Munin: Distributed Shared Memory Based on Type–Specific Memory Coherence

John K. Bennett[*]
John B. Carter[**]
Willy Zwaenepoel[**]

[*]Department of Electrical and Computer Engineering
[**]Department of Computer Science
Rice University
Houston, Texas

Abstract

We are developing Munin[†], a system that allows programs written for shared memory multiprocessors to be executed efficiently on distributed memory machines. Thus, Munin overcomes the architectural limitations of shared memory machines, while maintaining their advantages in terms of ease of programming. A unique characteristic of Munin is the mechanism by which the shared memory programming model is translated to the distributed memory hardware. This translation is performed by runtime software, with the aid of semantic hints provided by the user. Each shared data object is supported by a memory coherence mechanism appropriate to the manner in which the object is accessed. This paper focuses on Munin's memory coherence mechanisms, and compares our approach to previous work in this area.

This research was supported in part by the National Science Foundation under Grants CCR-8716914 and DCA-8619893 and by a National Science Foundation Fellowship.

[†] In Norse mythology, the ravens Munin (Memory) and Hugin (Thought) perched on Odin's shoulder, and each evening they flew across the world to bring Odin knowledge of man's memories and thoughts. Thus, the raven Munin can be considered to have been the first distributed shared memory mechanism.

1 Introduction

We are developing Munin, a system that allows programs written for shared memory multiprocessors to be executed efficiently on distributed memory machines. Shared memory programs are easier to develop than distributed memory (message passing) programs, because the programmer need not worry about the explicit movement of data. Distributed memory machines, however, scale better in terms of the number of processors that can be supported. Anticipated increases in processor speed relative to memory speed, and the advent of very fast networks, also argue in favor of distributed memory machines. Hence, our goal is to provide the best of both worlds: the relative ease of programming of the shared memory model and the scalability of a distributed memory machine. We approach this goal through a runtime system for a distributed memory machine that provides the illusion of shared memory to the programmer and to the compiler. In essence, the runtime system provides a single large virtual address space, distributed over many machines and memory modules, with overall memory coherence similar to that provided by hardware cache coherence mechanisms on shared memory machines. All data movement necessary to achieve memory coherence is performed automatically by the runtime system, and need not be visible at the application level. Munin programmers aid the system by providing semantic hints about the anticipated access pattern of the program's shared data objects.

This paper focuses on Munin's memory coherence mechanisms, and compares our approach to previous work in this area. What distinguishes Munin from previous distributed shared memory systems is the

means by which memory coherence is achieved. Instead of a single memory coherence mechanism for all shared data objects, Munin employs several different mechanisms, each appropriate for a different class of shared data object. We contend that this approach provides a an abstraction of shared memory on a distributed memory machine, that is more efficient than can be achieved with a static coherence method. This use of type-specific coherence mechanisms is the primary distinction between Munin and Ivy [12], Clouds [14], and Amber [4]. For this approach to work, a large percentage of shared data accesses must fall into a relatively small number of access type categories, that can be supported efficiently. A detailed study of the sharing behavior of parallel programs [2] supports this claim.

Many of the coherence mechanisms used in Munin are well known (e.g. replication, migration, invalidation, remote load/store), but we have also developed a powerful new mechanism that we call *delayed updates* that significantly reduces the amount of unnecessary message traffic and synchronization imposed by a distributed shared memory system. The basic premise behind the delayed update mechanism is that programmers use explicit synchronization to specify a partial ordering on access to shared data objects, which allows Munin to delay updating remote copies of an object when it is changed without affecting program correctness. Delaying updates allows the system to combine updates to the same object, and allows the data motion to be combined with the synchronization that prompted the update(s) to be propagated. Ideally, this would reduce the amount of network traffic to that achieved by a hand-coded message passing implementation.

Section 2 briefly summarizes the results of our study of sharing in parallel programs. Section 3 defines the notions of loose coherence and delayed updates, and presents the various type-specific coherence mechanisms. Section 4 describes the current status of the project and the anticipated directions for implementation. We compare Munin with related work in Section 5 and draw conclusiosn in Section 6.

