To appear in *the Proceedings of Supercomputing '95*, to be held in San Diego in December 1995.

Architectural Mechanisms for Explicit Communication in Shared Memory Multiprocessors*

Umakishore Ramachandran
Gautam Shah
Anand Sivasubramaniam
Aman Singla
Ivan Yanasak

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280
Phone: (404) 894-5136
Fax: (404) 894-9442
e-mail: {rama,gautam,anand,aman,yanasak}@cc.gatech.edu

Abstract

The goal of this work is to explore architectural mechanisms for supporting explicit communication in cache-coherent shared memory multiprocessors. The motivation stems from the observation that applications display wide diversity in terms of sharing characteristics and hence impose different communication requirements on the system. Explicit communication mechanisms would allow tailoring the coherence management under software control to match these differing needs and strive to provide a close approximation to a zero overhead machine from the application perspective. Toward achieving these goals, we first analyze the characteristics of sharing observed in certain specific applications. We then use these characteristics to synthesize explicit communication primitives. The proposed primitives allow selectively updating a set of processors, or requesting a stream of data ahead of its intended use. These primitives are essentially generalizations of prefetch and poststore, with the ability to specify the sharer set for poststore either statically or dynamically. The proposed primitives are to be used in conjunction with an underlying invalidation based protocol. Used in this manner, the resulting memory system can dynamically adapt itself to performing either invalidations or updates to match the communication needs. Through application driven performance study we show the utility of these mechanisms in being able to reduce and tolerate communication latencies.

Key Words: Shared memory multiprocessors, cache coherence, latency tolerating techniques, synchronization, explicit communication

*This work has been funded in part by NSF grants MIPS-9058430 and MIPS-9200005, and an equipment grant from DEC.

1 Introduction

Shared memory multiprocessors are attractive and have been accepted as the model of choice for parallel programming due to ease of programming. One of the main problems in such machines is that the implicit communication via memory accesses could result in considerable network overhead due to the latency for remote accesses. The key to achieving good performance is to keep these overheads low. Coherent caches, and relaxed memory consistency models strive to keep the latencies for remote accesses low. Further, careful layout of the data in the caches to avoid false-sharing, and sufficiently large caches at each node would help in reducing the false-sharing and capacity misses (and the ensuing latencies and network traffic) in the caches. Despite all these tricks, an application can experience a considerable amount of network latency and contention due to cache misses resulting from true sharing. To reduce or tolerate these latencies, shared memory machines often include mechanisms for explicit non-blocking communication such as *prefetch* (which is receiver-initiated and brings the data to the processor before it is actually used), and *poststore* (which is sender-initiated and sends the data as soon as it is produced to potential consumer processors). Further, there have been several recent proposals to provide message-passing style communication primitives in a shared memory machine [16, 20, 22]. We refer to a specific combination of memory model and coherence protocol, together with some explicit communication primitives as a *memory system* in this paper.

All such latency reducing and tolerating mechanisms have simply one goal, namely, to make the parallel machine appear as close as possible to a zero-overhead machine from the point of view of an application. The appropriateness of a particular technique very much depends on the specifics of the communication pattern in the application, and it has been shown [19] that no one technique is universally applicable for all applications.

While there has been considerable amount of recent work in using synchronization information to relax memory consistency models, there has been relatively little research in exploiting synchronization information to trigger coherence actions as well as guide the use of explicit communication in shared memory machines. It is our belief that software-directed coherence management offers the ability to achieve this. We propose a set of simple primitives which are essentially generalizations of poststore and prefetch. The primitives allow selectively sending updates to a set of processors, and prefetching a stream of data spanning multiple cache blocks. As will become evident in Section 4, the selective update mechanism determines, either dynamically or statically, the set of processors who need to be sent updates. The set is determined by the sharing pattern of the application. Other than one additional state in the memory directory, the selective-store primitive does not incur any more space overhead compared to any directory-based coherence protocol.

These primitives when used in conjunction with a basic write-invalidate protocol allows the system to dynamically switch between the invalidation and update strategies for enforcing cache coherence based on the sharing pattern in the application. This allows the memory system to reap the advantages of both write-invalidate and write-update protocols with little or no added hardware complexity. We show that with a little bit of help from the application either in terms of dominant phases where the sharing pattern in the application is expected to change, and/or in terms of association of locks with the data they govern, we can considerably reduce the latencies for true-sharing using these primitives under software control (i.e. compiler). Our work is in line with the current trend in shared memory multiprocessor design [32, 21] that emphasizes the need for flexibility in the choice of the coherence protocol that a machine provides to meet the changing needs of an application.

Using an application-driven approach, this research makes several contributions. The primary thesis of this research is that software-directed management of caches with appropriate hardware assists for explicit communication is

a promising approach to achieving high performance in shared address space parallel machines. The main contribution is the set of architectural mechanisms (Section 4) for software-directed cache management. An equally important contribution is identifying the inherent communication patterns in the selected applications and matching the communication primitives to these patterns. An evaluation of four memory systems, ranging from purely hardware controlled to a combination of hardware/software mechanisms in the context of these specific applications is the third contribution of this work.

We give the motivation for our study in Section 2, the application characteristics that drive the choice of mechanisms in Section 3, the architectural mechanisms for explicit communication in Section 4, the details of the four memory systems being compared in Section 5, the performance results in Section 6, the related work in Section 7, and concluding remarks in Section 8.

2 Motivation

Shared memory machines provide the convenience of a shared address space across processors easing the burden of writing parallel programs. But supporting a shared address space alone (as is the case in NUMA machines without private caches) can result in performance penalties when processors access data that is external to their local memory. Since applications tend to exhibit spatial and temporal locality, it is important to provide hardware support for efficient locality management in such machines. In shared memory machines, it is customary to provide private caches with each processor that facilitate the exploitation of locality by replicating data at each node. As we mentioned earlier, the consistency model presented by the memory system and the coherence protocols used for the caches combine to keep the overheads associated with locality management low. Early designs [28, 14] used sequential consistency [24] for the memory model which presents a uniformly consistent view of shared memory for all processors at all times. Recently, use of some form of relaxed memory consistency model has been proposed as a means to improve performance of cache-based shared memory multiprocessors [27, 1, 13]. The basic premise is that most shared memory applications follow some synchronization model and expect consistent views of data only at well-defined synchronization points. Therefore, architectures based on such relaxed memory models overlap communication with computation by allowing global operations (such as invalidations) to go on in the background and only requiring such operations complete before a synchronization operation.

Coherence management in a cache-based shared memory multiprocessor is an equally important issue, in addition to the specific memory consistency model. Cache coherence protocols broadly fall into two categories: *write-invalidate* and *write-update*.

Invalidation-based schemes are more suited to migratory data and can become inefficient when the producer-consumer relationship for shared data remains relatively unchanged during the course of execution. On the other hand, update-based protocols can result in significant overheads due to repeated updates to the same data before they are used by another processor, as well as redundant updates when there are changes to the sharing pattern of a data item. The update and invalidation based schemes thus have their relative advantages and disadvantages, and based on application characteristics one may be preferable over the other. Invalidations are useful when an application changes its sharing pattern, and updates are useful to effect direct communication once a sharing pattern is established.

By dynamically switching between updating and invalidating, the competitive update [30] scheme attempts to get the advantages of both schemes. The basic protocol in this scheme is update-based; however, a history of redundant updates to each individual cache line is tracked and used by a processor to self-invalidate and stifle future updates to

this line. Similarly, Dahlgren et al. [11] show that we can benefit by extending hardware cache coherence mechanisms with techniques such as adaptive prefetching and migratory sharing. But such mechanisms which rely entirely on the hardware may not provide the flexibility to adapt to the inherent characteristics of different applications. Recognizing that a fixed protocol for cache consistency may not suffice, there have been recent proposals [21, 32] to have a menu of coherence protocols implemented in software so that an application can choose the right one depending on its communication pattern. Falsafi el al. [15] show that we can gain substantially by matching the coherence protocol to an application's communication pattern and memory semantics. However, there is a concern of increasing the programming complexity by requiring the application developer to choose the right coherence protocol.

