# Compiler-Directed Selective Update Mechanisms for Software Distributed Shared Memory

Sandhya Dwarkadas, Alan L. Cox, Honghui Lu, and Willy Zwaenepoel
Rice University
Department of Computer Science
P.O. Box 1892
Houston, TX 77251

Technical Report TR95-253

e-mail: sandhya@cs.rice.edu

**Abstract**

We estimate the performance gains of a compiler-directed selective update protocol for software distributed shared memory systems. Our base system uses an invalidate-based lazy release consistent protocol. The goal is to reduce the cost of communication, by aggregating messages, by reducing the effects of false sharing, and by eliminating the communication of unnecessary data. We achieve our goal through the use of compiler analysis for prefetching and for further relaxing the consistency model. Well-known compiler techniques, such as regular section analysis, can be adapted to determine data access patterns between synchronization points and supply the necessary information to the runtime system.

We propose three selective update techniques - *cross-synchronization prefetching, partial update, and consistency elimination*, and describe the compiler and runtime support necessary. *Cross-synchronization prefetching* updates requested data at the time of synchronization without changing the consistency model. *Partial update* relaxes the consistency model in addition to prefetching the requested data. Data not requested is not kept consistent, reducing both consistency maintenance overhead and the effects of false sharing. *Consistency elimination* relaxes the consistency model by ensuring that data is *not* updated when accessed. This technique deals with write-first data access patterns, where data is written before it is read, between synchronization operations.

We augmented the base runtime system to support the selective update protocol and hand-modified programs to take advantage of it. Our experimental platform is a 100Mbps ATM network connecting 8 DECStation-5000/240s running Ultrix. The compiler-directed selective update protocol improves performance for all the programs examined when compared to either an invalidate or a runtime hybrid protocol, with a 16% improvement for Integer Sort, 11% for Water, 35% for FFT, and a 10% improvement for SOR when compared to the invalidate protocol. Performance is equivalent to the hand-tuned message passing versions of the program in some cases.

## 1 Introduction

Parallel processing on workstation clusters is growing in popularity as the performance of general-purpose networks and microprocessors improves. A major obstacle to continued growth is the difficulty of programming in the message-passing paradigm that is native to the hardware. As one way to overcome this obstacle, researchers have developed software distributed shared memory (DSM) systems that enable processes on different workstations to share memory [20, 9, 7, 6, 15, 27, 16]. Despite the recent improvements in memory consistency algorithms [15, 27], hand-tuned message-passing programs frequently outperform comparable shared memory programs by a significant margin. We believe that compilation techniques developed for data parallel languages, such as regular section analysis [4, 3], can be used to reduce this gap for explicitly parallel programs. In this paper, we first describe how a compiler can help the DSM system to improve performance. In particular, we describe how compile-time analysis can drive a *selective update protocol* that is designed to reduce the amount of communication. We then measure the performance of several hand-compiled programs against two run-time protocols and message-passing versions of the same programs.

The first software DSM system, Ivy [20], provided a sequentially consistent memory, the same as hardware shared-memory multiprocessors of that period. Unfortunately, sequential consistency requires a total ordering of all shared data accesses, sometimes resulting in communication at every write to shared memory. To reduce the amount of communication, state-of-the-art software DSM systems now use relaxed memory models [15, 27]. For example, the release consistency model [10] permits several implementation techniques for reducing the amount of communication without significantly changing the programming model from sequential consistency. Two such techniques are lazy release consistency [14] and the multiple-writer protocol [6]. Lazy release consistency (LRC) delays the communication of invalidations or updates to shared memory until a synchronization point in the program. In contrast to earlier techniques, such as eager release consistency [6], LRC doesn't require a broadcast, only the synchronizing processors communicate. The multiple-writer protocol [6] allows several processes to simultaneously modify different parts of a page without having the page "ping-pong" between the processors.

Despite these techniques, certain shared-memory programs don't perform as well as their message-passing counterparts. We have found several reasons for this. On the one hand, the performance of shared-memory programs sometimes suffers from unnecessary communication that occurs because of false sharing and the reuse of memory. Although the multiple-writer protocol eliminates the ping-pong effect, falsely shared data is still communicated at synchronization points. Multiple readers of disjoint data in the same page will also incur false sharing overhead by bringing in unnecessary data. In addition, when memory is logically freed and reallocated by the program, the DSM system doesn't know that the next processor to use this memory doesn't have to request any updates. On the other hand, message-passing programs sometimes benefit from the combination of synchronization and data transfer as well as the aggregation of data into a smaller number of large messages. In this paper, we describe how to use compile-time analysis to reduce the unnecessary communication of data and to gain some of the advantages of the message-passing programs. The compile-time analysis is aimed at improving performance without complicating the user's programming model. In fact, the shared address space provided by the DSM system reduces the complexity of the compile-time analysis required to determine data access patterns, since local to global name space translation isn't necessary.

In our approach, the compile-time analysis is used to direct the DSM system to selectively update data. In the absence of any compiler directives, an invalidate-based protocol is used. We propose three selective update techniques - *cross-synchronization prefetching, partial update, and consistency elimination. Cross-synchronization prefetching* updates requested data at the time of synchronization without changing the consistency model. These directives are applied when the compiler can determine that certain pages are likely to be accessed after a synchronization operation completes. This technique reduces the number of messages by combining data transfer with synchronization. This technique is especially suitable for a migratory sharing pattern. *Partial update* relaxes the consistency model in addition to prefetching the requested data. Data not requested is not kept consistent, reducing both consistency maintenance overhead and the effects of false sharing. Hence, the compiler needs to be able to determine exactly what data will be accessed in an entire phase of a program. In addition, if the compiler can determine the producer of the data, then the synchronization can be replaced by a direct message exchange between the producer and the consumer. This technique implements a producer-consumer style interaction in an efficient way. *Consistency elimination* relaxes the consistency model by ensuring that data is *not* updated when accessed. This technique deals with write-first data accesses, where data is written before it is read, that occur between synchronization operations. This behavior is prevalent in programs that reuse data structures between phases in the program. We eliminate communication by avoiding updates to the write-first data.