2 Sharing in Parallel Programs

Type-specific memory coherence requires that there be a relatively small number of identifiable shared memory access patterns, for which corresponding memory coherence mechanisms can be developed, that characterize the majority of shared data objects. We studied six shared memory parallel programs written in the C++ language [15] using the Presto programming system [3] on the Sequent Symmetry shared memory multiprocessor [?]. We selected programs written specifically for a shared memory multiprocessor so that our results would not be influenced by the program being written with distribution in mind and would accurately reflect the memory access behavior that occurs when programmers do not expend special effort towards distributing the data across processors. The programs that we studied in detail were: Matrix multiply, Gaussian elimination, Fast Fourier Transform (FFT), Quicksort, Traveling salesman, and Life. Matrix multiply, Gaussian elimination and Fast Fourier Transform are well understood numeric problems that distribute the data to separate threads and access shared memory in predictable patterns. Quicksort is a representative sorting problem that uses divide-and-conquer to dynamically subdivide the problem. Traveling salesman is a representative graph problem that uses central work queues protected by locks to control access to problem data. Life is a representative "nearest-neighbors" problem in which data is shared amongst neighboring processes.

We have identified a limited variety of shared data objects: *Write-once*, *Write-many*, *Result*, *Migratory*, *Producer-Consumer*, *Private*, *Read-mostly*, *General Read-Write* and *Synchronization*. Intuitively, *Write-once* objects are read but never written after initialization. *Write-many* objects are frequently modified by multiple threads between synchronization points. *Producer-Consumer* objects are characteristically written (produced) by one thread and read (consumed) by a fixed set of other threads. *Migratory* objects are accessed in phases, where each phase corresponds to a run of accesses by a single thread. *Result* objects collect results. Once they are written, they are only read by a single thread that uses the results. *Private* objects are shared data objects that are only accessed by a single thread even though they are accessible to all threads. *Synchronization* objects, such as locks and monitors, are used by programmers to denote explicit inter-thread synchronization points. *Read-mostly* objects are read significantly more frequently than they are written. *General Read-Write* objects are those objects that are accessed in a way such that we could not characterize them as being in one of the previous categories. For our adaptive caching mechanism to work well, relatively few shared objects can fall into this class.

The general results of our analysis can be summarized as follows:

1. There are very few *General Read-Write* objects.
2. The notion of an object natural to the program-

mer often does not correspond to the appropriate granularity of data decomposition for parallelism. In particular, many objects that are write-shared are shared in such a way that different threads update independent portions of the object.

3. Parallel programs behave differently during different phases of their execution, and in particular exhibit significantly different access behavior during initialization than during the rest of their execution. The overwhelming majority of all accesses are reads, *except* during initialization.

4. The latency between accesses to synchronization objects (mainly locks) is significantly higher than the latency between accesses of other shared data items, even for programs with heavy use of synchronization.

These results strongly support our hypothesis that a distributed shared memory system employing a type-specific memory coherence scheme will outperform one that does not.

3 Memory Coherence

3.1 Overview

Munin treats the collection of all memories in the distributed system as a single address space, with coherence enforced by software. The virtual address space of each processor is partitioned into shared and private areas. The private area is local to each processor and contains non-shared data, the runtime structures used to manage memory coherence, and the system memory map used to record which segments of global shared memory are currently mapped into the local portion of shared memory. The system map may also contains hints about other processors' shared memory areas, but these hints may not always be reliable.

Munin views memory on each machine as a collection of disjoint segments. Munin servers on each machine interact with the applications program and the underlying distributed operating system to ensure that segments are correctly mapped into local memory when they are accessed. Munin performs fault handling in a manner analogous to page fault handling in a virtual memory system. When a thread accesses an object for which there is no local copy, a memory fault occurs. This causes Munin to suspend the faulting thread and invoke the associated server to handle the fault. The server checks what type of object the thread faulted on and invokes the appropriate fault handler. When Munin is unable to select a special memory coherence mechanism, a default mechanism similar to Ivy's is employed. In either case, Munin resumes the suspended thread after handling the fault.

