

# Adaptive Software Cache Management for Distributed Shared Memory Architectures

*John K. Bennett\**

*John B. Carter\*\**

*Willy Zwaenepoel\*\**

\*Department of Electrical and Computer Engineering

\*\*Department of Computer Science

Rice University

Houston, TX 77251-1892

## Abstract

An *adaptive* cache coherence mechanism exploits semantic information about the expected or observed access behavior of particular data objects. We contend that, in distributed shared memory systems, adaptive cache coherence mechanisms will outperform static cache coherence mechanisms. We have examined the sharing and synchronization behavior of a variety of shared memory parallel programs. We have found that the access patterns of a large percentage of shared data objects fall in a small number of categories for which efficient software coherence mechanisms exist. In addition, we have performed a simulation study that provides two examples of how an adaptive caching mechanism can take advantage of semantic information.

## 1 Introduction

We are developing Munin [4], a system that will allow programs written for shared memory multiprocessors to be executed efficiently on distributed memory machines. What distinguishes Munin from previous distributed shared memory systems [6, 12, 14] is the means by which memory coherence is achieved. Instead of a single memory coherence mechanism for all shared data objects, Munin will employ several different mechanisms, each appropriate for a different category of shared data object. We refer to this technique of providing multiple coherence mechanisms as *adaptive caching*. Adaptive caching maintains coherence based on the expected or observed access behavior of

each shared object and on the size of cached items. We contend that adaptive caching provides an efficient abstraction of shared memory on distributed memory hardware. Since coherence in distributed shared memory systems is provided in software, we expect the overhead of providing multiple coherence mechanisms to be offset by the increase in performance that such mechanisms will provide.

For adaptive caching to perform well, it must be possible to characterize a large percentage of all accesses to shared data objects by a small number of categories of access patterns for which efficient coherence mechanisms can be developed. In a previous paper [4], we have identified a number of categories, and described the design of efficient coherence mechanism for each. In this paper, we show that these categories capture the vast majority of the accesses to shared data objects in a number of shared memory parallel programs. We also show, through simulation, the potential for performance improvement of adaptive caching compared to static coherence mechanisms.

In Section 2 of this paper, we briefly reiterate the main results of our previous paper [4]. We describe the categories of access patterns, provide examples, and give a brief description of how each category can be handled efficiently. Section 3 describes the programs that we study in this paper, our technique for logging the accesses to shared memory by these programs, the method by which we analyze these logs to discover common access patterns, and the results of our logging study. Section 4 describes a simulation study that provides two examples of how an adaptive caching mechanism can take advantage of semantic information. We discuss previous work in this area in Section 5. Finally, we draw conclusions in Section 6.

---

This work was supported in part by the National Science Foundation under Grants CDA-8619893 and CCR-8716914.

## 2 Categories of Sharing

### 2.1 Intuitive Definitions

We have identified the following categories of shared data objects: *Write-once*, *Write-many*, *Producer-Consumer*, *Private*, *Migratory*, *Result*, *Read-mostly*, and *Synchronization*. We classify all shared data objects that do not fall into one of these categories as *General Read-Write*.

*Write-once* objects are read-only after initialization. *Write-many* objects frequently are modified by several threads between synchronization points. For example, in Quicksort, multiple threads concurrently modify independent portions of the array being sorted. *Producer-Consumer* objects are written (produced) by one thread and read (consumed) by a fixed set of other threads. *Private* objects, though declared to be shared data objects, are only accessed by a single thread. Many parallel scientific programs exhibit “nearest neighbors” or “wavefront” communication whereby the only communication is the exchange of boundary elements between threads working on adjacent sub-arrays. The boundary elements are *Producer-Consumer* and the interior elements are *Private*. *Migratory* objects are accessed in phases, where each phase corresponds to a series of accesses by a single thread. Shared objects protected by locks often exhibit this property. *Result* objects collect results. Once written, they are only read by a single thread that uses the results. *Read-mostly* objects are read significantly more often than they are written. *Synchronization* objects, such as locks and monitors, are used by programmers to force explicit inter-thread synchronization points. Synchronization events include attempting to acquire a lock, acquiring a lock, and releasing a lock. The remaining objects, which we cannot characterize by any of the preceding classes, are called *General Read-Write*. The categories define a hierarchy of types of shared data objects. When we identify an object’s sharing category, we use the most specific category possible under the following order (from most specific to least specific): *Synchronization*, *Private*, *Write-once*, *Result*, *Producer-Consumer*, *Migratory*, *Write-many*, *Read-mostly*, and *General Read-Write*.