Therefore, there is a need for striking a balance between flexibility and programming ease. We need hardware/software mechanisms that give the flexibility for dynamically adapting the way coherence actions are effected, while at the same time not imposing a heavy burden on the programmer in terms of programming complexity. This is the primary motivation behind this work. Toward this end, we first need to look at applications and study their communication/sharing characteristics (Section 3). These characteristics can in turn suggest primitives that should be supported in the underlying system which can be implemented either in hardware or software (Section 4). As far as possible, our objective is to relegate the task of using these primitives to the compiler so that the programming complexity for the application programmer is not increased. To achieve this objective, we need to determine the knowledge that can be gleaned from the application by the compiler and how this information can be used to automate the process of issuing the right commands to take the appropriate consistency actions at runtime (Section 5). In the next section, we examine the communication characteristics of typical parallel applications to help synthesize the mechanisms that will help in reducing their communication overhead.

3 Application Characteristics

Studying communication characteristics of parallel applications can intuitively suggest architectural primitives that can be beneficial. Communication resulting from the data access pattern of applications may be broadly classified into *static* and *dynamic*. If the data access pattern of an application can be pre-determined at compile-time then the resulting communication is defined as static. FFT (Fast Fourier Transform), SOR (Successive Over Relaxation), and Matrix Multiplication, are examples of such applications where the entire execution displays a static communication pattern. There are several other applications where the communication is static at least for certain phases. For instance, global sums which use a static logarithmic tree structure to effect communication are frequently used in applications. For such static communication, the producer knows the exact consumer set and can directly propagate the data item to the processors in this set. An invalidation-based protocol would be inefficient in these cases because of unnecessary invalidates to the consumer set. A naive update-based protocol may send unnecessary repeated updates, and updates to processors which may no longer be in the consumer set, particularly when the set changes across phases of execution. Hence, in order to optimize on static communication, it would be beneficial to have an explicit communication mechanism wherein a producer can selectively propagate a data item to a set of consumers that it specifies. The compiler may be able to use such a primitive to effect the communication using the application knowledge regarding the data access pattern without placing undue burden on the programmer.

Several other applications fall in the dynamic category where the communication varies with the dynamics of program execution. This dynamism poses some problem in determining the consumer set for effecting explicit communication. Fortunately, synchronization information in the program can be used sometimes for triggering such

actions. For instance, consider the data items produced in a critical section under the control of a mutual exclusion lock. Figure 1 shows this scenario where P0 writes a variable in a critical section which is subsequently read by P1 after obtaining the lock. If we had some mechanism for P0 to selectively send data to P1, that would reduce the observed data access latency for P1. Clearly some programmer help is needed in associating data items with the lock structure. Also some help is needed in implementing the lock library calls such that any waiting processor for a lock is identifiable through the lock structure. Assuming such help is available, if there was a mechanism that allows explicit communication then the compiler could use the association of data with the lock for P0 to send the data when produced to any waiting P1 (which can be looked up using the lock structure). Figure 1 shows this sender-initiated transfer using broken lines. This scheme potentially has two benefits over an invalidation scheme which is usually considered the best candidate for such migratory sharing pattern. Firstly, P1 will not incur a miss penalty for data access subsequent to procuring the lock. This is one of the advantages of combining synchronization with data transfer [5, 25, 18, 33]. Secondly and more importantly, the transfer of data from P0 to P1 can be started as soon as the data item is ready to be written and does not have to be delayed until the unlock point. Essentially, the synchronization information is being used at the earliest possible time to trigger the coherence action through such an explicit communication primitive. In application scenarios exhibiting significant lock contention (where combining synchronization with data transfer helps significantly) such a mechanism will help even more since there is very high likelihood for a consumer to be waiting for the lock. Producer-consumer applications such as those which manage queues dynamically can hope to gain from this scheme.

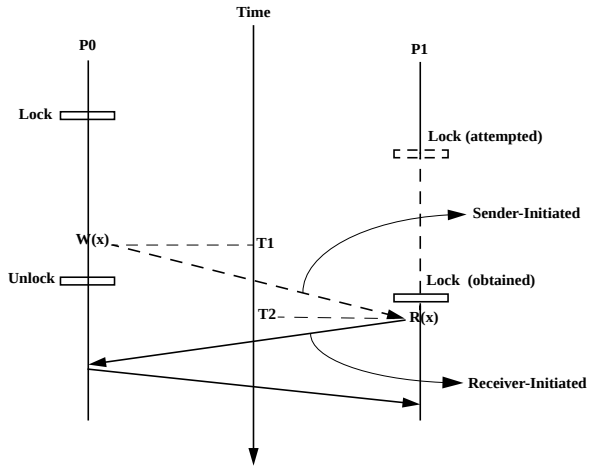

Figure 1: Accesses for variables governed by locks

Many applications in the dynamic category go through phases where the sharing pattern does not change within a phase but changes across phases. The producer may not be able to provide a consumer set to effect explicit communication as in the static cases. The Nbody simulation using the Barnes-Hut algorithm is a typical application which exhibits such behavior. The application simulates the motion of bodies over time steps, and in each step the motion of a body is determined by the state of other bodies that can affect its motion. In successive steps, the bodies are not expected to move drastically and the communication pattern would thus remain the same for these steps. But, over several time steps the bodies can move to establish a different sharing pattern. As we mentioned earlier, an invalidation based protocol would be unsuitable for successive steps where the sharing pattern does not change. A simple minded update protocol would send updates to a superset of the old consumer set and would not adapt itself to

changes in the sharing pattern. For such applications, it would be useful to have a mechanism that adapts itself to a new sharing pattern dynamically and sends updates to the consumers in the new set. For another class of applications such as EM3D [9] and CG in the dynamic category, the communication pattern may not be apparent at compile time but the communication becomes deterministic once the program begins execution (i.e. the input data is read in). Since communication is dependent on the input data set, the compiler cannot explicitly give the consumer set when the data item is produced. Dynamically providing an explicit consumer set as is done in [15] would involve a substantial change in the application code. However it can be seen that this sharing pattern is just a special case of the dynamic sharing pattern observed in applications such as Barnes-Hut. Therefore, if we had a mechanism for handling the general case then that would work for this special case as well. Use of this primitive can be directed by the compiler based on the synchronization information in the program (Section 5).

In all the above cases, we have attempted to exploit sender-initiated data transfer wherein the producer sends data to the consumers. Instead of a naive update based protocol we use application knowledge in determining the consumer set for sending updates. In the static case, the information is gleaned directly from the application. In the first of the two dynamic cases, the information could be obtained using association of data with locks and the synchronization structure itself (such as a lock). In the second dynamic case, the information has to be gleaned by tracking the sharing pattern of the application at runtime. However, there are several applications with dynamic communication characteristics where the sharing information is simply not available at the time the data is produced. For instance, in applications such as IS and CHOLESKY where there is no lock contention [35], the producer cannot effect the data transfer since it does not know the next consumer. For such scenarios, we need to resort to receiver-initiated communication where the consumer explicitly requests the data item. An advantage with sender-initiated communication is that the data transfer can be started as soon as the data item is produced. Receiver-initiated communication can at best be started immediately following a synchronization point as is shown in Figure 1 where P1 performs a read after acquiring the lock. Potentially, a considerable amount of overlap (T2 - T1 in Figure 1) can be lost because of this. For effectiveness of receiver-initiated transfer, the request for the data should be made as soon after the synchronization operation as possible and a sufficient gap should exist between the time the request is made and the time the data is actually needed. The prefetch mechanism is usually the best bet in such situations. However, since it is in units of the cache line it has limitations when used for fetching large structures. For instance, with reference to Figure 1 if we assume that the critical section is used to manipulate a queue; then several fields of the queue object (head, tail, etc.) may need to be prefetched all roughly at the same time. Since these fields may span several cache lines this would result in multiple prefetch request messages. To cut down on the number of messages, and to enable streaming of the data back to the requesting processor it would be beneficial to have a prefetch-like primitive that is decoupled from the cache line size. Multithreading [38] is another latency tolerating technique that can be useful in some cases where the prefetch cannot be initiated early enough. But this issue is beyond the scope of this paper and we do not discuss it further.

Having identified application needs towards minimizing communication overhead, we present architectural mechanisms that help us accomplish these goals in the next section.