Software coherence control exacts a certain cost, but it allows us to support more flexible ways of sharing than is possible in hardware. In particular, it allows us to support objects with coherence mechanisms tailored to their access characteristics, including using variable-sized cache items, and making dynamic decisions about coherence methods that adapt to the behavior of the program.

3.2 Loose Coherence and Delayed Updates

Delayed updates result from a relaxed definition of memory coherence:

> Memory is **loosely coherent** if the value returned by a read operation is the value written by an update operation to the same object that *could* have immediately preceded the read operation in some legal schedule of the threads in execution.

This contrasts with the more common definition used in Ivy [12] and Clouds [14]:

> Memory is **strictly coherent** if the value returned by a read operation is the value written by the most recent write operation to the same object.

Figure 1 illustrates the difference between these two definitions of coherence. `R1` through `R3` and `W0` through `W4` represent successive reads and writes, respectively, of the same object, and `A`, `B`, and `C` are threads attempting to access the object. Strict coherence requires that thread `C` at `R1` read the value written by thread `B` at `W2`, and that thread `C` at `R2` and `R3` read the value written by thread `B` at `W4`. Loose coherence, on the other hand, requires only that thread `C` at `R1` and `R2` read the value written at any of `W0` through `W4` such that the value read at `R2` does not logically precede the value read at `R1`, and that thread `C` at `R3` read either the value written by thread `A` at `W3` or the value written by thread `B` at `W4`. Essentially, strict coherence describes the implicit synchronization usually associated with message passing, and loose coherence describes the explicit synchronization normally associated with shared memory multiprocessors. Strict and loose coherence are closely related to the concepts of strong and weak ordering of events as described by Dubois et al. [8]. Programmers using

Munin specify only a partial order on the reads and writes of shared data objects.

As a result of their strict definition of coherence, Ivy and Clouds allow only one thread at a time to have write access to an object. This often leads to unnecessary memory coherence overhead when the programmer, knowing that the writes are independent, allows two threads to write to the same object without synchronization. In contrast, our loose definition of coherence allows updates to remote copies of a shared object to be delayed until it is convenient to perform them, or until the program's semantics requires them. For example, a synchronization event in a program requires that the delayed updates be propagated first. Delaying updates allows the system to combine updates to the same object, and allows the data motion to be combined with the synchronization that prompted the updates to be propagated. A simple example of this phenomenon occurs with matrix multiplication, where every thread computes a single element of the result matrix. With strict memory coherence, the result matrix (or cached portions thereof) travels between different machines. With delayed updates, the results are propagated once to their final destination.

We use a *delayed update queue* for each thread to maintain a list of the updates that have not yet been propagated. Whenever a thread modifies a shared object, we can delay sending out the update to remote copies of the object until remote threads could otherwise indirectly detect that the object has been modified before they receive its new value. Specifically, updates must be propagated in the order that they occur in the program execution, so that remote threads do not decide (erroneously) that an object has changed, and use the old value (believing it to be the new value). For example, if thread `A` updates object `X` and then updates object `Y`, the update to `X` must be propagated before the update to object `Y` because the program may make use of the fact that object `X` is modified before object `Y`. Analogously, the delayed update queue must be flushed whenever a thread synchronizes. The delayed update mechanism guarantees that updates eventually get propagated, because whenever a thread synchronizes (including during thread exit), the delayed update queue is flushed. Delaying updates allows the system to combine updates to the same object, thus reducing the network traffic. It also allows the data motion to be combined with the synchronization that prompted the updates to be propagated, which allows our system to benefit from the implicit synchronization provided by message passing.