### 2.2 Coherence Mechanisms

We have developed memory coherence techniques that can efficiently support these categories of shared data objects. A brief description of these mechanisms may provide insight into why these particular categories are chosen. A separate paper describes our

complete set of coherence mechanisms in more detail [4].

*Write-Many objects* appear in many parallel programs wherein several threads simultaneously access and modify a single shared data object between explicit synchronization points in the program. If the programmer knows that individual threads access independent portions of the data, and the order in which individual threads are scheduled is unimportant, the program can tolerate a controlled amount of inconsistency between cached portions of the data. The programmer uses explicit synchronization (such as a lock or monitor) to denote the points in the program execution at which such inconsistencies are not tolerable. We refer to this controlled inconsistency as *loose coherence* [4], as contrasted with *strict coherence*, in which no inconsistency is allowed. Strict and loose coherence are closely related to the concepts of *strong* and *weak ordering* of events as described by Dubois, et al [7]. Strong and weak ordering define a relation on the ordering of events (such as accesses to shared memory) in a system, while strict and loose coherence are operational definitions of the coherence guarantees that a system provides. Maintaining strict coherence unnecessarily is inefficient and introduces *false sharing*. The effects of false sharing can often be reduced by algorithm restructuring or careful memory allocation, but these efforts impose significant additional work on the programmer or compiler, are not possible for all algorithms, and are architecture dependent.

*Delayed updates*, based on loose coherence, allow *Write-many* objects to be handled efficiently. When a thread modifies a *Write-many* object, we delay sending the update to remote copies of the object until remote threads could otherwise indirectly detect that the object has been modified. In this manner, 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.

If the system knows that an object is shared in *Producer-Consumer* fashion, it can 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. In the best case, the new values are always available before they are needed, and threads never wait to receive the current values.

We propose to handle *Synchronization* objects with distributed locks. More elaborate synchronization objects, such as monitors and atomic integers,

can be built on top of this. When a thread wants to acquire or test a global lock, it performs the lock operation on a local proxy for the distributed lock, and the local lock server arbitrates with the remote lock servers to perform the lock operation. Each lock has a queue associated with it that contains a list of the servers that need the lock. This queue facilitates efficient exchange of lock ownership. This mechanism is similar to that proposed by Goodman, et al [9].

Several categories of shared data objects can be handled in a straightforward fashion. *Private* objects are only accessed by one thread, so keeping them coherent is trivial. Replication is used for *Write-once* objects. *Read-mostly* objects are also candidates for replication since reads predominate writes. A *Migratory* object can be handled efficiently by migrating a single copy of the object among the processors that access it. *Result* objects are handled by maintaining a single copy and propagating updates to this copy. Finally, *General Read-Write* objects are handled by a standard coherence mechanism.

## 3 Logging Study

### 3.1 Programs

We have studied six shared memory parallel programs written in C++ [17] using the Presto programming system [5] on the Sequent Symmetry shared memory multiprocessor [13]. The selected programs are written specifically for a shared memory multiprocessor so that our results are not influenced by the program being written with distribution in mind and accurately reflect the memory access behavior that occurs when programmers do not expend special effort towards distributing the data across processors. Presto programs are divided into an initialization phase, during which the program is single-threaded, and a computation phase.

The six programs are: Matrix multiply, Gaussian elimination, Fast Fourier Transform (FFT), Quicksort, Traveling salesman problem (TSP), and Life. Matrix multiply, Gaussian elimination, and Fast Fourier Transform are numeric problems that distribute the data to separate threads and access shared memory in predictable patterns. Quicksort uses divide-and-conquer to dynamically subdivide the problem. Traveling salesman uses central work queues protected by locks to control access to problem data. Life is a “nearest-neighbors” problem in which data is shared only by neighboring processes.

