

# 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. Munin attempts to overcome the architectural limitations of shared memory machines, while maintaining their advantages in terms of ease of programming. Our system is unique in its use of loosely coherent memory, based on the partial order specified by a shared memory parallel program, and in its use of type-specific memory coherence. 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. These type-specific mechanisms are part of a runtime system that accepts hints from the user or the compiler to determine the coherence mechanism to be used for each object. This paper focuses on the design and use of Munin's memory coherence mechanisms, and compares our approach to previous work in this area.

---

<sup>†</sup> In Norse mythology, Munin (Memory) was one of two ravens perched on Odin's shoulder. Each day, Munin would fly across the world and bring back to Odin knowledge of man's memory. Thus, the raven Munin might be considered the world's first distributed shared memory mechanism.

## 1 Introduction

Munin<sup>†</sup> is 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.

What distinguishes Munin from previous distributed shared memory systems are 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 refer to this technique of providing multiple coherence mechanisms as type-specific memory coherence. Since coherence in distributed shared memory systems is provided in software, we believe that the added overhead and complexity associated with providing multiple coherence mechanisms will be offset by the increase in performance that such mechanisms will provide. This is the primary distinction between our work and that of Kai Li [11], Ramachandran et al. [13], and Chase et al. [4].

We have developed a powerful new coherence mechanism for Munin, called *delayed updates*, that allows multiple object updates to be combined into the same network packet, and to be propagated when convenient, such as when the program performs a synchronization operation. The other coherence mechanisms used by Munin are well known. These mechanisms include replication, migration, invalidation, and remote load/store. We use each of these mechanisms only for the particular types of shared data objects for which they are most appropriate.

We have based our design decisions on the results of a study of sharing and synchronization behavior in a variety of shared memory parallel programs, in which we observed that a large percentage of shared data accesses fall into a relatively small number of access type categories that can be supported efficiently [2].

This paper focuses on the design and use of Munin’s memory coherence mechanisms. Section 2 briefly summarizes the results of our study of sharing in parallel programs. Section 3 describes the design of Munin, including our use of loose coherence and delayed updates, and the particular methods by which coherence for each type of shared data object is handled. 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 conclusions 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 that characterize the majority of shared data objects, and for which corresponding memory coherence mechanisms can be developed. We have studied several shared memory parallel programs written in the C++ language [15] using the Presto programming system [3] on the Sequent Symmetry shared memory multiprocessor [12]. We se-

lected programs written specifically for a shared memory multiprocessor so that our results would not be influenced by the programs being written with distribution in mind. These programs more accurately reflect the memory access behavior that occurs when programmers do not expend special effort towards distributing the data across processors.

Our study of sharing and synchronization in parallel programs distinguishes itself from similar work [8, 14, 16] in that it studies sharing at the programming language level (and hence it is relatively architecture-independent), and in that our selection of parallel programs embodies a wider variation in programming and synchronization styles. The details of our study, and its quantitative results, are reported in [2].

The results of our study support our approach, in that we have identified a limited number of shared data object types: *Write-once*, *Private*, *Write-many*, *Result*, *Synchronization*, *Migratory*, *Producer-consumer*, *Read-mostly*, and *General read-write*. Intuitively, *Write-once* objects are read but never written after initialization. *Private* objects are only accessed by a single thread even though they are accessible to all threads. *Write-many* objects are frequently modified by multiple threads between synchronization points. *Result* objects collect results, and, once written, are only read by a single thread that uses the results. *Synchronization* objects, such as locks and monitors, are used by programmers to denote explicit inter-thread synchronization points. *Migratory* objects are accessed in phases, where each phase corresponds to a series of accesses by a single thread. *Producer-consumer* objects are characteristically written (produced) by one thread and read (consumed) by a fixed set of other threads. *Read-mostly* objects are read significantly more often than they are written. *General read-write* objects are those objects that are accessed in some other way that we could not characterize as being in one of the previous categories.

The main 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 programmer often does not correspond to the appropriate granularity of data decomposition for parallelism. In particular, the data indicate significant advantages for a memory coherence mechanism that is able to support both fine and coarse granularity over a mechanism that supports only one level of granularity.
3. Parallel programs behave differently during dif-

ferent phases of their execution, and in particular exhibit significantly different access behavior during initialization than during the rest of their execution. The vast majority of all accesses are reads, *except* during initialization.