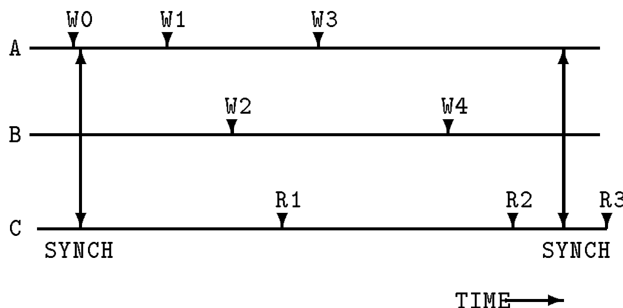

FIGURE 1: Strict and Loose Coherence

Compare how write sharing is handled by Ivy and Munin. Ivy enforcement of strict coherence only allows a thread to update a piece of data when no other thread is updating or reading it. In Munin, if two threads `A` and `B` attempt to update an object `X`, and one thread `C` attempts to read it such that the accesses to `X` are not synchronized, then `C`'s read can legally return the value written by `A`, the value written by `B`, or the original value of `X`. This is because there is no guarantee of the order in which the threads are scheduled, and our loose definition of coherence allows any of these values to be returned. In Ivy, if process `A` happens to write to `X` before `C` reads `X`, `C` is guaranteed to see the value written by `A`. This difference has a major effect on how the system can handle objects that are updated by multiple threads. In particular, Ivy introduces unnecessary synchronization that is avoided by Munin.

3.3 Type-specific Coherence Mechanisms

3.3.1 Write-once Objects

Write-once objects are written during initialization, but afterwards only read. Write-once data objects are frequently read concurrently by many threads. This can be supported efficiently via replication. When a thread tries to read an object for which there is no local copy, the local Munin server gets a copy of the object, without disturbing any copies of the object stored at other nodes. Replicating an object allows it to be accessed locally at each site. However, replication of large objects can lead to inefficient memory utilization, and can restrict the size of the problems that can be solved. It is also difficult to keep replicated objects coherent. When a write is done to an object, all of its copies must be updated or invali-

dated. Munin addresses these problems by allowing portions of large read-only objects to "page-out".

3.3.2 Write-many Objects

It is common for multiple threads to write to a single object concurrently. This can occur when the programmer knows that updates to different parts of the same object do not conflict. Programs written for shared memory multiprocessors must take into account arbitrary thread scheduling, and synchronize access to an object whenever different threads could simultaneously read from or write to it. Data access and synchronization are logically separate. Combining data motion and synchronization is one way that programs written on distributed systems achieve good performance. We believe that for distributed shared memory to be efficient, the underlying pattern of message passing used to support the illusion of shared memory for a particular program should closely resemble the pattern of message passing for an efficient message-passing implementation of the same program. We use *delayed updates* to combine data motion and synchronization in Munin.

3.3.3 Migratory Objects

Migratory objects [16] are accessed by a single processor at a time, as would be the case with an object accessed within a critical section of code. Migratory objects can be handled efficiently by integrating their movement with that of the lock associated with their critical section. If the lock queue is non-empty when a processor unlocks the critical section, then the object is "migrated", together with the lock itself, to the next thread in the lock queue. If the queue is empty, and assuming the system has no other knowledge about which thread will acquire the lock next, then the object is migrated with the lock to the next thread requesting the lock.

3.3.4 Producer-Consumer Objects

In some algorithms, processors only share data along object boundaries. For example, in a "nearest neighbors" algorithm, the new value for a particular matrix element is a function of the old values of neighboring elements. Thus, if the matrix is divided into a number submatrices, communication between processors only occurs at submatrix boundaries. "Wavefront" algorithms have similar data sharing characteristics. In all previous systems, efficiently handling this type of algorithm requires the programmer to substantially modify the algorithm to reduce the amount of synchronization required in passing data across boundaries.

If the system knows the producer-consumer relationship, it can perform *eager object movement*. Eager object movement is a mechanism that moves objects to the node at which they are going to be used before when they are required. In the nearest neighbors example, this involves propagating the boundary element updates as soon as they occur. In the best case, the new values are always available before they are needed, and threads never wait to receive the current values.

3.3.5 Read-mostly Objects

A Read-mostly is not accessed in a pattern that can be exploited with one of the above mechanisms, but is read far more often than it is written. They potentially can be handled relatively efficiently in a variety of ways, including via replication using delayed updates to keep the copies coherent or via remote load-store. The Munin prototype system uses remote load-store to handle read-mostly objects.