As a measure of the “quality” of these parallel programs, Figure 3.1 shows a speedup plot for each of the six programs. The programs exhibit nearly

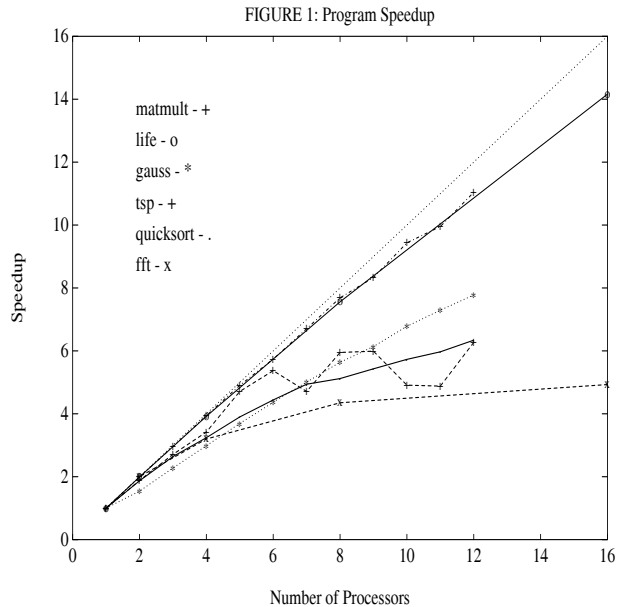
linear speedup for small numbers of processors. The decrease in speedup seen for larger numbers of processors is due primarily to the unavailability of processors and the effects of bus contention.

### 3.2 Logging Technique

We collect logging information for a program by modifying the source and the run-time system to record all accesses to shared memory (13 microseconds to record each access). The program modifications are currently done by hand. A call to a logging object is added to the program source after every statement that accesses shared memory. The Presto run-time system is modified so that thread creations and destructions are recorded, as are all synchronization events. The end of each program’s initialization phase is logged as a special event so that our analysis tool can differentiate between the initialization and the computation phase.

A program executed with logging enabled generates a series of log files, one per processor. Each log entry contains an *Object ID*, a *Thread ID*, the *Type of Access*, and the *Time of Access*. Examples of *Type of Access* include creation, read, write, and lock and monitor accesses of various types. *Time of Access* is the absolute time of the access, read from a hardware microsecond clock, so the per-processor logs can be merged to form a single global log.

We can specify the granularity with which to log accesses to objects. The two supported granularities are *object* and *element*. At object granularity, an



access to any part of an object is logged as an access to the entire object. At element granularity, an access to a part of an object is logged as an access to that specific part of the object. For example, the log entry for a read of an element of a matrix object indicates only that the matrix was read at object granularity, but indicates the specific element that was read at element granularity.

Our study of sharing in parallel programs distinguishes itself from similar work [8, 15, 18] in that it studies sharing at the programming language level, and hence is relatively architecture-independent, and in that our selection of parallel programs embodies a wider variation in programming and synchronization styles.

An important difference between our approach and previous methods [1, 16] is that we only log accesses to shared memory, not all accesses to memory. Non-shared memory, such as program code and local variables, generally does not require special handling in a distributed shared memory system. A useful side effect of logging only accesses to shared memory is that the log files are much more compact. This allows us to log the shared memory accesses of relatively long-running programs in their entirety, which is important because the access patterns during initialization are significantly different from those during computation.

Logging in software during program execution combines many of the benefits of software simulation [16] and built-in tracing mechanisms [1], without some of the problems associated with these techniques. As with software simulation, with software logging it is easy to change the information that is collected during a particular run of the program. For example, if only the accesses to a particular object are of interest, such as the accesses to the lock protecting a central queue, only the logging associated with that object need be enabled. On the other hand, software-based logging does not slow down program execution to the extent that software simulation of the program and architecture does. Unlike with address tracing techniques, it is possible to collect higher-order information about particular accesses. For example, we can log an attempt to acquire a monitor, successful acquisition of the monitor, or sleeping on a monitor condition variable. This information is not easily recreated from a standard address trace.