In order to determine the performance gains, we modified four programs by hand to simulate the compiler directives to the selective update protocol - Integer Sort, M-Water, SOR, and 3-D FFT. We compare the performance of the selective update protocol to that of an invalidate-based protocol, a hybrid invalidate/update-based protocol that only uses run-time information, and hand-tuned message-passing versions of the four programs. The invalidate-based protocol has the advantage that it communicates data only when necessary. However, it has the disadvantage that it suffers from the most access misses. The run-time hybrid invalidate/update-based protocol [8] is intended to reduce communication by aggregating data communication and combining it with synchronization. In this protocol, data is piggy-backed on the message releasing a synchronization object when the releaser believes that the acquiring processor caches that data. However, this protocol is limited by its ability to predict future accesses based upon past accesses. As a result, it may ship data that is never accessed.

Our experimental platform is a 100Mbps ATM network of 8 DECstation-5000/240s running Ultrix. The compiler-directed selective update protocol improves performance for all the programs presented when compared to either the invalidate or the hybrid protocol, with a 16% improvement for Integer Sort, 11% for Water, 35% for FFT, and a 10% improvement for SOR when compared to the invalidate protocol. For SOR, performance is equivalent to the hand-tuned message passing version, and performance is competitive with the message passing versions for all the other programs as well. The performance is consistently better than the invalidate protocol. This is in contrast to the run-time hybrid protocol, which sometimes performs worse.

The rest of the paper is organized as follows. Section 2 presents some background about the base invalidate-based system. Section 3 describes our selective update mechanisms and our proposed compile-time support. Section 4 compares the performance of four applications under the various protocols. Section 5 describes related work. Section 6 presents our conclusions.

# 2 Background

Distributed shared memory (DSM) enables processes on different machines to share memory, even though the machines physically do not share memory (see Figure 1). The shared memory approach is attractive since most programmers find it easier to use than a message passing paradigm, which requires them to explicitly partition data and manage communication. For programs with complex data structures and sophisticated parallelization strategies, we believe this to be a major advantage. Using explicitly parallel programs allows the exploitation of application-specific information not available to a compiler. Providing compiler support allows taking advantage of current compiler technology to reduce communication. This section describes the techniques used to achieve good performance for shared memory programs in the TreadMarks system, which we will use as the baseline runtime system in our work.

## 2.1 Lazy Release Consistency

The provision of memory *consistency* is at the heart of a DSM system: the DSM software must move data among the processors in a manner that provides the illusion of globally shared memory. The consistency model seen by an application determines when modifications to data may be expected to be seen at a given processor.

Traditionally, *sequential consistency* [19] has been the model provided to users of shared memory

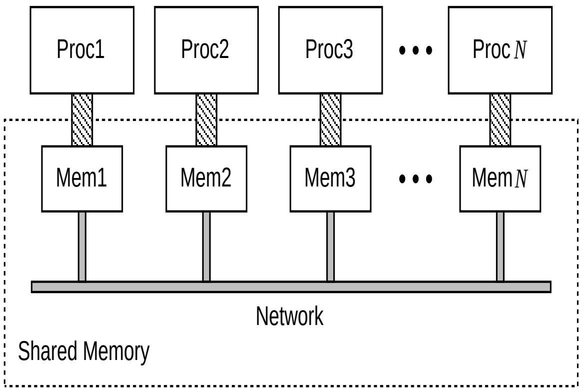


**Figure 1** Distributed Shared Memory

machines. Sequential consistency, in essence, requires that writes to shared memory are visible "immediately" at all processors. The IVY system [20] implemented this model. The disadvantage of this model is that it can result in an excessively large amount of communication since modifications to shared data must be sent out immediately in the form of invalidates or updates.

*Release consistency* (RC) [10] is a relaxed memory consistency model that addresses the above-mentioned efficiency problems of sequential consistency. In RC, synchronization operations are made explicit and categorized into *acquires* (corresponding to getting access to data) and *releases* (corresponding to providing access to data). RC allows the effects of non-synchronizing shared memory accesses to be delayed until a subsequent *release* synchronization by the same processor is performed. This optimization is based on the assumption that any other process will expect to see the *release* before looking for previously modified data. RC, however, requires that the non-synchronizing shared memory accesses be visible at all processors before the corresponding release can perform at any processor. This potentially implies communication with every processor at a synchronization point.

TreadMarks uses *lazy release consistency* (LRC) [14], which is a lazy implementation of the release consistency model. The LRC algorithm delays the propagation of data modifications to a processor until that processor executes an *acquire*. This optimization is based on the knowledge that any other process will perform an *acquire* before accessing data modified before the last *release*.

At *acquire* time, the last releaser piggybacks a set of *write notices* on the lock grant message sent to the acquirer. These write notices describe the shared data modifications that precede the *release*. Communication occurs only between the synchronizing processes, and transmission of unnecessary consistency information and data is much reduced. The acquiring processor can then invalidate or update the pages for which the incoming write notices represent modifications, depending on the protocol and implementation.

## 2.2 A Multiple-Writer Invalidate Protocol

The base protocol we use in our system is invalidation. At the time of an acquire, the acquiring processor determines the pages for which the incoming write notices contain more recent modifications than those already applied to the processor's copy of the page. These pages are then invalidated by protecting access to them.