4 Architectural Mechanisms

4.1 The Base Hardware

The base hardware is a CC-NUMA machine. Each node has a piece of the shared memory with its associated full-mapped directory information, a private cache, a write buffer [27], and a write merge buffer [12]. Figure 2 shows the

relevant details of the node architecture. A cache block can exist in one of three states, INVALID, VALID and DIRTY. VALID state is a potentially shared clean state, while DIRTY state is an exclusive state requiring a write-back on cache line replacement. The state information in the memory directory, and the exact state transitions will depend on the specifics of the memory systems to be presented in the next section. The write buffer is used in the implementation of memory systems with weaker memory models. For such memory systems, the write buffer keeps track of pending writes, and stalling the processor on a write when the buffer is full. The write buffer is flushed when a processor reaches a synchronization fence. Read operations are assumed to be blocking, and the processor is stalled until the operation is complete. The write-merge-buffer [12] is a small associative cache that records updates to a cache block from the processor. It is specific to memory systems that use updates and helps combine updates to the same cache line into a single message to the directory controller. A write-merge-buffer entry is moved out to the write-buffer when an entry has to replaced (LRU style) to accommodate a processor write to a cache block not already in the merge buffer, or when the processor reaches a synchronization fence. The interconnection network is a 2-D mesh. Links in the North, South, East and West directions enable a processor in the middle of the mesh to communicate with its four immediate neighbors. Processors at corners and along an edge have only two and three neighbors respectively. Messages are circuit-switched and are routed along the row until they reach the destination column, upon which they are routed along the column.

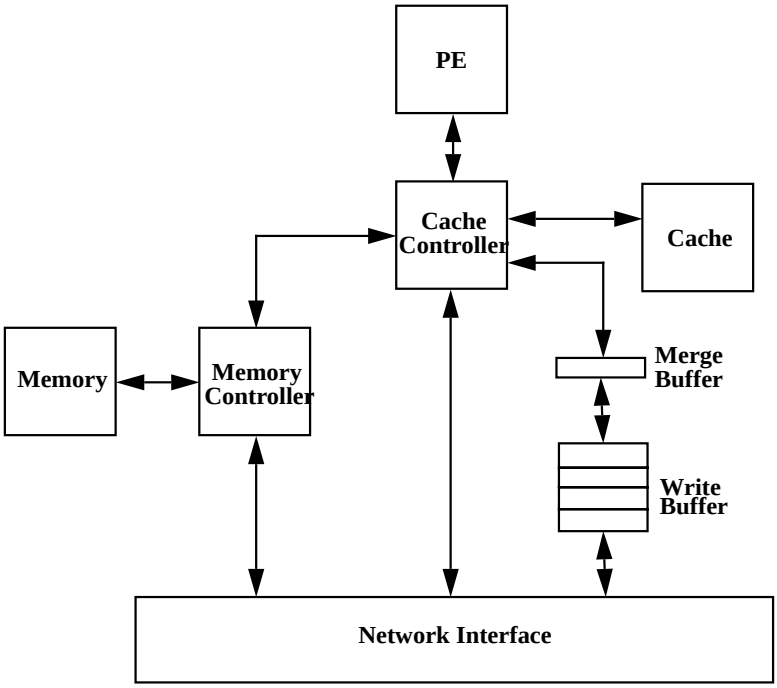

Figure 2: The Node Architecture

The above hardware is not significantly different from the node and directory structure of other CC-NUMA machines such as Stanford Dash [27], or MIT Alewife [2]. To accommodate the diversity of the explicit communication mechanisms we plan to propose in the next few subsections, we require the memory directory controller to be slightly more flexible. As in any other design, the directory controller will normally use the information in the “presence" vector for taking coherence actions. However, the directory controller can also be told to ignore the “presence" vector and explicitly send coherence messages (invalidations or updates) to a set of processors specified to it. The hardware

cost for this flexibility is negligible (just a few gates for generating the processor set mask), and it adds at most a two-level gate delay to the latency. As will become evident the different explicit communication mechanisms that we propose will use this feature to generate the processor set to match the communication requirements of the application.

4.2 The Mechanisms

We now present a set of architectural mechanisms for direct communication to be implemented on top of the base hardware. We will use these mechanisms to define a new memory systems in the next section. These mechanisms are assumed to be implemented through the cache and directory controller hardware. The important point to note is that these mechanisms are integrated with the basic underlying directory-based coherence maintenance. In this sense they are very different from the explicit communication primitives such as those proposed in MIT Alewife [22] or Stanford Flash [39], in that there is no address space management nor explicit coherence maintenance burden at the application level on the programmer for using our primitives. Of the proposed primitives, PSET_WRITE provides explicit communication for the static case, SYNC_WRITE provides explicit communication for the dynamic case within critical sections, and the two variants of the RSTREAM primitive provide arbitrary-sized prefetch capabilities. For the dynamic case with varying sharing patterns across different phases of execution, we provide a primitive called SEL_WRITE to be used in conjunction with normals reads and writes.

PSET_WRITE(address, value, pmask): This primitive is used for static communication where the processor issuing this request gives an explicit mask (*pmask*) of the processors to which the new *value* for the *address* is to be propagated. The request is sent to the directory controller which writes the new value to memory, invalidates any existing cached copies of this memory block (obtained from the “presence" vector in the corresponding directory entry) that are not specified in the pmask, and sends the modified memory block to all the processors in the pmask. The corresponding cache block is set to the VALID state in the processors specified in the pmask, the directory state is also set to VALID for this block, and the presence vector is modified to reflect the new set of active sharers.

SYNC_WRITE(address, value, pmask): This primitive is just a special case of the PSET_WRITE primitive to be used by the software when association of locks with the data they govern is available via programmer annotations. Presumably this primitive is issued by a processor that is currently holding a mutual exclusion lock. The compiler can generate code by de-referencing the lock structure governing the critical section to send update to the first processor (if any) waiting on this lock. Thus ‘pmask’ is a singleton processor that is next in line to get the lock. Upon issue of this request, the directory controller updates the memory block, invalidates any cached copies for this block, and sends the modified memory block to the processor identified by pmask.

SEL_WRITE(address, value): The SEL_WRITE (Selective Write) primitive has the same intent as PSET_WRITE with the difference that the set of processors for sending updates is dynamically determined using the directory information. Upon receiving this request, the directory controller writes to memory, and sends updates to the caches that have a copy of this memory block as determined from the presence vector. As we mentioned earlier, applications sometime go through phases where the sharing pattern does not change within a phase but changes across phases. Thus within a phase this primitive serves to send updates. Across phases, when the sharing pattern changes the presence vector has to modified appropriately to reflect this change. In the next section when we describe the memory system that incorporates this mechanism we will show how this modification is accomplished.

The above three explicit write mechanisms compute the following two boolean expressions at the directory to figure out if a processor i should be updated or invalidated

update(i) == TRUE if (((pmask==0) && presence[i]) || pmask[i])
invalidate(i) == TRUE if (!update[i] && presence[i])
where pmask is the processor mask passed by the explicit write primitive (0 for SEL_WRITE), and presence[i] indicates the corresponding presence bit.

RSTREAM(address, # bytes), RSTREAM_EX(address, # bytes): These primitives are generalizations of the prefetch mechanism for use in receiver-initiated communication scenarios. Traditional prefetch operations work at a block level and attempt to bring the block to the issuing processor in a VALID or EXCLUSIVE mode. The RSTREAM and RSTREAM_EX primitives allow specifying the number of bytes that need to be prefetched in addition to shared or exclusive mode of access. The memory directory treats a single request as a sequence of block requests (commensurate with the number of bytes requested) starting with the block containing 'address'. Appropriate coherence actions are taken for each block before sending the block to the requester. Semantically these primitives work as though the processor issued multiple block level prefetch requests, but the optimization comes in being able to bundle multiple prefetch requests into one. The request is sent to the directory controller that handles 'address'. If the request straddles multiple pieces of the distributed shared memory then the requester will get back only the blocks that can be serviced by this controller. Clearly for performance reasons (i.e. to reduce the number of misses), it will be best if the requested blocks can be serviced by a single directory controller.

5 Memory Systems

The design choices in realizing a cache-coherent shared memory system include the *memory consistency model*, the *cache coherence protocol*, and any *explicit communication primitives*. In this study we will consider Sequential Consistency (SC) [24] and Release Consistency (RC) [17] as the two choices for the memory model. Write-invalidate and write-update are the choices for the cache coherence protocol. The explicit communication primitives that we consider are the mechanisms that we proposed in the previous section. A specific combination of these three parameters gives rise to a unique memory system.

While the choice of the memory model and coherence protocol to implement the model may be independent in principle, certain combinations may not perform very well and hence can be eliminated from consideration. For example, SC with a write-update protocol is not expected to perform very well in large-scale multiprocessors. Thus in this work we consider the following four (SCinv, RCinv, RCupd, RCexp) memory systems for comparison.