The flexibility, power and low overhead of our system does not come without cost. Only accesses to shared memory performed by the applications program and run-time system are collected, so our system suffers from what Agarwal refers to as *omission distortion* [1], the inability of a system to record the

complete address stream of a running program. The omission distortion is not significant in this study because we are not trying to determine how any particular cache coherence mechanism will perform, but rather are attempting to characterize patterns of sharing that are common in parallel applications programs. Also, because only accesses to shared memory are collected, our logs may experience *temporal distortion* in the sense that periods with frequent accesses to shared memory will be slowed down to a greater extent than periods when accesses to shared memory are infrequent. The temporal distortion is limited by synchronization events, which constrain the relative ordering of events.

### 3.3 Analysis

We now formalize the intuitive definitions of the different categories of shared data objects given in Section 2 for the purpose of the log analysis.

Objects that are accessed by multiple threads between consecutive synchronization events sufficiently often (default: during 50% of the inter-synchronization periods) are categorized as *Write-Many*. Whenever a thread synchronizes, the analysis program examines each object that the thread has modified since it last synchronized to determine if another thread has accessed the same object during that period.

A producer-consumer phase for a particular object *Obj* is characterized by the following sequence of events. Thread *A* writes *Obj* and synchronizes (thread *A* may access *Obj* additional times). Then some other threads read *Obj* before *A* writes it again. The analysis program counts the number of accesses to an object that occur in producer-consumer phases, and if this number exceeds a specified percentage of all accesses to the object (default: 75%), then the object is declared to be *Producer-Consumer*.

*Migratory* objects are accessed in long runs. A *run* or *write-run* [8] is a sequence of accesses to a single object by a single thread. For the purposes of our analysis, the minimum length of a long run is variable (default: 8). An object is declared to be migratory if the percentage of all accesses to it that are contained in long runs exceeds a threshold (default: 85%).

*Read-Mostly* objects are primarily read (default: 75%). *Write-once*, *Result*, *Private*, and *Synchronization* objects can be easily identified using their intuitive definitions (see Section 2.1).

We have developed a tool to analyze the shared memory access logs in the manner just described. The tool detects common access patterns and identifies objects according to the characteristic kind of sharing

that they exhibit. It also collects statistics specific to each category of shared data object, such as the average number of consumers for a *Producer-Consumer* object, the average run length of a *Migratory* object, and the average number of accesses to a *Write-Many* object between consecutive synchronization events. We have experimented with different values for the various thresholds, and the results do not appear to be very sensitive to variations in these thresholds.

### 3.4 Results

The results of our analysis are summarized in Tables 1 through 7. Odd-numbered tables present results from analysis runs where data was logged “by object.” Even-numbered runs present results where data was logged “by element.”

Tables 1 and 2 give the relative frequency of each type of shared access for each of the programs studied. These tables indicate the potential advantages of a memory coherence mechanism that is able to support both object and element granularity over a mechanism that supports only one level of granularity. With object-level logging, *Write-Many* accesses dominate other forms of shared data access (81.4, 100, 99.1, 99.2, and 47.3 percent), except for Matrix Multiply (only 2.8 percent). The other sharing category into which a large portion of the accesses fall at object-level granularity is *Write-Once* (18.6 percent in FFT, 97.2 percent in Matrix Multiply, and 23.9 percent in TSP). Parallel programs in which the granularity of sharing is fine tend to have their underlying fine grained behavior masked when the logging is performed on a per-object basis. With per-element logging, Matrix Multiply retains its *Write-Once* behavior, indicating that these are the inherent results. However, the access behavior of other programs are considerably different when examined per element. The best example of this unmasking is the Life program, where 82.4 percent of the shared data is in fact *Private* (the interior elements of the board) and 16.8 percent is *Producer-Consumer* (the edge elements).

The results in Tables 1 and 2 can be related to user-level objects in the programs as follows:

- The input arrays in Matrix Multiply exhibit *Write-Once* behavior, and references to these arrays dominate all other forms of access regardless of the granularity of logging. Accesses to the output matrix show up as *Result* when logged by element.
- Edge elements in the Life program exhibit *Producer-Consumer* behavior when shared accesses are logged by element. Internal elements are *Private*.