On an access fault, a page is validated by sending out a request message to bring in the necessary modifications to the local copy in the form of *diffs*. A diff is a run-length encoding of the changes made by a processor to a single virtual memory page. The use of diffs avoids the necessity of sending entire pages on every access miss and also allows multiple writers to access a page at the same time. Figure 2 provides an example of how the protocol works. At each acquire, invalidations are piggybacked on the reply message for all pages modified "preceding" the acquire (The term "preceding" is to be interpreted here as preceding in the happens-before-1 partial order [1]). When an invalidated page is later accessed, a page fault is generated and the modifications to the page are retrieved in the form of diffs.

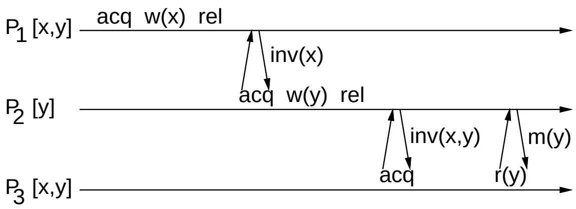


**Figure 2** Lazy Invalidate

### 2.3 An Alternative Runtime Hybrid Protocol

A runtime approach to address the problem of access miss latency is the *hybrid* invalidate/update protocol described in Dwarkadas et al. [8]. At a synchronization operation, updates are sent for pages for which the acquirer is in the releaser's approximate copyset and for which the releaser has all the necessary diffs present. Invalidations are sent for the remaining modifications. The copyset is the releaser's knowledge of the processors that cache the particular page. The copyset is modified during any interaction between the processors that indicates that the page has either been requested or invalidated at a processor. Hence, the copyset is only an approximation of the list of processors caching the page at any given time. All the updates can be piggybacked on the lock grant message, thereby reducing the number of messages exchanged.

While the runtime system can optimize communication by keeping track of past access patterns, it cannot predict changes in future access patterns, potentially resulting in large amounts of excess data being brought in during phases of the program with different access patterns. We will compare the performance of our compiler-directed approach to this runtime hybrid protocol in Section 4.

## 3 Compiler-Directed Selective Update Protocol

One of the drawbacks of a pure runtime approach to providing an efficient shared memory environment is that communication must necessarily be demand-driven. In addition, the runtime system must be conservative in determining the amount of consistency information and data transferred between processors. Our goal is to explore how compiler analysis can be used to predict accesses and how that information can be fed to the runtime system to either aggregate communication, reduce the amount of communication, reduce consistency maintenance overhead, or to overlap communication and computation. The rest of this section presents three examples of commonly occurring access patterns for which compile-time prediction can improve accuracy. The runtime system can then take advantage of this information to *selectively update* data, rather than using invalidation that would result in a page fault on each access miss. These optimizations are by no means exhaustive, and as this work progresses, we expect to find other possible optimizations.

### 3.1 Cross-Synchronization Prefetching

Cross-synchronization prefetching takes advantage of compiler analysis by using it to provide an estimate of data accesses made after a synchronization point. This information is used by the runtime system to bring in the data at the synchronization point. This optimization is especially suited to migratory data access patterns. Both synchronization by locks or by barriers can be handled by this method. For simplicity, the method is explained here for locks.

The directives inserted by the compiler for cross-synchronization prefetching consist of a list of starting addresses, the size of data accessed starting at the address, and the order in which data is traversed. This information is then converted by the runtime system into a list of pages. At the time of an acquire, these page numbers are piggybacked on the lock acquire message, along with information specifying the latest updates seen by the acquiring processor for these pages. The latter information is used to determine whether updates need to be sent to the acquirer for the requested pages.

When the releaser sends out the message allowing the acquire request to succeed, it piggy-backs the modifications for the requested pages, and sends invalidation notices for the rest in order to keep memory consistent (see Figure 3). If the releaser does not have an up-to-date copy of any data item requested, it invalidates the acquirer's copy as well. Currently, the runtime system brings in this data at the time of an access miss. Alternatively, the data could be prefetched prior to the access miss. While this might reduce access latencies, it does result in some additional messages to send out the prefetch requests.

The goals of cross-synchronization prefetching are to combine synchronization and data transfer, avoid access misses at the acquiring processor, and to take advantage of bulk data transfer in sending the data from the releaser to the acquirer. In the case of large prefetch requests, the data size requested may be larger than the size of a single message in the underlying communication protocol. In this situation, the acquirer is not required to wait for all the update messages, but continues computation, stalling only if it accesses a page that has not yet been updated. Using the traversal order provided by the compiler analysis to determine the order in which updates are sent out greatly reduces the probability of stalling for an update. This asynchronous prefetching

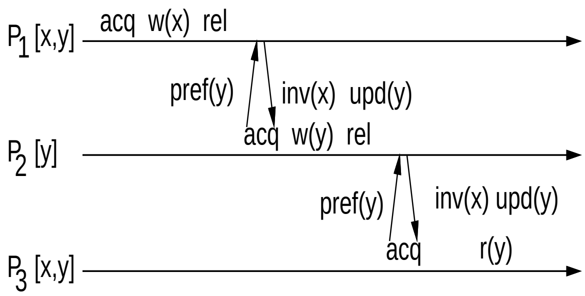


**Figure 3** Cross-Synchronization Prefetching

reduces the potential problem of larger acquire latencies due to data shipping at the time of an acquire, while still taking advantage of the benefits of aggregating data communication, especially on a network of workstations. In addition, computation and communication are overlapped.