SCinv: A memory system with SC memory model, a Berkeley-style state transition write-invalidate cache coherence protocol, and with no other explicit communication primitive is used as the base memory system. In SCinv, as should be evident the processor stalls on every read and write operation that involves the network. Therefore, the write buffer which we described in the base hardware (see Section 4) does not play any role for this memory system.

RCinv: This memory system uses RC memory model, a Berkeley-style write-invalidate protocol, and no explicit communication primitives. In this system, a processor write that misses in the cache is simply recorded in the write-buffer without stalling the processor. The write is completed when ownership for the block is obtained from the directory controller, at which point the write request is retired from the buffer. The processor may have to stall if the write-buffer is full when it incurs the write-miss. It may also have to stall if the buffer is non-empty at a release point.

RCupd: This memory system uses RC memory model, a simple write-update protocol not unlike the one used in Firefly [37] for state transitions, and no explicit communication primitives. From the point of view of the processor,

writes are handled exactly similarly as in RCinv. At a release point, the processor stalls until the pending writes in the write-buffer and the write-merge-buffer are complete.

RCexp: This is a memory system which is RCinv augmented with all the new primitives for explicit communication proposed in the previous section. Figure 8 is a state transition diagram of the cache and directory for this new system which shows how the new primitives can be integrated with the underlying write-invalidate protocol. The normal read and write operations work as in a traditional invalidation-based scheme. The state transitions for RSTREAM and RSTREAM_EX are similar to normal read and write respectively. PSET_WRITE and SYNC_WRITE invalidate processors with valid copies that are not in the given mask and update the ones specified in the mask setting the state to VALID in the directory and caches. The SEL_WRITE primitive is the most interesting in terms of the coherence actions. This primitive serves two purposes: it is an explicit way for a processor to inform the hardware that a sharing pattern has been formed; and it also serves to send updates to the processors that are in this sharer set. The way these two purposes are achieved is explained below.

Forming a Sharer Set Dynamically: Consider a memory block with no replicas in any cache. As each processor incurs a read miss for this block, the block is fetched from the directory setting the appropriate presence bit in the directory for this block. Thus the presence vector reflects the current sharing pattern. Now consider a processor that wishes to write to this memory block. A normal invalidation-based protocol would simply invalidate the current copies to give exclusivity to the writer. However, if the program is written to obey some synchronization model (such as RC) then no processor should be reading this particular address being written while some processor is attempting to write it. Further, after a release point it is highly likely (assuming that the sharing set does not change) that the processors who had copies of this block prior to the write will re-read the block. This is the key observation behind the SEL_WRITE primitive. Upon receiving this request, the directory controller sends updates to the processors in the presence vector. The memory is updated as well and the state of this block in the caches having a copy of this block remain VALID. The directory state is then set to SV (*Special Valid*) to indicate that a sharing pattern has formed. Subsequent SEL_WRITEs by processors in this set result in updates to the set of sharers.

Changing the Sharer Set Dynamically: We provide two ways by which the sharing pattern can be changed. The first one is reader-initiated and is effected when a new reader tries to join the set. This reader incurs a read miss and sends a request to the directory. If the directory state for this block is SV then it is an indication that a sharing pattern has already been set by the producer. The fact that there is a read-miss for a block in SV state is an indication of a change in the sharing pattern. This is a logical conclusion since this read-miss could have happened only in a phase of the application different from the one in which the SV state was set. Thus the directory will assume that this read-miss request to be its cue for forming a new sharer set. In response to this read-miss request, the directory controller sends invalidations to the existing sharer set given by the presence vector, sends the memory block to the requesting processor, and sets the state for this block to VALID. Subsequent read-misses from other processors for this block would result in the growth of this sharer set until it gets frozen again by the next SEL_WRITE.

Alternatively, we could have simply allowed the new reader to join the sharer set (retaining the SV state). This would allow for the possibility that the new sharer set may have considerable overlap with its immediate predecessor. We consciously decided against this alternative since in the limit such a scheme would degenerate to write-update. Our solution of deciding a new sharer set on each read-miss in SV state seems to be in line with the observed sharing pattern changes in several applications (such as Barnes-Hut) wherein processors join and withdraw from the pattern at

around the same time.

Ideally, we would like for the uninterested processors to be dropped from the sharer set. This can be accomplished either by an explicit primitive (such as the reset-update primitive proposed in [26]) that can be issued under software control, or an implicit mechanism such as competitive update [30] that allows a processor to self-invalidate a block based on the history of updates to that block. It is not clear how the former can be used without considerable help from the application, and the latter has a potential for being too slow in retiring members of the old sharer set. We have not considered these alternatives to changing the sharer set in this work, and plan to do so in the future.

We also provide a writer-initiated method for changing the sharer set. When a processor issues a normal write when the directory state for that block is SV, all cached copies are invalidated giving exclusive ownership to the writer. Thus the old sharing pattern is discarded to be re-formed again with subsequent read misses. The protocol thus uses a mix of invalidations to effect changes in sharing patterns, and updates to effect direct communication once a sharing pattern is established.

Using the Explicit Communication Primitives: As we mentioned in section 3, we would like to leave the task of using the explicit communication primitives in the new RCexp memory system to the compiler so that the programming complexity for the application programmer is not increased. To achieve this objective, we need to determine the knowledge that can be gleaned from the application by the compiler and how this information can be used to automate the process of issuing the right primitive towards optimizing communication. It should be noted that an incorrect use of a primitive in our memory system will not affect the correctness of the execution (since all primitives work through the coherence protocol). The compiler can use data-dependence analysis of the application program towards identifying the write operations that can benefit from the use of PSET_WRITE. For accesses to variables that involve static communication, dependence analysis can reveal the processors in the consumer set to which updates should be propagated.

Since mutual exclusion locks are typically used to regulate accesses to some shared object, it is natural for the programmer to make an association between the object and the lock which governs it. If this association were to be made available in the program (can be easily provided as a language construct itself), the compiler can directly generate SYNC_WRITE calls for the writes to the object within the lock and unlock regions. With regard to the SEL_WRITE primitive, the application programmer can give useful hints to the compiler to demarcate phases of execution and the compiler would substitute normal writes with the SEL_WRITE version for shared variables within such phases. For example, a program may pass through distinct phases of execution such as a Computational Fluid Dynamics application which contains a conjugate gradient kernel followed by an integer sort. In such cases, the programmer can explicitly demarcate these phases, and the compiler can use this information to perform the first write in a phase as a normal write (to destroy the old sharing pattern) and subsequent writes as SEL_WRITEs. In the absence of such hints from the application, the compiler may choose to use SEL_WRITE for all writes to shared objects, and rely on the underlying hardware mechanism for changing the sharer set when new readers join the set. As an aside, it should be noted that use of SEL_WRITE eliminates the false-sharing problem in invalidation-based schemes.

Considerably more experience is needed in analyzing applications to determine how much information can be easily gleaned to automate the process of using these primitives in the compiler. The work by Cytron et al [10], Cheong and Veidenbaum [7, 6], and Min and Baer [29] have goals similar to ours in attempting to reduce the global communication by providing software-directed cache coherence.

6 Performance Evaluation

In this section, we compare the performance of the above-mentioned memory systems using a set of applications. We simulate the relevant details of the shared memory hardware on top of SPASM [35, 36], an execution-driven parallel architecture simulator. Table 1 gives the specific parameters of the base hardware used in the simulation. Our focus is on designing and evaluating mechanisms for tolerating communication overhead resulting from true sharing in applications. Therefore, we assume an infinite cache at each node in our simulations and eliminate capacity misses.

Cache Access	1 cycle
Memory Access	10 cycles
Link Bandwidth	1.65 cycles/byte
Network Messages	8-40 bytes
Cache Block Size	32 bytes
Write Buffer	4 entries
Merge Buffer	1 Line
Network Interface Buffer	16 entries

Table 1: Hardware Parameters

Our main objective is to understand how efficiently these different memory systems are able to cope with the communication overheads present in these applications due to true sharing. These overheads are:

- **read stall**: this is the wait time seen by a typical processor for read-misses;
- **write stall**: this is the wait time seen by a typical processor for write misses;
- **buffer flush**: this is the wait time seen by a typical processor at synchronization points for outstanding writes to complete;
- **synchronization stall**: this is the time seen by a typical processor at synchronization points waiting for other processors (e.g. work imbalance at a barrier, or lock contention). In our results, the synchronization stalls are typically due to mutual exclusion operations and we refer to them as mutex stalls.