Type	FFT	Qsort	Gauss	Life	Mult	TSP
Private						
RdMostly						
WriteOnce	18.6				97.2	23.9
Result						
Prod/Cons						
Migratory						15.0
WriteMany	81.4	100	99.1	99.2	2.8	47.3
Synch			.9	.8		12.4
GeneralRW						1.4

TABLE 1: Percent of Shared Access (By Object)

Type	FFT	Qsort	Gauss	Life	Mult	TSP
Private	.7	2.8	29.5	82.4	1.6	
RdMostly			30.0			13.2
WriteOnce	17.9		1.7		95.8	25.5
Result			3.7		2.6	
Prod/Cons			10.0	16.8		
Migratory	73.4	.8	2.6			40.8
WriteMany	8.0	96.4	21.6			4.4
Synch			.9	.8		12.0
GeneralRW						4.1

TABLE 2: Percent of Shared Access (By Element)

- In TSP, the input array containing the path weights is *Write-Once*. At object granularity, the work queue is accessed in a migratory fashion, since it is a single object protected by a lock. The different partially computed tours are treated as one single object, and thus accesses to them are categorized as *Write-Many*. At element granularity, the partially computed tours are treated as independent objects and also exhibit *Migratory* behavior.
- At object granularity, the input coefficient array in Gaussian Elimination exhibits *Write-Many* behavior. At element granularity, access behavior to this array is less well-defined, indicating the different manner in which row, column, and pivot elements are accessed.
- The array being sorted in Quicksort exhibits *Write-Many* sharing at both object and element granularity. This is because of how Quicksort is implemented. The programmer knows that different threads access independent parts of the array, so accesses to the array are not synchronized.
- The input sample array in FFT exhibits *Write-Many* sharing behavior at object granularity. At

element granularity, this array exhibits *Migratory* behavior because elements are passed between workers in phases. The  $\omega$  array, an array of numeric coefficients that is initialized at the start of the algorithm and used extensively thereafter, is *Write-Once*. A temporary array, which is used to re-order the input array (in parallel) so that the elements required by each worker thread are contiguous, is *Write-Many*.

These observations correspond closely to our expectations, based on an informal understanding of how the programs access shared memory.

Tables 3 and 4 illustrate the dramatic differences between element and object granularity. These tables record the average size of the data element being accessed for each type of access for each of the programs. The objects at object granularity are generally arrays or matrices, and are thus relatively large. The 4-byte elements in Quicksort, Gaussian Elimination, Life, and Matrix Multiply are long integers. The 16-byte elements in FFT are complex numbers, represented by a pair of 8-byte double precision floating point numbers. Except for the TSP program, which uses fairly large arrays to store partial solutions, the elements being accessed are quite small. Thus, if coherence is maintained strictly on a per-object basis,

Type	FFT	Qsort	Gauss	Life	Mult	TSP
Private						
RdMostly						
WriteOnce	16K				4.9K	324
Result						
Prod/Cons						
Migratory						12K
WriteMany	32K	4K	2.5K	5K	2.5K	24K
GeneralRW						40

TABLE 3: Avg Access Size (bytes)  
(By Object)

Type	FFT	Qsort	Gauss	Life	Mult	TSP
Private	16	4	4	4	4	
RdMostly			4			12
WriteOnce	16		4		4	168
Result			4		4	
Prod/Cons			4	4		
Migratory	16	4	4			114
WriteMany	16	4	4			1.5K
GeneralRW						29

TABLE 4: Avg Access Size (bytes)  
(By Element)

the coherence protocol moves much larger units of memory than the program requires. However, several large objects can be handled easily on a per-object basis. For example, the *Write-Once* (e.g., the input arrays in Matrix Multiply) objects can be replicated. It is therefore advantageous that the coherence protocol knows the size of the shared data objects, and their internal elements, in addition to their sharing behavior.

Tables 5 and 6 present data specific to *Write-Many* objects. The average number of different objects accessed between synchronization points indicates the average number of delayed updates that will be queued up at a time. If this number is small, as the data indicate, managing the queue of delayed updates may not require significant overhead. The remaining *Write-Many* data are also encouraging. *Write-Many* objects are written about one-half as many times as they are read. Large numbers of accesses occur between synchronization points. We call a series of accesses to a single object *by any thread* between two synchronization points in a particular thread a “no-synch run.” The large size of the no-synch runs indicate that delayed updates offers substantial performance improvement. No-synch runs differ from Eggers’s “write-runs” in that they do not end when a remote thread accesses the object, but rather whenever a thread synchronizes. Intuitively, write-runs end when a standard coherence mechanism, such as write-invalidate and write-update, would ensure consistency. No-synch runs end when the programmer *requires* consistency.