4. The average period between accesses to synchronization objects (mainly locks) is significantly longer than the average period 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 systems using only a single mechanism.

## 3 The Munin System

### 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 contain 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 application 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 shared object the thread faulted on and invokes the appropriate fault handler. The suspended thread is then resumed after handling the fault.

Software coherence control exacts a certain cost, but it allows us to support more flexible ways of sharing than are 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 allows us

to make dynamic decisions about coherence methods that adapt to the behavior of the program.

### 3.2 Loose Coherence and Delayed Updates

A shared memory parallel program specifies only a partial order on the events within the program, both through explicit synchronization between threads of the program, and through implicit knowledge within one thread of the order of events in another thread. Munin exploits this partial order by using a relaxed definition of memory coherence:

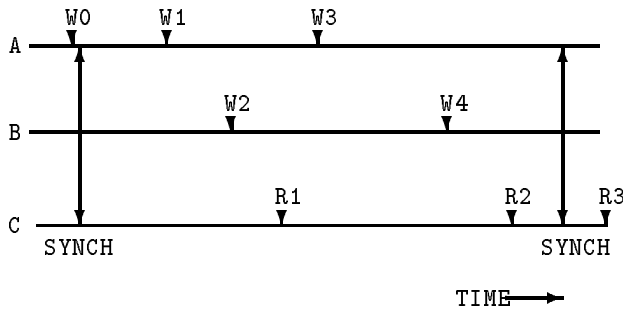
Memory is **loosely coherent** if the value returned by a read operation is the value written by an update operation to the same object that *could* have immediately preceded the read operation in some legal schedule of the threads in execution.

This definition contrasts with the more common definition of coherence used by Ivy [11] and Clouds [13]:

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 that 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**. Strict and loose coherence are closely related to the concepts of strong and weak ordering of events as described by Dubois et al. [7].

By enforcing only loose coherence, we avoid unnecessary synchronization that is not required by the program's semantics, and reduce the number of network packets needed for data motion and synchronization. When a thread modifies a shared object, we can delay sending the update to remote copies of the object until remote threads could otherwise



**Figure 1** Strict and Loose Coherence

indirectly detect that the object has been modified. Specifically, updates must be propagated in the order that they occur in the program execution, so that remote threads do not erroneously decide 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.

We use a *delayed update queue* for each thread to maintain the list of the updates that have not yet been propagated. Most outgoing updates of shared objects are done through the delayed update queue. Logically, whenever a replicated (partially or fully) data object is updated at a particular node, the delayed update queue on that node enqueues a record of the change. In practice, the number of such records actually made may be reduced by clever use of adequate paging hardware. Whenever a particular thread synchronizes, the delayed update queue is purged by sending out the new value of each object to the memory servers on the processors that have a copy. The remote memory servers acknowledge receipt of the data, possibly by responding that no copy resides there or that the copy there was invalidated. The local memory server then updates its directory of remote objects based on the acknowledgements. This allows the local server to dynamically adapt to changes in how the data is partitioned. The acknowledgements also allow the local memory server to determine where copies reside after only a single broadcast update, so broadcasts will be infrequent unless the data is repartitioned very frequently.

As a result of their strict definition of coherence, previous distributed shared memory systems 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.

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. 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. A simple example of how the use of delayed updates approximates the message passing behavior of a hand-coded distributed memory program can be seen 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.

### 3.3 Type-specific Coherence Mechanisms

Munin treats each shared data object as one of the nine general types that we observed in our study of shared memory programs, described in Section 2. Memory coherence for each type of object is supported by a fault handler specific to that object type.

#### 3.3.1 Write-once Objects

*Write-once* objects are written during initialization, but afterwards only read. These objects can be supported efficiently via replication. 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. Munin addresses this problem by allowing selected portions of large objects to be replicated.

#### 3.3.2 Private Objects

*Private* objects, once identified, need not be managed by the runtime system. Since all accesses to such objects are local, these accesses can proceed without

interference or delay. If an attempt is made to remotely access a private object, the object is brought back under the purview of the Munin runtime system.

### 3.3.3 Write-many Objects

*Write-many* objects are frequently modified by multiple threads between synchronization points. Our delayed update mechanism allows write-many objects to be handled efficiently.