Before presenting the simulation results, some general observations are in order. The SCinv system is the base system, which is the simplest in terms of both hardware and programming complexity. The RCinv system is expected to perform better than SCinv due to the ability to tolerate write stalls. The read stall time in RCupd system would be purely due to cold misses while the read stall time in RCinv would arise from cold and coherence misses. Thus we expect RCinv to incur a higher read stall overhead compared to the RCupd. Repeated writes to the same cache block and redundant updates would increase the network traffic in RCupd resulting in two detrimental effects compared to RCinv. The first effect is a possible increase in network contention. The second effect is a possible increase in buffer flush time. Thus for a given application, we expect RCupd to have a lower read stall time while RCinv to have a lower buffer flush time. However, there may be a penalty in using the write-merge buffer in the form of increased buffer flush time at synchronization points. This is due to the fact that flushing from this buffer cannot be overlapped with the computation. We can conceive of other implementations where the merge-buffer is flushed if the network is not loaded. Such an implementation can overcome the stated penalties at the cost of increased network traffic.

So, in order to illustrate the merits of RCexp it is sufficient to show that it tracks read stall times comparable to RCupd, and buffer flush times comparable to RCinv across applications. Further, we would like to incur as few network messages as possible to achieve these effects.

In the next few subsections we discuss results of executing four applications – Barnes-Hut, Cholesky, Integer Sort (IS), and Maxflow – on the four memory systems. We have conducted experiments over a range of processors. The results presented are for 16 processors.

6.1 Barnes-Hut

Barnes-Hut is an N-body simulation application. The application simulates over time the movement of these bodies due to the gravitational forces exerted on one another, given some set of initial conditions. The parallel implementation [34] statically allocates a set of bodies to each processor and goes through three phases for each simulated time step. The regions of space are organized in a tree data structure with the bodies forming the leaves. The first phase (*make-tree*) in each time step associates bodies to a specific region in space, and computes the center of mass for the regions hierarchically. In the second phase (*consumer-phase*, each processor computes the velocity component for the bodies assigned to it which could involve visiting various nodes of the tree. In the third phase (*producer-phase*), each processor computes the coordinates of its bodies in space based on the velocities computed in the previous phase.

As bodies move in space the set of nodes that each body has to interact with in the consumer-phase could gradually change over time. With an invalidation protocol this sharing pattern leads to invalidations during the producer-phase followed by re-reads during the consumer-phase. On the other hand, the change of sharing pattern over time makes this application unsuitable for an update protocol. While this producer-consumer relationship is well-defined, the access pattern during the make-tree phase has a non-deterministic effect on the sharing pattern of the application as a whole. So we conduct two sets of experiments. In the first one, we study the behavior of the memory systems for this fairly random sharing pattern of the application as a whole. In the second one, we study the memory systems for the well-defined producer-consumer sharing pattern by ignoring the interactions in the make-tree phase.

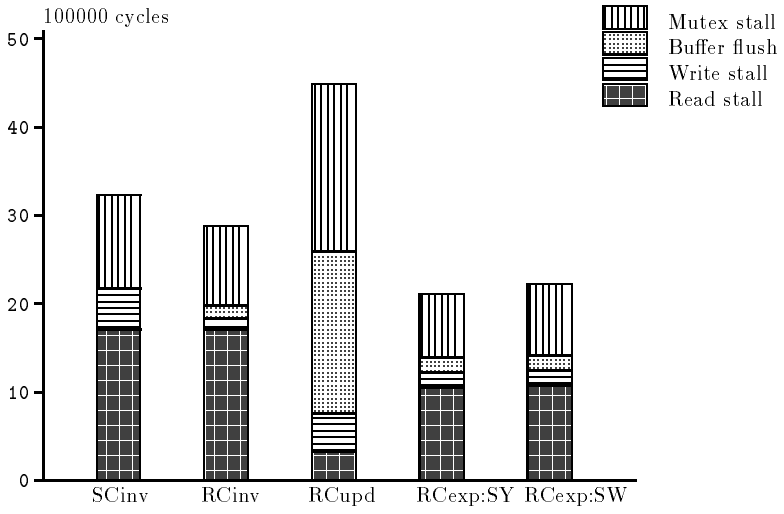

Figure 3: Barnes-Hut

Figure 3 shows the relevant overheads for the four memory systems for the entire application. The vertical axis gives the time in simulated cycles. The simulated systems in the case of the Barnes-Hut application do not use a

Message type	SCinv	RCinv	RCupd	RCexp:SY	RCexp:SW
Read request	18226	18182	351	9579	11022
Write request	1596	1595	4425	1662	1618
Invalidate	27439	26968	0	21963	21988
Update	0	0	160979	9026	7549

Table 2: Barnes-Hut: number of messages sent by a processor

write-merge-buffer.

Comparing SCinv with RCinv, we see that the reduction in write stalls in RCinv helps in reducing the overall overheads despite the buffer flush time. With RCupd the sharer set continually increases over time. This combined with the frequent synchronizations in the make-tree phase results in a large write stall time and even larger buffer flush time. For RCexp memory system, we use SEL_WRITE for writes to shared objects during the producer-phase, and SYNC_WRITE during locked accesses in make-tree phase (bar labeled RCexp:SY in Figure 3). Despite the non-deterministic nature of the sharing pattern of the application as a whole, RCexp:SY significantly reduces the read stall time compared to invalidate protocols, and also reduces the write stall and buffer flush times compared to RCupd. The bar labeled RCexp:SW only uses SEL_WRITE and no SYNC_WRITEs. Its performance is almost the same as RCexp:SY, but for a small decrease in read stall time in the latter due to SYNC_WRITEs. Table 2 compares the resulting message traffic for the different memory systems.

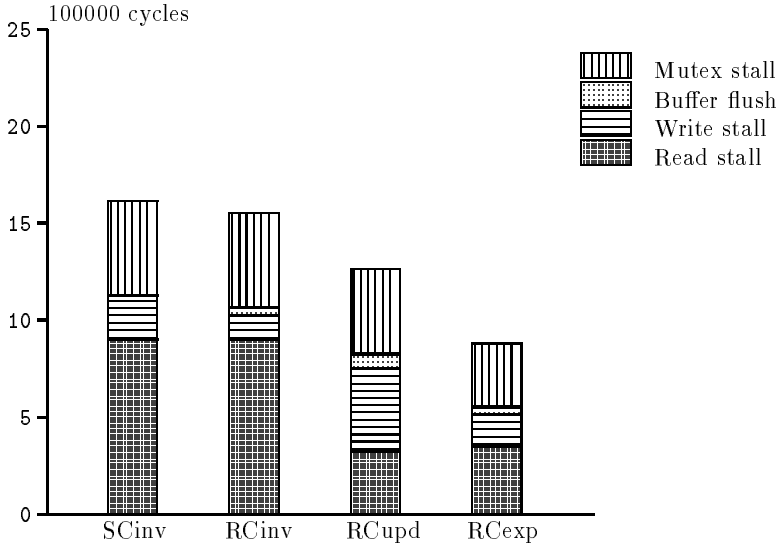

Figure 4: Barnes-Hut (producer-consumer)

Figure 4 shows the relevant overheads for the four memory systems by only considering the producer-consumer sharing pattern. RCupd does much better in this case since the protocol is able to track the producer-consumer relationship more closely without the interference of the make-tree phase. But RCexp (only uses SEL_WRITE) performs better than RCupd since our SEL_WRITE primitive adapts to the changing sharer set. The fact that the read stall times for the two are almost the same indicates that our system tracks the true sharing pattern of the application. Further, the elimination of unnecessary updates (see Table 3) compared to RCupd results in lower write stall and buffer flush times. In fact the write stall time is fairly close to that observed for invalidation protocols.

Message type	SCinv	RCinv	RCupd	RCexp
Read request	9255	9262	369	1981
Write request	831	834	1488	837
Invalidate	8450	8454	0	3538
Update	0	0	62757	7680

Table 3: Barnes-Hut (producer-consumer): number of messages sent by a processor

6.2 IS

Integer Sort is a kernel that occurs in Numerical Aerodynamic Simulation applications and is part of the NAS benchmark suite [4]. The kernel uses a parallel bucket sort to rank a list of integers. The parallel implementation that we study has been described in [31]. The problem size is 32K and the bucket size is 1K. It should be noted that this implementation has very good scalability on KSR-2 for large problem sizes (1 M). For our simulation we used a smaller problem size (which amplifies the overheads relative to the overall execution time) to keep the simulation time low. However, this does not affect the generality of our results since we are only interested in showing the relative merits of the memory systems.