Table 7 presents the data recorded for locks, and contains both good and bad news. The good news is that the same thread frequently reacquires the same lock, which can be handled locally. Another piece

	FFT	Qsort	Gauss	Life	Mult	TSP
Avg No Diff Objs Accsd Btwn Synchs	1	2	1.7	1	2	1.5
Avg No of Local Accs Btwn Synchs	8.3K	160	510	3.1K	38	20
Avg No of Loc Writes Btwn Synchs	3.5K	36	110	320	37	12
Avg No of Rmt Accs Btwn Synchs	48K	52K	17K	9.4K	2.4K	47
Avg No of Rmt Writes Btwn Synchs	21K	10K	3.6K	970	12K	28

TABLE 5: Write-Many Data (By Object)

	FFT	Qsort	Gauss	Life	Mult	TSP
Avg No Diff Objs Accsd Btwn Synchs	96	20	5.5			1.0
Avg No of Local Accs Btwn Synchs	817	127	36			1.3
Avg No of Loc Writes Btwn Synchs	375	35	74			1.1
Avg No of Rmt Accs Btwn Synchs	1.5K	1.5K	1.9K			2.6
Avg No of Rmt Writes Btwn Synchs	750	720	610			2.4

TABLE 6: Write-Many Data (By Element)

of good news is that usually the number of threads waiting on the same lock is quite small, indicating that lock arbitration will not require excessive network traffic. The bad news, in terms of being able to achieve the same performance on distributed memory multiprocessors as on shared memory multiprocessors, is that we observe small delays between attempts to acquire a lock and lock acquisition. Even an optimized distributed lock scheme requiring only a single message to release and reacquire the lock will be hard pressed to exhibit this small delay for feasible network latencies in a distributed system.

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

1. There are very few *General Read-Write* objects. Coherence mechanisms exist that can support the other categories of shared data objects efficiently, so a cache coherence protocol that adapts to the expected or observed behavior of each shared object will outperform one that does not.
2. The conventional notion of an object often does not correspond to the appropriate granularity of

	FFT	Qsort	Gauss	Life	Mult	TSP
Avg dt Btwn Lock and Acquire ( $\mu$ sec)			83	1900		350
Pct This Acquire Same Thrd as Last			41	47		25
Avg No of Thrds Waiting on Same Lock			0	.6		.4

TABLE 7: Lock Data

data decomposition for parallelism. Often it is appropriate to maintain coherence at the object level, but sometimes it is more appropriate to maintain coherence at a level smaller or larger than an object. Thus, a cache coherence protocol that adapts to the appropriate granularity of data decomposition will outperform one that does not.

## 4 Simulation Study

To test our hypothesis that adaptive caching mechanisms can outperform standard static caching mechanisms, we simulate two memory coherence mechanisms (write-invalidate and write-update) and two coherence mechanisms that are well-suited for particular types of sharing. All simulation runs are fed identical input streams from the logs generated from actual running programs. The simulation model allows us to select the cache coherence mechanism and set the cache line size for each run. It assumes an infinite cache so no replacement is performed except as required for coherence. The simulation collects information such as: the total number of shared memory accesses, the number of transactions, the total amount of data transmitted (bandwidth consumed), and the number of cache misses. For the discussion below, when the term *line size* is used in the context of a distributed shared memory system, it refers to the minimum granularity of memory that the system handles.

In the first simulation, we compare write-invalidate and write-update with our delayed update mechanism. With delayed updates, whenever the updates are propagated, the remote caches are updated rather than invalidated, so the worst case performance of this mechanism should be equivalent to that of a standard write-update cache. However, if multiple accesses to a single shared variable occur between user-specified synchronization points, delayed updates can significantly improve on a standard write-update mechanism.

The delayed update mechanism is particularly useful when the size of the elementary shared data items is smaller than the cache line size, and there can be a significant amount of false sharing. On the other hand, it must be recognized that false sharing can be minimized by prudent memory allocation by the compiler and/or the programmer. Since the programs used in these simulations are not optimized to avoid false sharing, the following results for write-invalidate and write-update must be interpreted as an upper bound on the negative effects of false sharing.