### 3.3.4 Result Objects

*Result* objects are a restricted subset of write-many objects, and are also handled with the delayed update mechanism. Since result objects are not read until all of the data is collected, the system knows that updates to different parts of a result object will not conflict. This allows the delayed write mechanism to be utilized to maximum benefit.

### 3.3.5 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 the server 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.

### 3.3.6 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.

### 3.3.7 Producer-consumer Objects

*Producer-consumer* objects are written (produced) by one thread and read (consumed) by a fixed set of other threads. Producer-consumer sharing behavior is common in scientific programming. There are several types of algorithms in which processes only share data along observable boundaries. For example, in a “nearest neighbors” algorithm, calculating the new value for a particular matrix element involves computing 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 required the programmer to substantially modify the algorithm to reduce the amount of synchronization required in passing data across boundaries. In Munin, if the system knows the producer-consumer relationship, we perform *eager object movement*. Eager object movement moves objects to the node at which they are going to be used in advance of when they are required. In the nearest neighbors example, this involves propagating the boundary element updates to where they will be required 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.

### 3.3.8 Read-mostly Objects

*Read-mostly* objects are read far more often than they

are written. Munin replicates read-mostly objects and performs in-place update of these objects via broadcast. Because updates to read-mostly objects are infrequent, the use of broadcast is appropriate.

### 3.3.9 General Read-write Objects

*General read-write* objects are the general case of shared data objects. This 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 [10]. 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.

## 3.4 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 write-once and read-mostly 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. If instead, 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 update mechanism to reduce the number of network packets required.

Previous systems have used only replication, but we believe that each approach may be appropriate

under different circumstances. It is often possible to determine when replication or a single remote copy is preferable based on semantic information derived from the program. Munin makes this decision on a per-object basis so that 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 is 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 [9] 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. In our sharing study, delayed update was shown to outperform both write-invalidate and write-through. We found that write-many objects are read about half as often as they are written, and that there are large numbers of accesses between synchronization points. These large “no-synch” runs are similar to Eggers’ “write-runs” [8], and indicate that delayed updates, which

take advantage of our relaxed write semantics, will offer substantial performance improvement.

We are using the V kernel [6] to provide high-speed communication 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 type-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. Thus, 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 can therefore 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 locking. We are interested in the performance of Munin on hardware with different performance characteristics, such as higher network bandwidth or increased processor speed. Finally, the provision of fault tolerance and support for heterogeneity are important extensions that we wish to investigate.

## 5 Related Work

Previous research in distributed memory multiprocessing has lead to four basic approaches: *distributed programming*, where programmers explicitly specify all the communications and synchronization using low level message passing routines, *language based distributed computing*, where specially designed parallel programming languages try to make the underlying communications transparent to the user, *compiler based distributed computing*, where the compiler automates much of the required data motion, and finally, the approach to which Munin belongs, *shared memory distributed computing*. In this approach shared memory is simulated, generally using a distributed cache management scheme.

The Ivy system [11] 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 [13]. 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. Amber 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 line 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 an adaptive 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 obtaining 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 [10] found

that their protocol can perform significantly better with limited information about how different data objects are accessed. Furthermore, Eggers and Katz [9] found that no single cache coherence protocol performed best for all types of shared data objects.

## 6 Conclusions

This paper describes 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. Memory coherence mechanisms were selected based on the observed behavior of a variety of parallel programs. A small number of mechanisms was found to be sufficient to support the data sharing behavior that we have observed so far. A new coherence mechanism, called *delayed update*, was introduced and shown to allow data motion and synchronization to be combined, which reduces the amount of unnecessary network traffic needed to support distributed shared memory.

## Acknowledgements

David Johnson provided a number of suggestions for improving the structure of the paper. 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.

## 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] Michel Dubois, Christoph Scheurich, and Fayé A. Briggs. Synchronization, coherence, and event ordering in multiprocessors. *IEEE Computer*, 21(2):9–21, February 1988.
- [8] 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.
- [9] 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.
- [10] 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.
- [11] Kai Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors*. PhD thesis, Yale University, September 1986.
- [12] Tom Lovett and Shreekanth Thakkar. The Symmetry multiprocessor system. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 303–310, August 1988.
- [13] 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.
- [14] Richard L. Sites and Anant Agarwal. Multiprocessor cache analysis using ATUM. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 186–195, 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.