3.3.6 General Read-Write Objects

General Read-Write sharing is the general of arbitrary data sharing. It occurs when multiple threads are reading from and writing to the same data objects, and there is no particular pattern to the sharing that can be exploited. Munin handles general read-write objects using a mechanism based on the Berkeley Ownership cache consistency protocol [11]. By default, objects that are not recognized as some other specific type will be treated as general read-write. Our study showed that general read-write objects account for a very small percentage of all accesses to shared data.

3.3.7 Synchronization Objects

Synchronization objects are used to give threads exclusive access to other objects. When multiple threads access a single synchronization object, these accesses must be ordered while allowing threads to get fair access.

Munin supports distributed synchronization with *distributed locks*. More elaborate synchronization objects, such as monitors and atomic integers, are built on top of this. Our distributed locks employs *proxy objects* [1] to reduce network overhead. When a

thread wants to acquire or test a global lock, it performs the lock operation on a local proxy for the distributed lock. Proxy objects are maintained by a collection of distributed lock servers, one per processor. When a lock server detects an attempt to lock a local proxy object, it interacts with the other lock servers to acquire the global lock associated with the local proxy. When it has acquired the global lock, it allows the blocked thread to continue by releasing the local proxy lock to the thread. Unlocking is handled similarly.

Munin passes lock ownership amongst the distributed lock servers. Each lock has a queue associated with it that contains a list of the servers requiring access to the lock. This queue facilitates efficient exchange of ownership. Our distributed synchronization protocol also benefits from semantic information. For example, if the system can determine which thread is most likely to attempt to acquire a particular lock next, ownership of the lock can be migrated to the distributed lock server on the same processor as that thread. We plan to study several variants of this protocol to determine which is most efficient. The functional separation that the proxy mechanism provides facilitates this experimentation.

3.4 Dynamic System Decisions

Even objects with the same access type are not used in the same way by all programs. Munin must make dynamic decisions in handling objects to efficiently support a wide variety of programs. In this section we discuss two of these decisions, and their implications.

3.4.1 Replication vs. Remote Load Store

As we have discussed, replication is often useful in supporting read-shared objects. In some circumstances, replication may also be an appropriate mechanism for general read-write objects. In such cases, replication reduces read latency, but increases update (write) latency due to the added expense of updating or invalidating all remote copies of the object. Instead, when there is only a single remote copy of an object, it is relatively inexpensive to perform updates by performing a remote store to the single copy. However, this approach makes reads relatively expensive because every read requires a remote load. There are instances when each of these techniques is most appropriate. Since most programs perform many more reads than writes, replication will be the dominant mechanism for handling sharing. However, when an object is primarily written to, such as an object that collects results, maintaining a single copy is more efficient. Updates can be merged using our delayed write scheme to reduce the number of network packets required.

Previous systems have used only replication, but we believe that each approach is optimal under different circumstances. It is often possible to determine when replication or a single remote copy is preferable in a given situation based on program semantic information. Munin makes this decision on a per-object basis so the system can take advantage of any semantic knowledge that it obtains, either by inference, or directly from the user.

3.4.2 Invalidation vs Refresh

There are two fundamentally different ways to perform an update to a replicated object. One approach is to invalidate all remote copies of the object. If remote threads need to access the object after the update, they "page" it back in again. This approach is inefficient when a large number of threads frequently read the object. Another approach for handling remote updates is to refresh every remote copy of the object by propagating the new value of the object to each node maintaining a copy. This is more difficult than invalidation, because the new value rather than an invalidation message must be sent. Refresh using multicast reduces the amount of network traffic when many threads will request the new information eventually, but is not always a good idea. If the remote copies are not going to be used, or if several updates are going to occur between uses, invalidation iss better.

Previous distributed shared memory systems have assumed that only invalidation is appropriate, but again, each approach is preferable under different circumstances. Eggers and Katz [10] have shown that invalidation is preferable when the program exhibits a high degree of per-processor locality. Conversely, refresh is preferable when there is a high degree of fine-grained sharing.

