Robert J. Fowler, and Alan L. Cox. "NUMA Policies and Their Relation to Memory Architecture". In *ASPLOS-IV Proceedings*, pages 212–221, Santa Clara, California, April 8–11 1991.

[BSU89] David R. Blythe, Michael Stumm, and Ron Unrau. Hurricane exception architecture – source specification. Technical report, University of Toronto, Toronto, Ontario, August 1989.

[CBZ91] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and performance of Munin. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 152–64. Association for Computing Machinery SIGOPS, October 1991.

[CF78] L. M. Censier and P. Feautrier. A new solution to coherence problems in multicache systems. *IEEE Transactions on Computers*, C-27(12):1112–1118, December 1978.

[CF89] Alan L. Cox and Robert J. Fowler. "The Implementation of a Coherent Memory Abstraction on a NUMA Multiprocessor: Experiences with PLATINUM". *Operating Systems Review*, 23(5):32–44, 1989. Reprinted from *Proc. 12th ACM SOSP*, Dec. 1989, Litchfield Park, Arizona.

[CKA91] D Chaiken, J. Kubiatowicz, and A. Agarwal. LimitLESS directories: A scalable cache coherence scheme. In *Proceedings of the 18th Int. Symp. on Computer Architecture*, pages 224–234. ACM SIGARCH, 1991.

[CKM88] Ron Cytron, Steve Karlovsky, and Kevin P. McAuliffe. Automatic management of programmable caches. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 229–238, August 1988.

[DL92] Czarek Dubnicki and Thomas LeBlanc. Adjustable block size coherent caches. In *Proc. of the 19th Annual International Symposium on Computer Architecture*, pages 170–180, Gold Coast, Australia, May 1992. IEEE.

[DS90] Michel Dubois and Christoph Scheurich. Memory access dependencies in shared-memory multiprocessors. *IEEE Transactions on Software Engineering*, 16(6):660–673, June 1990.

[EJ91] Susan J. Eggers and Tor E. Jeremiassen. Eliminating false sharing. *Proceedings of the 1991 International Conference on Parallel Processing*, I, Architecture:I–377–I–381, August 1991.

[EK88] S. J. Eggers and R. H. Katz. A characterization of sharing in parallel programs and its application to coherency protocol evaluation. In *Proc. 15th Annual International Symposium on Computer Architecture*, pages 373–383, Honolulu, HA, May 1988.

[EK89] Susan J. Eggers and Randy H. Katz. The effect of sharing on the cache and bus performance of parallel programs. In *Proc. 3rd ASPLOS*, pages 257–270, April 1989.

[FL92] Michael J. Feeley and Henry M. Levy. Distributed shared memory with versioned objects. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, October 1992.

[FSUZ88] George Feil, Michael Stumm, Ron Unrau, and Songian Zhou. Hurricane Operating System — A Preliminary Design. Technical report, University of Toronto, Toronto, Ontario, Canada, September 1988.

[GGH91] K. Gharachorloo, A. Gupta, and J. Hennessy. Performance evaluation of memory consistency models for shared memory multiprocessors. In *Proceedings of the 18th Int. Symp. on Computer Architecture*, pages 245–257. ACM SIGARCH, 1991.

[GGH92] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Hiding memory latency using dynamic scheduling in shared-memory multiprocessors. In *Proc. of the 19th Annual International Symposium on Computer Architecture*, pages 22–33, Gold Coast, Australia, May 1992. IEEE.

[GGK+83] Allan Gottlieb, Ralph Grishman, Clyde P. Kruskal, Kevin P. McAuliffe, Larry Rudolf, and Marc Snir. The NYU Ultracomputer – designing an MIMD shared memory parallel computer. *IEEE Transactions on Computers*, C-32(2):175–189, 1983.

[GGV90] Edward H. Gornish, Elana D. Granston, and Alexander V. Veidenbaum. Compiler-directed data prefetching in multiprocessors with memory hierarchies. In *International Conference on Supercomputing*, pages 354–368, 1990.

[GH89] R. Gupta and C. R. Hill. A scalable implementation of barrier synchronization using an adaptive combining tree. *International Journal of Parallel Programming*, 18(3):161–180, June 1989.