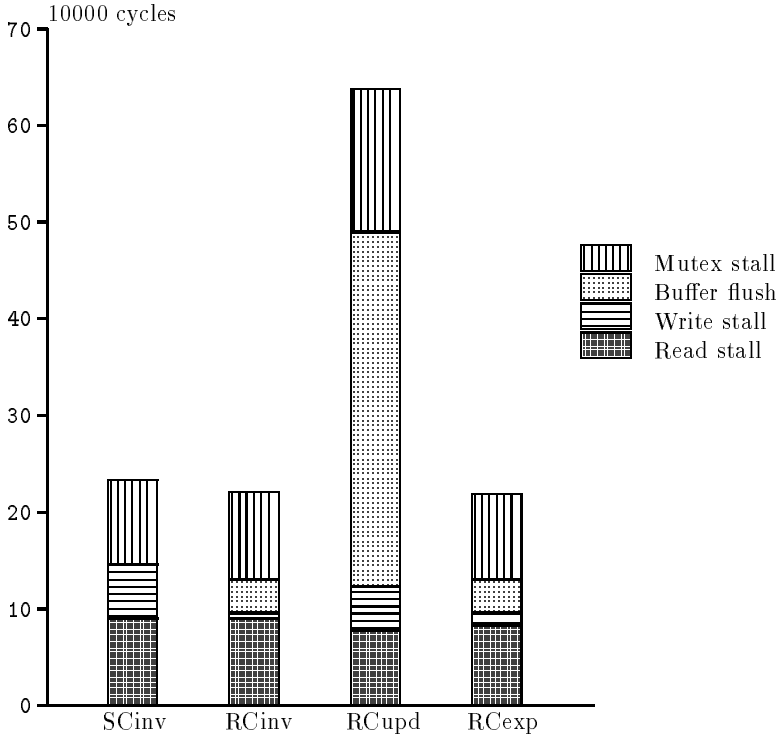

Figure 5: IS

Figure 5 shows the relevant overheads for the four memory systems for IS. The implementation goes through six phases, each separated by a barrier synchronization. The sharing pattern changes across phases. It exhibits a static communication pattern in which data use is migratory across phases. Thus RCupd would progressively send updates to more number of processors than is necessary. However, the data produced by one processor in a given phase is needed by a another processor in the next phase. Therefore a invalidation protocol would incur higher read stalls. In

Message type	SCinv	RCinv	RCupd	RCexp
Read request	503	503	503	375
Write request	507	507	3962	441
Invalidate	360	360	0	412
Update	0	0	10661	60

Table 4: IS (16 processors, 32K data points): number of messages sent by a processor

RCexp we use a combination of PSET_WRITE, SYNC_WRITE and READ_STREAM primitives.

RCinv does not perform significantly better than SCinv because any saving in the write stall time is lost in the buffer flush time. As expected the read stall time is lesser for RCupd compared to RCinv. However, the dominant overhead for RCupd is the buffer flush time which makes RCupd worse overall compared to both SCinv and RCinv. We notice that RCexp performs equally well and marginally better than RCinv. In particular we see that the buffer flush and write stall time for RCexp is close to RCinv, and the read stall time is only slightly worse than RCupd. The use of PSET_WRITE in RCexp does prove to be effective in reducing the read stall time, but due to the data dependent nature of the writes, all the PSET_WRITEs have to be issued simultaneously at the end of the phase. This fills up the write buffer rather quickly because there is no computation overlap with this explicit communication and hence results in increased write stall times. Table 4 gives the message counts for the different memory systems.

6.3 CHOLESKY

This application performs a Cholesky factorization of a sparse positive definite matrix. The sparse nature of the input matrix results in an algorithm with a data dependent dynamic access pattern. The algorithm requires an initial symbolic factorization of the input matrix which is done sequentially because it requires only a small fraction of the total compute time. Only numerical factorization [34] is parallelized and analyzed. Sets of columns having similar non-zero structure are combined into supernodes at the end of symbolic factorization. Processors get tasks from a central task queue. Each supernode is a potential task which is used to modify subsequent supernodes. A *modifications_due* counter is maintained with each supernode. Thus each task involves fetching the associated supernode, modifying it and using it to modify other supernodes, thereby decreasing the *modifications_due* counters of supernodes. Communication is involved in fetching all the required columns to the processor working on a given task. When the counter for a supernode reaches 0, it is added to the task queue. Synchronization occurs in locking the task queue when fetching or adding tasks, and locking columns when they are being modified.

The dynamic access pattern of the application precludes the use of an explicit static primitive such as PSET_WRITE. Further, an earlier scalability study of this application [35] revealed that there is little lock contention, suggesting that we may not hope to gain significantly by the use of SYNC_WRITE. Experiments also revealed that prefetch mechanisms are not very useful for this application since there is not a sufficient time duration between the time the prefetch primitive can be issued and the point where the data is actually used. Hence, in the RCexp system we use the SEL_WRITE primitive to track the sharing pattern in the application and propagate updates to a restricted set of processors. Owing to the dynamics of the execution, it is not clear to the compiler or the programmer as to which of the “writes" should be SEL_WRITEs. Consequently, we use the SEL_WRITE for every “write" to a shared variable in this application.

Figure 6 shows the overheads for the four memory systems. As pointed out earlier, RCinv does better than SCinv owing to lower write stall time. RCupd incurs lower read stall time than RCinv, but the total execution time is larger

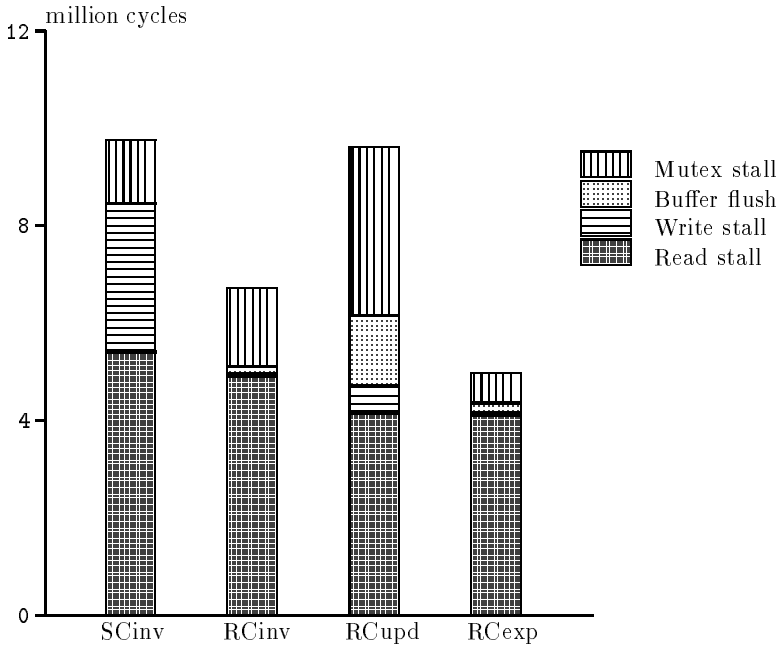

Figure 6: CHOLESKY

Message type	SCinv	RCinv	RCupd	RCexp
Read request	22084	20454	13084	18453
Write request	19347	17749	20918	26318
Invalidate	3336	3289	0	1753
Update	0	0	20794	14383

Table 5: CHOLESKY (16 proc, 1806x1806 matrix): number of messages sent by a processor

owing to increased write stall and buffer flush times. The update traffic (see Table 5) tends to increase network contention even resulting in higher mutex stall time. Figure 6 clearly shows that the RCexp system reaps the benefit of the lower read stall time of RCupd and the lower write stall and buffer flush times of RCinv. The lower update (compared to RCupd) and invalidate (compared to RCinv) traffic (see Table 5) in RCexp also tends to bring down the mutex stall time.

6.4 Maxflow

The Maxflow application finds the maximum flow from the distinguished source to the sink, in a directed graph with edge capacities. In the implementation [3], each processor accesses a local work queue for tasks to perform. These may in turn generate new tasks which are added to this local work queue. Each task involves read and write accesses to shared data. The local queues of all processors interact via a global queue for load balancing. Thus the producer-consumer relationship for data in this application is quite dynamic and random. The amount of computation required in the application is small and most of the time is spent in data movement. The problem size we consider has 200 nodes and 800 edges.