4 Status and Directions

We are currently implementing Munin on an Ethernet network of SUN workstations. This implementation will allow us to assess the runtime costs of the delayed update queue and the other type-specific coherence mechanisms, as well as their benefits relative to standard static coherence mechanisms. We are using the V kernel [6] to provide high-speed communi-

cation between the different processors, and we have chosen to support the Presto [3] parallel programming environment to develop our shared memory parallel programs. Presto is a shared memory parallel programming environment that provides parallelism (lightweight processes) and synchronization (locks and Mesa-style monitors) for the object-oriented language C++ [15]. Programmers write their programs using a shared memory model, inserting declarations to provide object-specific information to the Munin runtime system. These declarations are processed by the compiler, and allow the runtime system to select the appropriate coherence mechanism for each object. Munin allows programs to be written in essentially the same way that they are written for shared memory multiprocessors. At the lower levels, our system uses only generic send and receive message passing primitives, and thus it can easily be ported to a variety of message passing architectures.

We chose Presto as our parallel programming environment for three reasons. First, the natural data encapsulation and inherent synchronization provided by object oriented programming languages makes them good candidates for distributed implementation. Data encapsulation makes it relatively easy for the system to determine the amount of memory that needs to be loaded or remotely updated. Second, it allows us to compare our system's performance with that of a true shared memory multiprocessor, as Presto currently runs on our Sequent Symmetry. Finally, we have experience using Presto and have a local community of users. We anticipate that this will make development and testing easier.

In the Munin prototype system, the server associated with each processor is a user-level process running in the same address space as the threads on that processor. This makes the servers easier to debug and modify, which serves our goal of making the prototype system expandable, flexible and adaptable. We will be able to add mechanisms should we discover additional typical memory access patterns. We will be able to profile the system to evaluate system performance, and determine the performance bottlenecks. Running at user-level, the Munin servers will have access to all operating systems facilities, such as the fileserver and display manager, which will facilitate gathering system performance information.

When Munin is fully operational we anticipate several related investigations. We currently rely on the programmer to provide all of the semantic information required by the Munin runtime system. In the future we plan to integrate a more powerful compiler into our system, in order to relieve the programmer of some of this burden. We plan to investigate the possibility of using the runtime system to determine the type of an object. Profiling information may enable Munin to "learn" about objects in the system. For example, the system might be able to detect that an object is being continuously updated by one thread and read by another. Upon noticing this, Munin could define the object as a producer-consumer shared object and treat it accordingly. We also plan to study what underlying system and/or hardware support would significantly improve Munin's performance. For example, a well designed network interface could reduce the overhead on each processor by performing some useful functions itself, such as reliable multicast and distributed locks. Performance on hardware with different performance characteristics, such as higher network bandwidth or increased processor speed, retains our active interest. Finally, the provision of fault tolerance and support for heterogeneity might be required in an operational system.

5 Related Work

The Ivy system [12] provides shared memory on a collection of Apollo workstations using a distributed memory manager. Ivy's *shared virtual memory* provides a virtual address space that is shared among all the processors in the system. Global virtual memory is divided into pages corresponding to physical pages. Each processor has a memory mapping manager that views local memory as a cache of the shared virtual address space. Ivy essentially uses a directory-based write-invalidate approach. Unlike Munin, Ivy enforces strict coherence and does not use any knowledge of access patterns of shared data (other than reads and writes). As a result, there are no special provisions for synchronization objects, and all sharing is on a per-page basis, entailing the possibility of significant amounts of false sharing. While less "transparent" than Ivy, because of the need for user annotations, we believe Munin provides a more efficient abstraction of distributed shared memory for a large variety of shared data types and the programs that use them.

The Clouds distributed operating system was extended to provide a form of shared memory [14]. The distributed shared memory controller allows objects to be mapped into the address space of any thread (process). Shared memory is divided into logical segments corresponding to Clouds objects, reducing the potential for false sharing. Objects may be locked to a particular processor while performing a series of operations on it, allowing the programmer to utilize

application specific knowledge to reduce the potential for "thrashing". Munin uses loose coherence to efficiently support multiple independent threads updating a single object, and also provides a general-purpose synchronization mechanism.