[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 *Proc. of the 17th Annual Symposium on Computer Architecture*, pages 15–26, New York, May 1990. IEEE.

[HKS91] Mark A. Holliday, Darrel S. Kindred, and Michael Stumm. On the scalability of ring-based, hierarchical shared memory multiprocessors. In preparation, 1991.

[HLRW92] Mark D. Hill, James R. Larus, Steven K. Reinhardt, and David A. Wood. Cooperative shared memory: Software and hardware for scalable. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, October 1992.

[JE91] Richard P. LaRowe Jr. and Carla Schlatter Ellis. Experimental comparison of memory management policies for NUMA multiprocessors. *ACM Transactions on Computer Systems*, 9(4):319–363, November 1991.

[JEK91] Richard P. LaRowe Jr, Carla Schlatter Ellis, and Laurence S. Kaplan. The robustness of NUMA memory management. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 137–51. Association for Computing Machinery SIGOPS, October 1991.

[JHE92] Richard P. LaRowe Jr., Mark A. Holliday, and Carla Schlatter Ellis. An analysis of dynamic page placement on a numa multiprocessor. In *Proceedings of the 1992 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 23–34, 1992.

[JLGS90] D. V. James, A. T. Laundrie, S. Gjessing, and G. S. Sohi. Distributed-direcotry scheme: Scalable coherent interface. *IEEE Computer*, 23(6):74–77, June 1990.

[Joh92] Kirk Johnson. The impact of communication locality on large-scale multiprocessor performance. In *Proc. of the 19th Annual International Symposium on Computer Architecture*, pages 392–402, Gold Coast, Australia, May 1992. IEEE.

[JWE91] Richard P. LaRowe Jr., J. T. Wilkes, and Carla Schlatter Ellis. Exploiting operating system support for dynamic page placement on NUMA shared memory multiprocessors. In *The Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Williamsburg, VA, April 1991.

[KCZ92] Pete Keleher, Alan L. Cox, and Willy Zwaenepoel. Lazy consistency for software distributed shared memory. In *Proc. of the 19th Annual International Symposium on Computer Architecture*, pages 13–21, Gold Coast, Australia, May 1992. IEEE.

[KEWP85] R. Katz, S. Eggers, D. A. Wood, and R. G. Perkins, C. Sheldon. Implementing a cache consistency protocol. In *Proc. of the 12th Annual IEEE Int. Symp. on Computer Architecture*, pages 276–283, Boston MA (USA), July 1985.

[Lam79] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9), 1979.

[LeB86] T. J. LeBlanc. Shared memory versus message-passing in a tightly-coupled multiprocessor: A case study. *Proceedings of the 1986 International Conference on Parallel Processing*, pages 463–466, August 1986.

[LH86] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *Proceedings of the 1986 5th Annual ACM Symposium on Principles of Distributed Computing*, pages 229–239, 1986.

[LLG+90] D. Lenoski, J. Laudon, K. Gharachorloo, A Gupta, and J. Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proc. of the 17th Annual Symposium on Computer Architecture*, pages 148–160, New York, May 1990. IEEE.

[LS89] Kai Li and Richard Schaefer. A hypercube shared virtual memory system. *Proceedings of the 1989 International Conference on Parallel Processing*, 1:125–131, 1989.

[LS90] Calvin Lin and Lawrence Snyder. A comparison of programming models for shared memory multiprocessors. *Proceedings of the 1990 International Conference on Parallel Processing*, pages II–163–II–170, August 1990.

[MG89] Margaret Martonosi and Anoop Gupta. Tradeoffs in message passing and shared memory implementations of a standard cell router. *Proceedings of the 1989 International Conference on Parallel Processing*, pages III–88–III–96, August 1989.

[MG91] Todd Mowry and Anoop Gupta. Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *Journal of parallel and Distributed Computing*, 12:87–106, 1991.

[MP85] M. Malkawi and J. Patel. Compiler-directed memory management policy for numerical programs. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, Orcas Island, Washington, December 1985.