Midway [27] attempts to provide a similar functionality to cross-synchronization prefetching at the expense of additional user complexity. Clouds [7] is a distributed system that uses objects to achieve a similar purpose. The disadvantage of these approaches is that data must be explicitly associated by the program with the synchronization variable or object in question, and additional communication is required to change this association. Our approach allows the association of different data items with a lock at each instantiation.

**Compiler Support.** Regular section analysis is used in compilers to determine array accesses in a loop nest [3, 4]. Regular section descriptors [4] provide compact representations of data accesses made, and can be used to efficiently supply the required access information to the runtime system. In the case of large prefetch requests, data access descriptors [3], which describe the order in which the accesses are made, may be used to determine the order in which the update messages are sent out. Pointers may also be handled in a restricted manner by supplying the address of the pointer to the runtime system so that the runtime system can then extract the base address of the section accessed. Scalars are normally handled by enumeration.

The regular section analysis will be applied to collect information about the accesses made by a process between synchronization points. Figure 4 provides an example section of code on which regular section analysis can be applied. This code is part of a parallel bucket sort algorithm. The shared array is divided into as many sections as there are processors. Each section is controlled by a per-processor lock. The processors access the sections in a staggered manner. A prefetch directive as shown can be added to the code by the compiler or user. The prefetch is actually performed in response to the lock acquire. `FORWARD` indicates the direction of data traversal.

If, for example, *shared_array* was a shared pointer whose value could change during execution, the compiler could provide the runtime system with the address of *shared_array* in the prefetch command. The releaser of the lock could then use this indirection to determine the data to be sent to the acquirer.

Using this form of compiler analysis on explicitly parallel shared memory programs broadens the range of programs that can take advantage of regular section analysis. Programs that cannot be expressed in the traditional HPF framework and are not easily automatically parallelizable can take advantage of current compiler technology on sections of straight-line code.

### 3.2 Partial Update

The partial update technique provides the ability to limit the portions of shared memory that are kept consistent between processors. This allows the update of only parts of a single page (the unit of consistency). This optimization applies to programs where phases are separated by barriers.

Consider, for example, a program with three phases, with phases 1 and 2 separated by *barrier 1* and phases 2 and 3 separated by *barrier 2*. The partial update technique is applicable when the

```
shared int shared_array[MAX_INTS];

for (k = 1; k <= nof_procs; k++) {

    p = (Tmk_proc_id+k)%nof_procs;

    Tmk_prefetch(&shared_array[p*pre_proc_els], per_proc_els*sizeof(int), FORWARD);
    Tmk_lock_acquire(p);

    for (j = p*per_proc_els; j < (p+1)*per_proc_els; j++) {
        shared_array[j] += local_array[j];
        local_array[j] = shared_array[j];
    }

    Tmk_lock_release(p);

}
```

**Figure 4** Cross-Synchronization Prefetching Example

compiler can determine exactly what data is accessed by each processor between barriers 1 and 2. Given that knowledge, it suffices to make that data consistent, and leave the rest of memory, even within the same page, inconsistent. Support for partial update must be integrated with the underlying consistency mechanism to ensure a consistent view of memory for further phases in the program. The normal consistency actions at the global *barrier 2* bring the shared memory back to a consistent state.

As before, the directive emitted by the compiler consists of a set of address ranges. These are piggy-backed on the barrier synchronization messages, instead of using a list of pages. The runtime system then determines the last modifier of any piece of data, updating only those parts of the pages that will be accessed. This technique alleviates the problem of false sharing inherent to the large consistency unit in software DSM systems.

If, in addition to determining the access pattern between *barriers 1 and 2*, the compiler can also determine the producer of the data before *barrier 1*, then *barrier 1* can be replaced by a set of direct message exchanges between producer and consumer, avoiding the general-purpose barrier mechanism. This technique optimizes producer-consumer style communication, where the producer and consumer of data can be statically determined. The need for global synchronization is eliminated and synchronization and data transfer are combined.

**Compiler Support.** The above techniques are particularly applicable to phases in programs with regular access patterns that are easily analyzed by existing compiler technology, once again broadening the class of applications that can be optimized. Regular section analysis can be used to determine the exact communication requirements in the region of code between two barriers. The data access information is then supplied at the barrier. Control flow analysis can be used in conjunction with regular section analysis to determine data access intersection between any two adjacent regions. If this intersection can be determined on a per processor basis [26], the producers and consumers of each piece of data can be identified, allowing further optimization.

Figure 5 provides an example of a piece of code on which this optimization could be applied. The shared accesses in the loop are to the `DISP` field of the *molecule_type* data structure. No other field is accessed, and no other shared data is accessed. In this case, the compiler can insert directives for the runtime system similar to those in the *Tmk_prefetch_only* statement. Both the total size of data to be prefetched and the size of each element are specified. The *stride* specifies the distance between accessed elements.

For programs where exact analysis is not possible, using the underlying runtime shared memory system as opposed to conservatively bringing in accessed data using compiler analysis may result in the most efficient program.

### 3.3 Consistency Elimination

In programs that reuse data structures between phases of a program, for instance, phases separated by a barrier, it is sometimes the case that the values from the previous phase are immediately overwritten with new values, without ever being read. Since the data is not read before being written, it need not be made consistent. However, allowing normal consistency actions to proceed will result in unnecessary updates being brought in for the page. At the phases in the program

```
struct molecule_type {
    .......;
    double DISP;
    .......;
}

Tmk_prefetch_only(&VAR[per_proc_els*Tmk_proc_id].DISP,
        sizeof(double)*per_proc_els, stride, sizeof(double));
Tmk_barrier(0);
for (i = per_proc_els*Tmk_proc_id; i < per_proc_els*(Tmk_proc_id+1); i++) {
    if (VAR[mol].DISP < MAX_DISP) {

        ...............

    }
}
Tmk_barrier(0);
```

**Figure 5** Partial Update Example

when the data is determined to be write-first, data is selectively *not* updated to reduce the amount of data communicated and the number of messages necessary for the program.