Amber [4] uses an object model as a basis for providing a shared address space spanning multiple processors. It enforces strict coherence by always migrating threads to the objects that they access. This works well for some programs, but often requires programmers to substantially modify their algorithms in order to reduce the overhead of migration and ensure that all of the threads do not migrate to the same host, thus eliminating all parallelism.

Cheriton et al. show that a software-controlled cache using a very large cache page size (an entire physical page) can provide the high performance needed to support fast multiprocessors [5]. This supports our claim that Munin, which is essentially a distributed caching mechanism provided in software, can efficiently provide a shared memory abstraction on a distributed system. The VMP scheme works well for many programs, but the large cache line size causes poor performance if there is a significant amount of fine-grained sharing. They did not investigate the possibility of having different cache coherence mechanisms available to handle different types of shared objects because their cache controller had no way of getting program-specific semantic information.

Our use of type-specific cache coherence mechanisms is further supported by earlier studies of the performance of snooping caches for parallel programs on shared memory multiprocessors. The designers of the Berkeley cache consistency protocol [11] found that their protocol can perform significantly better with a limited amount of information about how different data objects are accessed. Furthermore, Eggers and Katz [10] found that no single cache coherence protocol performed best for all types of shared data objects.

6 Conclusions

We have described the motivation and memory coherence mechanisms of Munin, a distributed shared memory system that selects a memory coherence mechanism for each data object based on the manner in which that object is expected to be accessed. We have argued that this provides distributed shared memory more efficiently than using a single memory coherence mechanism. We have selected these mechanisms based on the observed behavior of a a variety of parallel programs. We have found that a small number of mechanisms is sufficient to support most data sharing behavior that we have observed. We have described out *delayed update* mechanism, and showed that it allows data motion and synchronization to be combined, which reduces the amount of unnecessary network traffic needed to support distributed shared memory.

References

[1] John K. Bennett. The design and implementation of Distributed Smalltalk. In *Proceedings of the Second ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 318–330, October 1987.

[2] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Shared memory access characteristics. Technical Report COMP TR89-99, Rice University, September 1989. Submitted to 1990 International Symposium on Computer Architecture.

[3] 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.

[4] Jeffrey S. Chase, Franz G. Amador, Edward D. Lazowska, Henry M. Levy, and Richard J. Littlefield. The Amber system: Parallel programming on a network of multiprocessors. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 147–158, December 1989.

[5] David R. Cheriton, Gert A. Slavenburg, and Patrick D. Boyle. Software-controlled caches in the VMP multiprocessor. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, December 1986.

[6] David R. Cheriton and Willy Zwaenepoel. The distributed V kernel and its performance for diskless workstations. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, pages 129–140, October 1983.

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

[8] Michel Dubois, Christoph Scheurich, and Fayé A. Briggs. Synchronization, coherence,

and event ordering in multiprocessors. *IEEE Computer*, 21(2):9–21, February 1988.

[9] Susan J. Eggers and Randy H. Katz. A characterization of sharing in parallel programs and its application to coherency protocol evaluation. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 373–383, May 1988.

[10] Susan J. Eggers and Randy H. Katz. The effect of sharing on the cache and bus performance of parallel programs. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Systems*, pages 257–270, April 1989.

[11] R. Katz, S. Eggers, D. Wood, C.L. Perkins, and R. Sheldon. Implementing a cache consistency protocol. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 276–283, June 1985.

[12] Kai Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors*. PhD thesis, Yale University, September 1986.

[13] Susan Owicki and Anant Agarwal. Evaluating the performance of software cache coherence. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Systems*, pages 230–242, May 1989.

[14] Umakishore Ramachandran, Mustaque Ahamad, and M. Yousef A. Khalidi. Unifying synchronization and data transfer in maintaining coherence of distributed shared memory. Technical Report GIT-CS-88/23, Georgia Institute of Technology, June 1988.

[15] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1987.

[16] Wolf-Dietrich Weber and Anoop Gupta. Analysis of cache invalidation patterns in multiprocessors. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Systems*, pages 243–256, April 1989.