Table 8 presents the results of simulating the performance of the parallel Quicksort algorithm. The adaptive caching mechanism significantly reduces the amount of bus traffic required to maintain coherence. Compared to write-invalidate, the delayed update mechanism reduces the amount of bus traffic by 31%, 38%, and 52% for 4, 16, and 64 byte cache lines, respectively. Compared to write-update, the delayed update mechanism reduces the amount of bus traffic by 35%, 73%, and 86% for 4, 16, and 64 byte cache lines, respectively. The improvement is caused by the fact that Quicksort performs many writes to shared data objects between synchronization points, so many updates to the same data object are combined before they are eventually propagated. These results are a conservative estimate of the benefits of a delayed update mechanism, because updates to different data objects being sent to the same remote cache were counted as separate transfers. An efficient delayed update mechanism would coalesce updates to the same cache.

In the second simulation, we compare the standard coherence mechanisms with a write-invalidate mechanism that brings an entire object into the cache whenever any portion of it is accessed. This mechanism helps to alleviate object fragmentation caused by a cache line that is too small to hold an entire object. Both the cache line size and the basic object size that we examine are fairly small, but the results are valid whenever the objects used in a computation are larger than a single cache line. Larger cache line sizes are being proposed, but there will always be problems with very large objects that cannot fit into a single line, such as programs that manipulate rows or columns of a matrix.

Table 9 presents the results of this simulation for the parallel FFT. Since the majority of the data objects accessed by the FFT algorithm are complex variables (consisting of a pair of eight-byte double precision floating point variables), a cache line size of less than 16 bytes is inefficient, but the adaptive mech-

Coherence Mechanism	Line size	Transfers (1000's)	Data Copied (kilobytes)
Delayed Update	4	19.6	87.7
	16	5.48	78.5
	64	1.65	106
Write-Invalidate	4	28.5	114
	16	8.81	141
	64	3.39	217
Write-Update	4	30.2	121
	16	14.9	239
	64	11.5	734

TABLE 8: Simulation of Quicksort

Coherence Mechanism	Linesize	Transfers (1000's)	Data Copied (kilobytes)
Adapts to Object Size	4	11.5	188
	16	11.5	188
	64	3.36	218
Write-Invalidate	4	46.9	188
	16	11.7	188
	64	3.41	218
Write-Update	4	45.2	181
	16	11.3	181
	64	6.00	384

TABLE 9: Simulation of FFT

anism overcomes this by automatically loading the entire complex data object when any part of it is accessed. The adaptive coherence mechanism requires the same amount of bandwidth as the write-invalidate mechanism, but when the cache line size is 4 bytes, it does so with 25% as many messages (thus, the average message is 4 times as large). The slight differences between write-invalidate and the adaptive scheme for cache line sizes larger than 4 bytes are caused by the relatively few accesses to objects that are even larger than complex variables, such as threads and monitors.

These two examples provide evidence that adaptive cache coherence mechanisms can significantly improve upon the performance of standard cache coherence mechanisms in a distributed shared memory system where all coherence is performed in software and network latencies are relatively high.

## 5 Related Work

Archibald and Baer discuss a variety of cache coherence protocols [3], most of which are variations of *write-invalidate* and *write-update*. Each works well in some instances and poorly in others. For example, when a single data item is frequently read and written by multiple processors (*fine-grained sharing*), a write-update cache tends to outperform a write-invalidate cache because the data item always resides in the local caches, and the needless cache misses and reloads after each invalidation are avoided. On the other hand, write-invalidate caches outperform write-update caches when one processor is performing most of the reads and writes of a particular data item or when a data item migrates between processors (*sequential sharing*). This is because after the invalidation associated with the first write to a data item, a write-invalidate cache does not needlessly broadcast the new value during subsequent writes.

Archibald described a cache coherence protocol that attempts to adapt to the current reference pattern and dynamically choose to update or invalidate the other copies of a shared data object depending on how they are being used [2]. His protocol is designed for hardware implementation, and therefore is fairly simple and not as aggressive in its attempts to adapt to the expected access behavior as what we propose. Nevertheless, his simulation study indicates that even a simple adaptive protocol can enhance performance.