Consistency elimination is done with the help of a directive to the runtime system to validate the address range in question. The runtime system then validates the pages in the address range without bringing in any modifications made to them, and consequently throws away all previous modifications. Currently, only those pages that are entirely within the specified address range are validated.

**Compiler Support.** Data flow analysis can be used to determine the location of access of any data element and the type of access made, in other words, the definition-use chain for any block of code. From this definition-use chain, the set of variables whose first access in the block is a write can be determined. This set is the union of the set of all variables used within the block that have prior definitions within the block for every use and the set of all variables defined but not used in the block. A block in our case would be the computation between any two synchronization points.

Figure 6 provides an example of how and where this technique can be applied. In this example, each processor initializes a part of the shared array in the beginning of each phase. Since this data is written and not read, it need not be brought up-to-date. The compiler can insert a directive as in the `Tmk_validate` statement, specifying the address and range of the data to avoid updating.

# 4 Performance

This section estimates the benefits achievable for various programs using the selective update mechanisms described above. Programs modified to take advantage of the mechanisms (*S_update*) are compared to an invalidate protocol (*Inval*), to the runtime hybrid protocol (*Hybrid* - see Section 2.3), and to a message passing version of the program (*Msg*).

```
p = Tmk_proc_id;
Tmk_lock_acquire(p);
Tmk_validate(&shared_array[p*pre_proc_els], per_proc_els*sizeof(int));
for (j = p*per_proc_els; j < (p+1)*per_proc_els; j++) {
    shared_array[j] = local_array[j];
}

Tmk_lock_release(p);
```

**Figure 6** Consistency Elimination Example

### 4.1 Experimental Environment

The system used to evaluate the utility of the update mechanisms consists of 8 DECstation-5000/240 workstations, each with a 40Mhz MIPS R3000 processor, a 64 Kbyte primary instruction cache, a 64 Kbyte primary data cache, and 24 Mbytes of memory. The workstations are connected to a high-speed ATM network using a Fore Systems TCA-100 network adapter card supporting communication at 100 Mbits/second. In practice, however, user-to-user bandwidth is limited to 25 Mbits/second. The ATM interface provides a high aggregate bandwidth because of the capability for simultaneous, full-speed communication between disjoint workstation pairs. The workstations run the Ultrix version 4.3 operating system. The TreadMarks DSM on which our optimizations are prototyped, is implemented as a user-level library that is linked in with the application program. No kernel modifications are necessary.

### 4.2 Results

We present results for four applications - Integer Sort and 3-D FFT from the NAS benchmark suite [2], M-Water from the SPLASH benchmark suite [24], and SOR, a locally developed program. Integer Sort uses cross-synchronization prefetching and consistency elimination. M-Water uses partial updates to avoid false sharing. SOR uses the enhanced version of the partial update technique since both the producer and the consumer of the data are known. FFT uses both partial updates and consistency elimination to avoid false sharing.

Figures 7 to 10 present the speedups obtained for each of the applications with various runtime systems. We discuss the relative performance of the different protocols and compare them to the message passing versions of the same programs in Section 4.3. Table 1 presents the communication statistics for each of the programs at 8 processors. Table 2 presents the uniprocessor execution times for each of the programs. These times are generated by running the program without linking to TreadMarks. `Messages` and `KBytes` represent the total number of messages and amount of data communicated during each timed execution. The performance of each application is described in detail in the following sections.

### 4.3 Discussion

#### 4.3.1 Integer Sort

Integer Sort (IS) is a program from the NAS benchmark suite [2]. IS requires ranking an unsorted sequence of $N$ keys. The *rank* of a key in a sequence is the index value $i$ that the key would have if the sequence of keys were sorted. All the keys are integers in the range $[0, B_{max}]$ and the method used is counting, or bucket sort. The amount of computation required for this benchmark is relatively small – linear in the size of the array $N$. The amount of communication is proportional to the size of the key range, since an array of size $B_{max}$ has to be passed around between processors. The values of $N$ and $B_{max}$ we used were $2^{20}$ and $2^{12}$ respectively, and we ran the benchmark for 10 iterations.

| Application | Protocol | Messages | KBytes |
|---|---|---|---|
| IS | Sel-Update | 1306 | 2661 |
| | Hybrid | 2106 | 10884 |
| | Invalidate | 2139 | 2633 |
| | Msg-based | 588 | 2293 |
| M-Water | Sel-Update | 5293 | 8416 |
| | Hybrid | 17564 | 31612 |
| | Invalidate | 26466 | 29041 |
| | Msg-based | 2006 | 5801 |
| SOR | Sel-Update | 5543 | 5665 |
| | Hybrid | 5633 | 6144 |
| | Invalidate | 8323 | 6081 |
| | Msg-based | 5543 | 5619 |
| FFT | Sel-Update | 3940 | 6518 |
| | Hybrid | 5916 | 14107 |
| | Invalidate | 6279 | 11384 |
| | Msg-based | 3724 | 6423 |

**Table 1** Communication Statistics - 8 Processors

| Application | Time (secs) |
|---|---|
| IS | 11.68 |
| M-Water | 522.75 |
| SOR | 121.21 |
| FFT | 19.61 |