Given the dynamic data sharing nature of the application, it is difficult to specifically use an explicit communication primitive. We change all writes to shared variables to SEL_WRITE in order to track any sharing pattern that gets

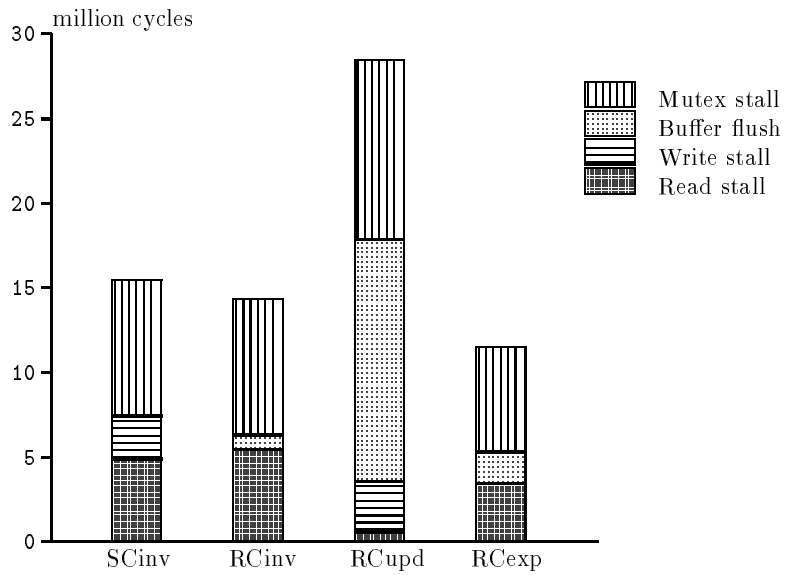

Figure 7: Maxflow

established. The experiments (see Figure 7) actually confirm that the read stall time for RCexp is noticeably lesser than that for RCinv. RCexp fails to capture the sharing patterns in all cases thus RCupd has lower read stall times. However, RCupd pays considerably more for buffer flush and mutex owing to contention on the network arising from the increased traffic of the update messages (see Table 6). We can observe that the reduction in traffic by using RCexp

Message type	SCinv	RCinv	RCupd	RCexp
Read request	21457	21946	1336	14648
Write request	13973	14319	28318	24725
Invalidate	89304	77134	0	10798
Update	0	0	678978	443335

Table 6: Maxflow (16 proc, 200 nodes, 800 edges): number of messages sent by a processor

is useful to lower the buffer flush and mutex stall significantly. The write stall times in RCexp are insignificant as in the RCinv system.

Table 7 highlights the communication overheads for the applications discussed above. The overhead for each of the memory systems is a sum of the various components (namely read stall time, write stall, buffer flush and synchronization stall). We observe a reduction in communication overhead ranging from 2% (for IS) to 26% (for Cholesky). This reduction translates to a saving in overall execution in all cases except IS. The reduction in the execution times varies from 2.8% (for Barnes-Hut) to 17.9% (for Maxflow). For IS, there is virtually no difference in the execution times of RCexp compared to RCinv.

From our initial experiments, we observe that the use of explicit communication primitives integrated into the coherence mechanism brings us closer to the goal of realizing a zero overhead machine. Looking at it from an applications perspective such a goal indicates a need for hardware mechanisms that are general enough and which can be invoked in different ways based on application needs and characteristics. Towards this end we have suggested a couple of mechanisms and indicated how the same mechanism can be used by different primitives from the application

Application	RCinv	RCupd	RCexp	Primitives	% Reduction (RCexp relative to RCinv)
Cholesky	6.80	9.70	5.03	sel-write	26.0
IS	0.23	0.64	0.22	rstream pset-write	2.1
Maxflow	14.58	28.63	11.81	sel-write	19.0
Nbody	4.97	7.07	4.03	sync-write sel-write	18.9

Table 7: Communication Overheads (in millions of cycles)

viewpoint without additional programming complexity. On the other hand, the choice of a single protocol may not be clear based on the application. Further an application might need to switch between protocols as its data sharing and access patterns change. These issues need further investigation and are part of our ongoing work.

7 Related Work

There are many related projects that have similar goals to ours. One commercial product, the Cray-T3D [8] provides shared memory but does not maintain cache coherence. It is totally up to the system software to ensure coherence of cached data. In terms of primitives for explicit communication, they offer mechanisms to access the remote memory, block transfer, and prefetch. Another shared memory multiprocessor, the KSR-2 [33], uses a hardware write-invalidate protocol to maintain sequential consistency. The KSR-2 provides prefetch and poststore primitives to overlap computation with communication. However, they are not as general as the mechanisms we propose. For instance, in KSR-2 there is no way to issue a prefetch to more than one cache line in one request. Similarly, there is no guarantee in KSR-2 that a poststore will result in a data item being placed in a specific remote cache.

The approach used in the Tempest project [32] from University of Wisconsin comes closest to our approach. They propose an interface for developing either shared memory or message passing programs. The interface is fairly general and is independent of the underlying hardware. For shared memory programming, the interface allows realizing a variety of coherence protocols. However, the specific mechanisms that they provide appear to facilitate the invalidation style coherence protocol. Further, from the point of view of software directed management of caches, these mechanisms are at a fairly low level. On the other hand, the primitives we propose are at a higher level with the specific intent of enabling applications to specify their coherence requirements commensurate with their needs. It should be pointed out that we can implement our primitives on top of the Tempest interface.

The Stanford Flash project [23] implements a shared memory system using a separate protocol processor. However, the primary goal is to reduce the hardware complexity of maintaining the directory structure. Therefore, it implements the traditional directory-based cache coherence protocol in software. Of course, since the protocol engine is now implemented in software, it can be used to implement any memory system, including ours.

8 Concluding Remarks

While shared memory parallel programming is appealing, it is generally recognized that achieving good performance in shared memory machines requires considerable effort. The implicit communication in terms of shared memory accesses can result in considerable communication overheads. Managing the caches through the software can give more control over when and how coherence actions are performed. In this work we have proposed a set of explicit communication mechanisms to augment the normal cache coherence protocol that handles implicit communication. These mechanisms are integrated with the underlying cache coherence management. The intent of these mechanisms is to enable tolerating and minimizing the communication overheads. The mechanisms are intended to be inserted automatically by the compiler using application hints coupled with synchronization information in the application. The use of these primitives requires very little change in the application programs, and involves marginal increase in hardware complexity. When these primitives are used in conjunction with an invalidation protocol, we show that we can achieve read stall times that are comparable to update protocols while keeping the write stall and buffer flush times close to invalidation based protocols used for implementing weaker memory models.

There are several issues that require closer scrutiny. We would like to explore other ways of tracking and changing the sharing pattern. We would also like to compare our results with competitive update protocol which is another heuristic to track changes in the sharing pattern. We expect that our primitives should perform better than competitive update because these can be more flexibly applied. Our primitives also come at with a lower hardware cost compared to competitive update schemes. Further, we assumed infinite caches to eliminate the effects of capacity misses and study only the impact of our primitives on true sharing misses. We believe our primitives would have even more impact with limited size caches and studying these effects are also part of our ongoing research efforts.

References

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

[2] A. Agarwal et al. The MIT Alewife Machine: Architecture and Performance. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 2–13, June 1995.

[3] R. J. Anderson and J. C. Setubal. On the parallel implementation of goldberg's maximum flow algor ithm. In *4th Annual ACM Symposium on Parallel Algorithms and Archite ctures*, pages 168–77, June 1992.

[4] D. Bailey et al. The NAS Parallel Benchmarks. *International Journal of Supercomputer Applications*, 5(3):63–73, 1991.

[5] P. Bitar and A. M. Despain. Multiprocessor cache synchronization: Issues, innovations, evolution. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 424–433, June 1986.

[6] H. Cheong and A. V. Veidenbaum. A cache coherence scheme with fast selective invalidation. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 299–307, June 1988.

[7] H. Cheong and A. V. Veidenbaum. Stale data detection and coherence enforcement using flow analysis. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages I: 138–145, August 1988.

[8] Cray Research, Inc., Minnesota. *The Cray T3D System Architecture Overview Manual*, 1993.

[9] D. E. Culler et al. Parallel Programming in Split-C. In *Proceedings of Supercomputing '93*, pages 262–273, November 1993.

[10] R. Cytron, S. Marlovsky, and K. P. McAuliffe. Automatic management of programmable caches. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages II–229–238, August 1988.

[11] F. Dahlgren, M. Dubois, and P. Stenstrom. Combined performance gains of simple cache protocol extensions. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 187–197, April 1994.