Other adaptive caching schemes have been proposed, including competitive snoop caching [10] and read-broadcast [11]. Each appears to be appropriate for particular types of sharing behavior, and we plan to examine them in more detail as our work continues.

Weber and Gupta attempt to link the observed invalidation patterns back to high-level applications program objects [18]. They distinguished several distinct types of shared data objects: *Code and read-only*, *Mostly-read*, *Migratory*, *Synchronization*, and *Frequently read/written*. The first four of their categories have corresponding categories in our classification and are handled similarly. *Frequently read/written* data had the worst invalidation behavior and their coherence protocols could not handle them efficiently. They advised that this type of data object be avoided if at all possible. Our approach is more aggressive. We have identified two types of shared data objects (*Write-Many* and *Producer-Consumer*) that would fall into Weber-Gupta's *Frequently read/written* category, yet can be handled efficiently by an appropriate protocol.

Eggers and Katz analyze the sharing characteristics of four parallel programs [8]. Two of the applications exhibited a high percentage of *sequential sharing* while the other two exhibited a high degree of *fine-grained sharing*. This indicates that neither write-broadcast nor write-invalidate is clearly better for all applications. The observed low contention for shared data objects led them to conclude that write-invalidate outperforms write-broadcast on the average, but one major cause of their low contention is their programming methodology (SPMD). Each process executes on an independent piece of data, and the only contention occurs at the central task queue. Thus, most computation occurs without contention, which favors write-invalidate.

## 6 Conclusions

We have characterized several distinct categories of shared data access: *Write-once*, *Write-many*, *Producer-Consumer*, *Private*, *Migratory*, *Result*,

*Read-mostly*, *Synchronization*, and *General Read-Write*. We have briefly described efficient memory coherence mechanisms for each of these categories. By studying logs of shared memory accesses of a variety of parallel programs, we have shown that a large percentage of shared memory accesses fall in these categories. These results support our contention that adaptive cache coherence techniques that are designed to exploit the anticipated or observed access behavior of a particular data object can significantly outperform standard cache coherence mechanisms, at least in the context of a distributed shared memory system. We have also described two simulations studies to further support this hypothesis.

## Acknowledgements

The authors would like to thank the referees and the members of the Rice computer systems group (Elmootazbellah Elnozahy, Jerry Fowler, David Johnson, Pete Keleher, and Mark Mazina) for their helpful suggestions. Trung Diep assisted with the acquisition and plotting of program speedup data.

## References

- [1] Anant Agarwal, Richard L. Sites, and Mark Horowitz. ATUM: A new technique for capturing address traces using microcode. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 119–127, June 1986.
- [2] James Archibald. A cache coherence approach for large multiprocessor systems. In *International Conference on Supercomputing*, pages 337–345, November 1988.
- [3] James Archibald and Jean-Loup Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems*, 4(4):273–298, November 1986.
- [4] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proceedings of the 1990 Conference on Principles and Practice of Parallel Programming*, March 1990.
- [5] 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.

- [6] 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.
- [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] James R. Goodman, Mary K. Vernon, and Philip J. Woest. Efficient synchronization primitives for large-scale cache-coherent multiprocessor. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Systems*, April 1989.
- [10] A. R. Karlin, M. S. Manasse, L. Rudolph, and D.D. Sleator. Competitive snoopy caching. In *Proceedings of the 16th Annual IEEE Symposium on the Foundations of Computer Science*, pages 244–254, 1986.
- [11] Kai Li. Private communication. March 1990.
- [12] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.
- [13] Tom Lovett and Shreekanth Thakkar. The Symmetry multiprocessor system. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 303–310, August 1988.
- [14] Umakishore Ramachandran and M. Yousef A. Khalidi. An implementation of distributed shared memory. *Distributed and Multiprocessor Systems Workshop*, pages 21–38, 1989.
- [15] 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.
- [16] K. So, F. Darema-Rogers, D. George, V.A. Norton, and G.F. Pfister. PSIMUL: A system for parallel simulation of the execution of parallel programs. Technical Report RC11674, IBM Research, 1986.
- [17] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1987.
- [18] 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.