**Table 2** Uniprocessor Execution Times

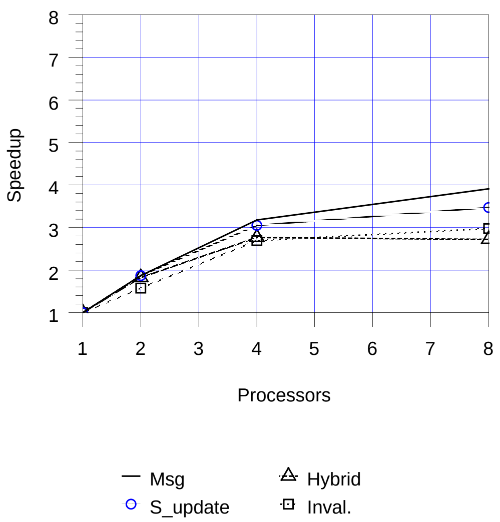


**Figure 7** Integer Sort - Performance

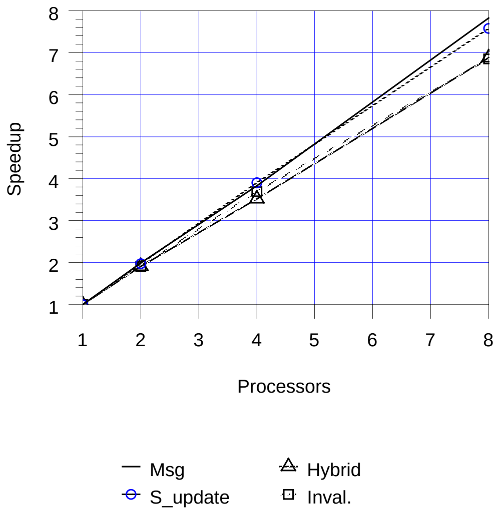


**Figure 8** Water - Performance

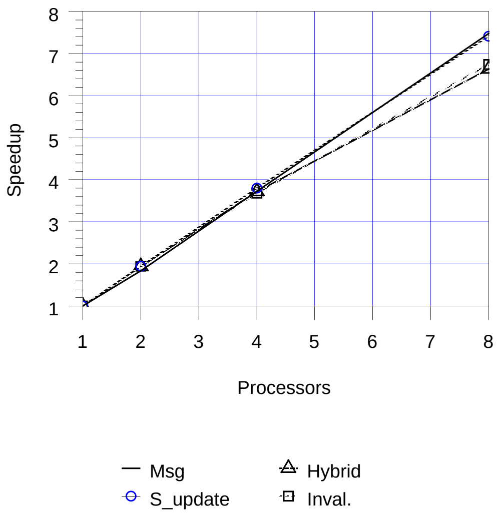


**Figure 9** SOR - Performance

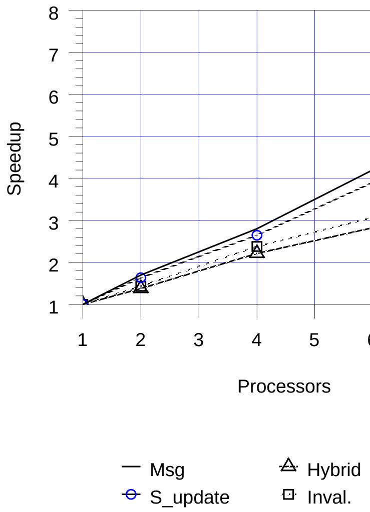


**Figure 10** FFT - Performance

The parallel version of IS divides up the keys among the processors. At first, processors count their keys in their private buckets. In the next phase, the values in the private buckets are summed up using locks to acquire exclusive access to the shared buckets. The data is essentially migratory in this phase. In the final phase, separated by a barrier to ensure that all the processors are done with updating the shared buckets, the processors read the sums in the shared buckets and rank the keys in their own memory.

All the compiler analysis applied was assumed to be localized to between two synchronization points. Both between the lock acquire and release, and between the barriers, the shared data access patterns are easily analyzed. Cross-synchronization prefetching is used in the first case. The same technique is used in the second case, since there is only one shared data structure and no false sharing issue. The use of the partial update technique in this case would result in identical behavior. The barrier itself is not replaced since the last producer of the data can only be determined at runtime.

Figure 7 shows the performance of the four versions of the program. The selective update protocol is able to reduce the number of messages to a little more than half that used by the invalidate protocol (see Table 1). This results in a 16% improvement in performance over the invalidate protocol for this data set.

The message passing version still does about 13% better than the selective update protocol since data is essentially "pushed" to the processor next accessing it instead of being requested and the extra indirect synchronization messages are avoided.

The runtime hybrid protocol performs better than the invalidate protocol for a small number of processors (2-3). However, at larger numbers of processors, it performs worse than the invalidate protocol (9% lower speedup) since it results in a large amount of unnecessary data being sent. Each processor individually identifies the pages that need to be updated at the other processors, and sends them out prior to reaching the barrier. Due to the migratory nature of the shared data in this program, each processor sends out modifications to the others at the barrier, since it does not know who has made the most recent updates. The selective update protocol is able to continue to avoid this, since the modifications are sent out from the most recent updater of the page after the barrier has been reached.

#### 4.3.2 M-Water

M-Water, a modified version of the program from the SPLASH suite [24], is a molecular dynamics simulation. We present results for M-Water using 1000 molecules and run for 5 time steps. The original Water program obtains a lock on the record representing a molecule each time it updates the force vectors in the record's contents. We modified M-Water such that each processor instead uses a local variable to accumulate its updates to a molecule's record during this phase of each iteration. A similar modification was used in the second release of the SPLASH benchmark suitei [25]. At the end of the phase, it then acquires a per processor lock and applies the accumulated updates to the force vectors of the molecules owned by each processor at once. The next phase consists of using these force vectors to determine the new displacements of each molecule.