[12] F. Dahlgren and P. Stenstrom. Using write caches to improve performance of cache coherence protocols in shared memory multiprocessors. Technical report, Dept. of Comp. Eng., Lund Univ., April 1993.

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

[14] Encore Computer Corporation, 257 Cedar Hill St., Marlboro, MA 01752. *Multimax Technical Summary*, 1986.

[15] B. Falsafi et al. Application-Specific Protocols for User-Level Shared Memory. In *Proceedings of Supercomputing '94*, November 1994.

[16] Matthew I. Frank and Mary K. Vernon. A hybrid Shared Memory/Message Passing parallel machine. In *Proceedings of the 1993 International Conference on Parallel Processing*, August 1993.

[17] K. Gharachorloo, D. Lenoski, J. Laudon, P.Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, 1990.

[18] J. R. Goodman and P. J. Woest. The Wisconsin Multicube: A new large-scale cache-coherent multiprocessor. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 422–431, June 1988.

[19] A. Gupta, J. Hennessy, K. Gharachorloo, T. Mowry, and W-D. Weber. Comparative evaluation of latency reducing and tolerating techniques. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 254–263, May 1991.

[20] J. Heinlein, K. Gharachorloo, S. A. Dresser, and A. Gupta. Integration of Message Passing and Shared Memory in the Stanford FLASH Multiprocessor. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1994.

[21] M. Heinrich et al. The performance impact of flexibility in the Stanford FLASH Multiprocessor. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1994.

[22] D. Kranz, K. Johnson, A. Agarwal, J. Kubiatowicz, and B-H Lim. Integrating message-passing and shared memory: Early experience. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 54–63, May 1993.

[23] J. Kuskin et al. The Stanford FLASH multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 302–313, April 1994.

[24] L. Lamport. How to make a Multiprocessor Computer that Correctly executes Multiprocess Programs. *IEEE Transactions on Computer Systems*, C-28(9), 1979.

[25] J. Lee and U. Ramachandran. Synchronization with Multiprocessor Caches. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 27–37, 1990.

[26] J. Lee and U. Ramachandran. Architectural Primitives for a Scalable Shared Memory Multiprocessor. In *Proceedings of the Third Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 103–114, Hilton Head, South Carolina, July 1991.

[27] D. Lenoski, J. Laudon, K. Gharachorloo, W-D Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam. The Stanford DASH multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[28] T. Lovett and S. Thakkar. The symmetry multiprocessor system. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 303–310, August 1988.

[29] S. L. Min and J-L. Baer. A Timestamp-based Cache Coherence Scheme. In *Proceedings of the 1989 International Conference on Parallel Processing*, pages I: 23–32, August 1989.

[30] H. Nilsson, P. Stenstrom, and M. Dubois. Implementation and evaluation of update-based cache protocols under relaxed memory consistency models. Technical report, Dept. of Comp. Eng., Lund Univ., July 1993.

[31] U. Ramachandran, G. Shah, S. Ravikumar, and J. Muthukumarasamy. Scalability study of the KSR-1. In *Proceedings of the 1993 International Conference on Parallel Processing*, pages I–237–240, August 1993.

[32] S. K. Reinhardt, J. R. Larus, and D. A. Wood. Tempest and Typhoon: User Level Shared Memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 325–336, April 1994.

[33] Kendall Square Research. Technical summary, 1992.

[34] J. P. Singh, W-D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. Technical Report CSL-TR-91-469, Computer Systems Laboratory, Stanford University, 1991.

[35] A. Sivasubramaniam, A. Singla, U. Ramachandran, and H. Venkateswaran. An Approach to Scalability Study of Shared Memory Parallel Systems. In *Proceedings of the ACM SIGMETRICS 1994 Conference on Measurement and Modeling of Computer Systems*, pages 171–180, May 1994.

[36] A. Sivasubramaniam, A. Singla, U. Ramachandran, and H. Venkateswaran. A Simulation-based Scalability Study of Parallel Systems. *Journal of Parallel and Distributed Computing*, 22(3):411–426, September 1994.

[37] C. P. Thacker and L. C. Stewart. Firefly: A Multiprocessor Workstation. In *Proceedings of the First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 164–172, October 1987.

[38] W-D. Weber and A. Gupta. Exploring the benefits of multiple hardware contexts in a multiprocessor architecture: Preliminary Results. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 273–280, June 1989.

[39] S. C. Woo, J. P. Singh, and J. L. Hennessy. The performance advantages of integrating block data transfer in cache-coherent multiprocessors. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1994.

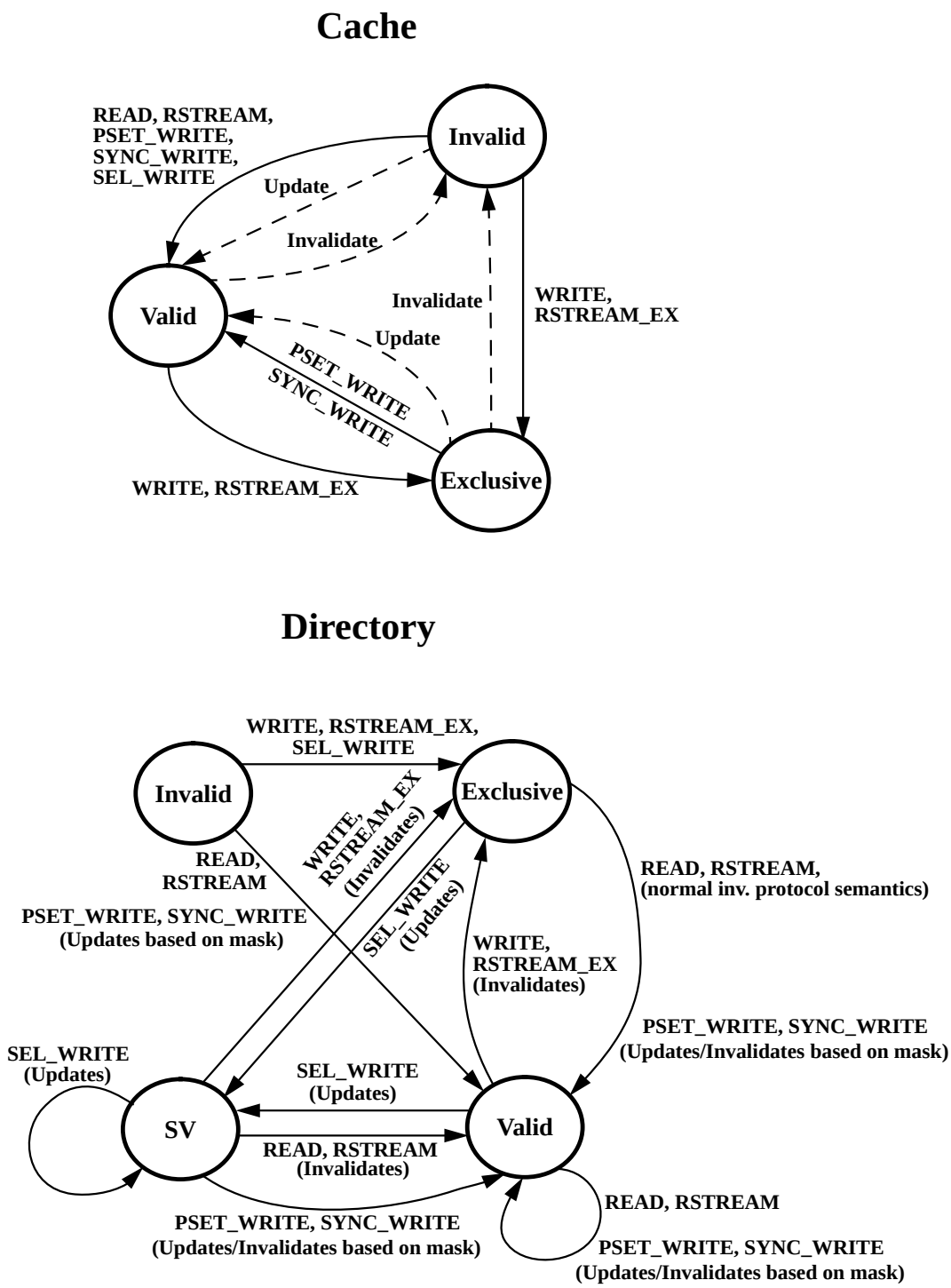

Figure 8: The Coherence Protocol of RCexp