[ON90] B. O'Krafka and A. R. Newton. An empirical evaluation of two memory-efficient directory methods. In *Proc. of the 17th Annual International Symposium on Computer Architecture*, pages 2–14, New York NY (USA), May 1990. IEEE.

[PL92] Thierry Priol and Zakaria Lahjomri. Trade-offs between shared virtual memory and message-passing on iPSC/2 hypercube. Technical Report IRISA-637, IRISA/INRIA, 1992.

[RAK89] Umakishore Ramachandran, Mustaque Ahamad, and M. Yousef A. Khalidi. Coherence of distributed shared memory: Unifying synchronization and data transfer. In *Proceedings of the 1989 International Conference on Parallel Processing*, pages 160–169, August 1989.

[SGZ92] H. S. Sandhu, B. Gamsa, and S. Zhou. Region-oriented memory management in shared-memory multiprocessors. Technical Report CSRI-269, Computer Systems Research Institute, University of Toronto, Toronto, Canada, April 1992.

[SH92] Jaswinder Pal Singh and John L. Hennessy. Data locality and memory system performance in the parallel simulation of ocean eddy currents. Technical report, Computer Systems Laboratory, Stanford University, Stanford, USA, 1992.

[SHG92] Jaswinder Pal Singh, John L. Hennessy, and Anoop Gupta. Implications of hierarchical n-body methods for multiprocessor architecture. Technical report, Computer Systems Laboratory, Stanford University, Stanford, USA, 1992.

[SJG92] Per Stenstrom, Truman Joe, and Anoop Gupta. Comparative performance evaluation of cache-coherence NUMA and COMA architectures. In *Proc. of the 19th Annual International Symposium on Computer Architecture*, pages 80–91, Gold Coast, Australia, May 1992. IEEE.

[Smi81] B. J. Smith. Architecture and applications of the HEP multiprocessor computer. In *Real-Time signal processing IV*, volume 298, pages 241–248, 1981.

[Ste90] P. Stenstrom. A survey of cache coherence schemes for multiprocessors. *IEEE Computer*, 23(6):12–24, June 1990.

[SZ90] Michael Stumm and Songnian Zhou. Algorithms Implementing Distributed Shared Memory. *Computer*, 23(5):54–64, May 1990.

[SZ91] Kenneth C. Sevcik and Songnian Zhou. Performance benefits and limitations of large NUMA multiprocessors. Technical Report CSRI-256, Computer Systems Research Institute, University of Toronto, Toronto, Canada, November 1991.

[Tan76] C. K. Tang. Cache system design in the tightly coupled multiprocessor sytsem. In *Proceedings AFIPS National Computer Conference*, pages 749–753, 1976.

[TLH90] Josep Torellas, Monica S. Lam, and John L. Hennessy. Shared data placement optimizations to reduce multiprocessor cache miss rates. In *1990 International Conference on Parallel Processing*, 1990.

[TS87] Charles P. Thacker and Lawrence C. Stewart. Firefly: a multiprocessor workstation. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 164–172, Palo Alto, California, October 1987.

[Unr92] Ron Unrau. *Scalable Memory Management for Scalable Multiprocessors.* PhD thesis, University of Toronto, 1992. In preparation.

[USK92] R. Unrau, M. Stumm, and O. Krieger. Hierarchical clustering: A structure for scalable multiprocessor operating system design. Technical Report CSRI-268, Computer Systems Research Institute, University of Toronto, Toronto, Canada, March 1992.

[VSWL90] Zvonko G. Vranesic, Michael Stumm, Ron White, and David Lewis. The Hector multiprocessor. *Computer*, 23(11), November 1990.

[WG89] Wolf-Dietrich Weber and Anoop Gupta. Analysis of cache invalidation patterns in multiprocessors. In *Proc. 3rd ASPLOS*, pages 243–256, April 1989.

[WH88] David H.D. Warren and Seif Haridi. The data diffusion machine - a shared virtual memory architecture for parallel execution of logic programs. In *Proceedings of the 1988 International Conference on Fifth Generation Computer Systems*, pages 943–952, Tokyo, Japan, December 1988.