Each molecule's data structure consists of fields with completely different sharing patterns - single writer - multiple reader, multiple writer - multiple reader, and private. These correspond to the displacement vectors, the force vectors, and the rest of the data structure. Since these varying access patterns are adjacent to each other and are all allocated in shared memory, a severe false sharing problem is incurred. While all the fields are modified in each iteration and therefore communicated, only some of them are actually shared. The intermixing of shared and private data also results in additional access misses.

Since the accesses to each molecule's vectors occur in tight loops, compiler analysis similar to that used in Figure 5 can be applied to take advantage of partial update. However, in the case of Water, global analysis across multiple synchronization points, specifically across locks accessed between barriers, is required. In this case, the fact that only the force and displacement vectors are actually shared can be recognized, and only those fields can be updated.

If only local analysis (within two synchronization points) is applied, the only optimization that can be applied to this program is cross-synchronization prefetching. Using cross-synchronization prefetching benefits this application by replacing multiple miss messages with a single update, reducing the number of messages by 35% percent over the invalidate protocol. However, due to the large amount of data transferred using this optimization alone and, similarly, using the runtime hybrid protocol, packets are lost and therefore no improvement in performance is gained. For this same reason (packet loss), the miss rate is reduced only to a third of that for the invalidate protocol.

Applying more global analysis and using partial update, the performance is improved and brought close to that of the hand-tuned message passing version. The selective update protocol performs 11% better than the invalidate protocol. The invalidate and hybrid protocols must be conservative in keeping memory consistent. Hence, all the fields, including those that are accessed by only one processor, are communicated. This results in almost 5 times as much data being communicated.

#### 4.3.3 SOR

Red-Black Successive Over-Relaxation (SOR) is a method for solving partial differential equations. We ran SOR for 100 iterations on a $1024 \times 1024$ matrix. The SOR program divides the matrix into roughly equal size bands of consecutive rows, assigning each band to a different processor. Communication occurs across the boundary between bands. Barriers are used to synchronize the processes, between iterations and between the two phases of an iteration.

A compiler can easily determine the producer-consumer nature of the data exchange at the barrier, or, in other words, it can exactly determine what data is going to be accessed by a processor after the barrier and which processor produced the data before the barrier. Applying the partial update method improves the performance over the invalidate protocol and makes it comparable (within 2% of the performance) to that of a program directly written using the underlying message passing (see Figure 9) since the number of messages is roughly the same.

Although the hybrid protocol is able to capture and optimize for the access pattern of this application, both the invalidate and the hybrid runtime protocols perform about 10% worse than the message-based program because of the extra messages due to separate synchronization and data transfers, as well as due to the diff creation consistency actions (see Table 1). The consistency overhead is avoided with the use of the selective update protocol.

#### 4.3.4 3-D FFT

The three-dimensional fast fourier transform program (3-DFFT) is also a program from the NAS suite. We ran 3-D FFT for a $32 \times 64 \times 32$ data set for 6 iterations. 3-D FFT numerically solves a certain partial differential equation using three dimensional forward and inverse FFT's. The program essentially performs several one dimensional FFTs in each direction. Each processor is assigned a contiguous section of the array, assigned along the first dimension.

The program therefore involves a transpose phase to improve access locality between direction changes. An inherent problem with the transpose in the shared memory approach is the false sharing within a page that has been completely modified by the process producing the data but is only partially needed by the process consuming the data (multiple reader false sharing). The selective update protocol provides only partial updates during the transpose. Consistency elimination is used for the write-first initialization phase.

The selective update protocol eliminates false sharing for this program, reduces consistency maintenance overhead, and reduces the amount of data transferred. The performance of the application is improved by 25% when compared to the invalidate protocol and is much closer to the message passing version. Roughly 75% of this improvement comes from the use of partial update, while 25% comes from the use of consistency elimination. Both the message passing and the partial update protocol pipeline the data in order to avoid buffer overflow and overlap communication with computation.

As borne out by Figure 10, the shared memory performance suffers, both for the invalidate and the runtime hybrid protocol due to the false sharing within a page accessed by any processor. The false sharing results in almost double the amount of data being transferred. The runtime hybrid protocol loses performance because of the large amount of data it sends out at the same time without allowing overlapped computation and communication. All data structures are updated at every phase regardless of whether they are used in that phase, resulting in more data being sent than in the invalidate protocol, and hence poorer performance.

# 5 Related Work

Distributed shared memory systems [20] have gained popularity because they hide the underlying details of message passing and data communication from the user. Implicit parallelism, as in HPF [13], achieves the same goal by relying on user-provided data distributions which are then used by the compiler to generate message passing code. This approach is suitable for data-parallel programs, such as SOR. Programs exhibiting dynamic parallelism, such as TSP, are not easily expressed in the HPF framework.

Two approaches to integrating message-passing and shared-memory in a hardware coherent environment are Alewife [17] and FLASH [18]. Both try to exploit the advantages of bulk data transfer afforded by message passing, and the techniques described here could be used on these machines as well. Carlos [16] is a distributed shared memory system that integrates support for message passing with and without coherence. This paper deals with methods of using the underlying message passing to implement various types of updates with the help of compiler assistance.

Tempest [23] is an interface that permits programmers and compilers to use hardware communication facilities and to modify the semantics and performance of shared-memory operations. Our techniques concentrate on developing methods to reduce the amount of data transferred and the number of messages communicated by a program.

Granston and Wijshoff [12] present techniques to perform data layout optimizations for distributed shared memory in order to reduce false sharing. These techniques work in conjunction with those described here for false sharing within a data structure. Similarly, the work in [21] describes techniques by which DSMs may be improved as a compilation target. Our work differs in that we deal with explicitly parallel programs and identify situations in which consistency overhead and false sharing can be eliminated, and communication can be aggregated.

Mowry et al. [22] discuss the design and evaluation of a compiler algorithm for prefetching. Their algorithm concentrates on improving the performance of cache-based systems and issues prefetch requests for data that are likely to incur a cache miss. Porterfield et al. [5] present an algorithm for inserting prefetches one loop iteration ahead. Gornish, Granston, and Veidenbaum [11] present an algorithm for determining the earliest time when it is safe to prefetch shared data. Our work differs in the granularity of information required, and takes advantage of the software-based consistency maintenance. Our optimizations perform prefetching and updating with a view to exploiting the explicit synchronization in relaxed consistency models.

## 6 Conclusions

In this paper, we present a compiler-directed selective update protocol for software distributed shared memory. The goal is to reduce the cost of communication, by aggregating messages, by combining synchronization and data transfer, by reducing the effects of false sharing, and by eliminating the communication of unnecessary data. The protocol uses three different selective update techniques - cross-synchronization prefetching, partial updates, and consistency elimination. Compiler analysis is used to direct the application of these mechanisms. We have provided the runtime support necessary to implement these techniques and demonstrated the feasibility of the compiler analysis required to make use of them. We evaluated the effectiveness of these techniques on four hand-compiled programs and showed a 10-35% improvement in performance when compared to the base invalidate protocol. The performance is consistently better than the base invalidate protocol, in contrast to the runtime-based hybrid protocol, which sometimes performs worse. These mechanisms are seen to effectively combat the underlying flaws in software distributed shared memory systems by taking advantage of the native message passing communication style.

## References


[1] S. Adve and M. Hill. Weak ordering: A new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 2–14, May 1990.

[2] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS parallel benchmarks. Technical Report TR RNR-91-002, NASA Ames, August 1991.

[3] V. Balasundaram and K. Kennedy. A technique for summarizing data access and its use in parallelism enhancing transformations. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, June 1989.

[4] D. Callahan. *A Global Approach to Detection of Parallelism.* PhD thesis, Rice University, March 1987.

[5] D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. In *Proceedings of the 4th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 40–52, April 1991.

[6] J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 152–164, October 1991.

[7] P. Dasgupta, R.C. Chen, S. Menon, M. Pearson, R. Ananthanarayanan, U. Ramachandran, M. Ahamad, R. LeBlanc Jr., W. Applebe, J.M. Bernabeu-Auban, P.W. Hutto, M.Y.A. Khalidi, and C.J. Wileknloh. The design and implementation of the Clouds distributed operating system. *Computing Systems Journal*, 3, Winter 1990.

[8] S. Dwarkadas, P. Keleher, A.L. Cox, and W. Zwaenepoel. Evaluation of release consistent software distributed shared memory on emerging network technology. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 244–255, May 1993.

[9] B. Fleisch and G. Popek. Mirage: A coherent distributed shared memory design. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 211–223, December 1989.

[10] 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, May 1990.

[11] E. Gornish, E. Granston, and A. Veidenbaum. Compiler-directed data prefetching in multiprocessors with memory hierarchies. In *Proceedings of the 1990 International Conference on Supercomputing*, 1990.

[12] F. Granston and H. Wijshoff. Managing pages in shared virtual memory systems: Getting the compiler into the game. In *Proceedings of the 1993 ACM International Conference on Supercomputing*, July 1993.

[13] High Performance Fortran Forum. High Performance Fortran language specification. *Scientific Programming*, 2(1-2):1–170, 1993.

[14] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 13–21, May 1992.

[15] P. Keleher, S. Dwarkadas, A. Cox, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the 1994 Winter Usenix Conference*, pages 115–131, January 1994.

[16] P.T. Koch, R.J. Fowler, and E. Jul. Message-driven relaxed consistency in a software distributed shared memory. In *Proceedings of the First USENIX Symposium on Operating System Design and Implementation*, pages 75–86, November 1994.

[17] D. Kranz, K. Johnson, A. Agarwal, J. Kubiatowicz, and B. Lim. Integrating message-passing and shared-memory: Early experience. In *Proceedings of the 1993 Conference on the Principles and Practice of Parallel Programming*, May 1993.

[18] J. Kuskin and D. Ofelt et al. The Stanford FLASH multiprocessor. In *Proceedings of the 21th Annual International Symposium on Computer Architecture*, April 1994.

[19] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[20] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[21] R. Mirchandaney, S. Hiranandani, and A. Sethi. Improving the performance of software dsm systems via compiler involvement. In *Proceedings Supercomputing '94*, 1994.

[22] T.C. Mowry, M.S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the 5th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 62–75, October 1992.

[23] Steven K. Reinhardt, James R. Larus, and David A. Wood. Tempest and typhoon: User-level shared memory. In *Proceedings of the 21th Annual International Symposium on Computer Architecture*, April 1994.

[24] J.P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared-memory. Technical Report CSL-TR-91-469, Stanford University, April 1991.

[25] J.P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared-memory. *Computer Aarchitecture News*, 20(1):2–12, March 1992.

[26] C-W. Tseng. *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines.* PhD thesis, Rice University, Houston, January 1993.

[27] M.J. Zekauskas, W.A. Sawdon, and B.N. Bershad. Software write detection for distributed shared memory. In *Proceedings of the First USENIX Symposium on Operating System Design and Implementation*, pages 87–100, November 1994.